

1-1-2011

High performance computing for linear acoustic wave simulation

Fouad Butt
Ryerson University

Follow this and additional works at: <http://digitalcommons.ryerson.ca/dissertations>



Part of the [Theory and Algorithms Commons](#)

Recommended Citation

Butt, Fouad, "High performance computing for linear acoustic wave simulation" (2011). *Theses and dissertations*. Paper 591.

This Thesis is brought to you for free and open access by Digital Commons @ Ryerson. It has been accepted for inclusion in Theses and dissertations by an authorized administrator of Digital Commons @ Ryerson. For more information, please contact bcameron@ryerson.ca.

HIGH PERFORMANCE COMPUTING FOR LINEAR ACOUSTIC WAVE SIMULATION

by

Fouad Butt

BSc., Ryerson University, Toronto, Ontario, 2009

A thesis

presented to Ryerson University

in partial fulfilment of the
requirements for the degree of
Master of Science
in the program of Computer Science
Ryerson University

August 2011

Toronto, Ontario, Canada, 2011

©Fouad Butt 2011

Author's Declaration

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Signed: _____

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signed: _____

Abstract

Parallel computing techniques are applied to a linear acoustic wave model to reduce execution time. Three parallel computing models are developed to parallelize computations. The fork-and-join, SPMD and SIMT models define the execution of parallel computations. The precision and efficiency of the linear acoustic wave model are improved through substantial speedups in all implementations.

Furthermore, axisymmetric properties of certain acoustic fields lead to a reduction in the spatio-temporal complexity of those acoustic fields by removing redundant computations. The same linear acoustic wave model is also modified and extended to describe wave propagation across multiple media instead of only a single medium.

The developed implementations are integrated into a particularly useful package for high performance simulation of two- or three-dimensional linear acoustic fields generated by realistic sources in various fluid media.

Acknowledgements

I would like to first extend my gratitude towards my supervisors Dr. Abdolreza Abhari and Dr. Jahan Tavakkoli. Without their invaluable insight and continuous assistance, this work would not have been possible.

Funding for this work was provided partly by the research account of Dr. Abdolreza Abhari from the Department of Computer Science at Ryerson University. Financial support was also provided through a scholarship from the Department of Computer Science at Ryerson University.

Additional monetary support was also obtained through positions as a Graduate Assistant and as an exam invigilator for various courses at Ryerson University within the Department of Computer Science.

Partial funding was also provided by the Natural Sciences and Engineering Research Council (NSERC) Discovery Grant and Ryerson Research Start-up Funds that were awarded to Dr. Jahan Tavakkoli who is a faculty member of the Department of Physics at Ryerson University.

This work was also made possible by the facilities provided by the Shared Hierarchical Academic Research Computing Network (SHARCNET)[1], Réseau Québec de Calcul de Haute Performance (RQCHP)[2] and Compute/Calcul Canada[3].

I would also like to thank Mosa Alhamami and Negar Zohouri from the Department of Physics for their diligence and helpfulness in testing some of the implemented algorithms.

Additionally, I would like to thank John Nickolls of NVIDIA for allotting time to clarify certain aspects of the Tesla unified computing architecture.

Lastly, and perhaps most importantly, I must acknowledge the unending support from my family but especially my parents to whom I dedicate this work.

Contents

1	Introduction	1
1.1	Motivation	5
1.2	Objectives and Scope	6
1.3	Thesis Contributions	7
1.4	Outline	8
2	Background	11
2.1	An Introduction to Linear Acoustics	12
2.1.1	Diffraction in Linear Acoustics	12
2.1.2	The Linear Acoustic Wave Model by Ocheltree and Frizzell	14
2.2	Bioheat Transfer Equation	17
2.3	An Introduction to Performance Enhancement Methods	18
2.3.1	Computer Architectures	19
2.3.2	Data-level Parallelism	21
2.4	Open Multi-Processing	23
2.5	Single Program Multiple Data Computation	24
2.6	Graphics Processing Units	26
2.6.1	Computational Intensity	26
2.6.2	Tesla Unified Computing Architecture	27
2.6.3	Thread Warps	29
2.6.4	Parallel Computing on The Tesla Architecture	30
2.6.5	Compute Unified Device Architecture	32
2.6.5.1	Kernels	32
2.6.5.2	Threads	33
2.7	Literature Review	35
2.7.1	Applications of Multithreading and Cluster Computing	35
2.7.2	General-Purpose Computing on Graphics Processing Units	40
2.7.3	Applications of Graphics Processing Units	42
3	Parallel Computing Models, Algorithms and a Multi-layer Linear Acoustic Model	45
3.1	Sequential Computation	46

3.2	Reducing Redundancies	48
3.3	Workload Definition and Decomposition	49
3.4	Fork-and-join Model and Algorithm	51
3.5	SPMD Model and Algorithm	53
3.6	SIMT Model and Algorithm	56
	3.6.1 Device and Host Code Separation	57
	3.6.2 Calculation Volume Mapping	59
3.7	Multi-layer Linear Acoustic Model	61
3.8	Summary	65
4	Parallelized Implementations, Multi-layer Simulation and The LATS	
	Software Package	67
4.1	Performance Metrics	69
4.2	Multithreaded and Cluster Computing	70
4.3	Graphics Processing Units	74
4.4	Multi-layer Simulation	80
4.5	Linear Acoustic and Temperature Simulator	83
4.6	Configuration	84
4.7	Create Intensity Field	86
	4.7.1 Setup annular array	89
4.8	Apply BHTE	90
4.9	Plots	93
4.10	Summary	96
5	Concluding Remarks	97
5.1	Future Work	99
	Bibliography	101

List of Tables

4.1	Relevant simulation parameters and their respective values* for each of the three experiments.	68
4.2	Sequential, multithreaded and cluster computation times with their respective Karp-Flatt metric values for 8 and 64* threads without quarter-field computation.	72
4.3	Sequential, multithreaded and cluster computation times with their respective Karp-Flatt metric values for 8 and 64* threads with quarter-field computation.	74
4.4	Sequential and GPU computation times with their respective Karp-Flatt metric values.	77

List of Figures

1.1	An illustration of the acoustic field or the calculation volume in front of the acoustic source.	4
2.1	A geometric illustration of an arbitrarily-shaped acoustic source vibrating into a homogeneous, isotropic, non-dissipative medium. The figure is adapted from [4].	13
2.2	Figure 2.2a depicts the simulation parameters in Equation (2.2). Figure 2.2b describes the coordinate system for a single element in the acoustic source. In Figure 2.2b, a single element is composed of smaller sub-elements.	16
2.3	OpenMP thread creation and merging	23
2.4	The Tesla graphics processing unit (GPU) architecture	28
2.5	Two-dimensional and three-dimensional views of the logical organization of threads, blocks and grids in the CUDA programming model.	34
3.1	Axial-symmetric workload division	48
3.2	Distribution of workload in the FJ model.	51
3.3	Distribution of workload through multi-processing and multithreading in the SPMD model.	54
3.4	Device-host or GPU-central processing unit (CPU) code separation in the single instruction-multiple thread (SIMT) model.	58
3.5	Division of points in a single plane of the calculation volume	60
3.6	Geometric depiction of simulation with two adjacent media	64
3.7	Steps involved in multi-layer computations with the considered linear acoustic model	65
4.1	Sequential, multithreaded and cluster computation times for step sizes and element sizes of 0.5, 0.2 and 0.1 without quarter-field computation enabled.	73
4.2	Sequential, multithreaded and cluster computation times for step sizes and element sizes of 0.5, 0.2 and 0.1 with quarter-field computation enabled.	75
4.3	A comparison between the GPU execution times and the execution times of the other implementations.	76

4.4	A comparison of peak intensity values observed with the single layer model and the multi-layer model at varying frequencies and a fixed interface location.	82
4.5	Plots depicting interface locations and the peak intensity values observed at those interface locations.	82
4.6	An empty (4.6a) and complete (4.6b) configuration page in the simulation GUI.	85
4.7	Default parameter values on the <i>Create intensity field</i> page and notification when simulation is complete.	89
4.8	Annular array setup dialog box with sample values for the phase angles and amplitudes of 7 annuli.	90
4.9	Default parameter values on the <i>Apply bioheat transfer equation (BHTE)</i> page and notification when simulation is complete.	93
4.10	Plotting features	94
4.11	X-Z plane colour map plotting explanation	95

List of Algorithms

1	Sequential field computation	47
2	Field computation in the FJ model	52
3	Field computation in the SPMD model	55
4	Field computation kernel in the SIMT model	58

Acronyms

ANL	Argonne National Laboratory. 37
API	application programming interface. 6, 23, 24, 38, 41
APP	Accelerated Parallel Processing. 41
ASMP	Asymmetric Multi-Processing. 20
BHTE	bioheat transfer equation. xiv, 17, 18, 83, 87, 88, 90–93, 95
BID	block ID. 34, 59
CPU	central processing unit. xiii, 1, 2, 5, 19, 20, 24, 26, 27, 29, 32, 37, 41, 43, 44, 50, 52, 54, 57, 58, 66, 67, 71, 72, 74, 84, 96–98
CTA	cooperative thread array. 30–32, 59
CUDA	Compute Unified Device Architecture. 11, 30, 32, 34, 41, 43, 46, 56, 59, 60, 66, 75, 76
CW	continuous wave. 14, 16
DRAM	Dynamic Random Access Memory. 28, 45, 59, 75, 77, 100
EPEX	Environment for Parallel EXecution. 25
FBA	flux balance analysis. 37
FJ	Fork-and-join. 46, 50–53
FLOPS	FLloating point OPERations per Second. 26
FSB	front-side bus. 20
GDDR3	Graphics Double Data Rate 3. 75
GLSL	Open Graphics Library (OpenGL) Shading Language. 43

GPR gene-protein-reaction. 37
 GPU graphics processing unit. xiii, 1, 2, 8, 11, 22, 26–29, 31–33, 35, 40, 41, 43, 44, 46, 56–59, 66, 67, 74–80, 83, 85, 96, 98–100
 GUI Graphical User Interface. 8, 35, 83–85

 HDD hard disk drive. 55, 100
 HIFU High Intensity Focused Ultrasound. 5, 96
 HPC high performance computing. 37, 38, 61, 67, 70, 76, 80, 96, 97, 100

 ICC Intel C/C++ Compiler. 71
 IPC inter-process communication. 6, 49, 51, 53, 55
 ISL Interactive Shading Language. 40
 ITC inter-thread communication. 6

 KZK Khokhlov-Zabolotskaya-Kuznetsov. 42, 100

 LATS Linear Acoustic and Temperature Simulator. 5, 8, 9, 17, 83, 99

 MAD multiply-add unit. 29
 MIMD multiple instruction-multiple data. 19–21
 MISD multiple instruction-single data. 19
 MP multi-processor. 25
 MPI message passing interface. xviii, 38, 39
 MT multi-threaded issue unit. 29

 NSERC Natural Sciences and Engineering Research Council. vii
 NUMA non-uniform memory access. 20

 OpenCL Open Computing Language. 32, 41, 43
 OpenGL Open Graphics Library. xvii, 40, 43
 OpenMP Open Multi-Processing. 11, 22–24, 26, 38, 52, 53, 97
 OpenMPI Open message passing interface (MPI). 38, 53
 ORB orthogonal recursive bisection. 39, 49

PARC	Palo Alto Research Institute. 26
PCIe	Peripheral Component Interconnect Express. 26, 75
PDE	Partial Differential Equation. 43
POSIX	Portable Operating System Interface for Unix. 24
PVM	Parallel Virtual Machine. 35
PW	pulsed wave. 14
RAM	Random Access Memory. 55, 71
ROC	radius of curvature. 16
ROP	Raster Operation Processor. 28, 29
RQCHP	Réseau Québec de Calcul de Haute Performance. vii, 70, 71, 76
SDK	Software Development Kit. 41
SFU	special function unit. 29
SHARCNET	Shared Hierarchical Academic Research Computing Network. vii, 76
SIMD	single instruction-multiple data. 19–21, 40
SIMT	single instruction-multiple thread. xiii, 22, 29–31, 47, 56–59, 66, 67
SISD	single instruction-single data. 19
SM	Streaming Multi-processor. 28, 29, 31, 32, 75, 79, 80, 98
SMC	streaming processor controller. 29, 31
SMP	Symmetric Multi-processing. 19, 20, 50, 52, 53, 67, 97, 98
SMT	Symmetric Multi-threading. 38
SP	Streaming Processor. 29, 30, 32, 75
SPMD	single process-multiple data. 11, 22, 24–26, 35, 46, 50, 51, 53–55, 67
SSH	Secure Shell. 76
TID	thread ID. 21, 22, 31, 33, 59
TPC	Texture/Processor Cluster. 28, 29

UMA uniform memory access. 20

VM virtual machine. 25

Chapter 1

Introduction

Processing units with multiple cores are now common due to advances in technology. CPUs composed of multiple cores provide an indication of a trend in performance enhancement towards greater throughput as opposed to faster processor clock cycle speed [5, 6]. The trend sways away from extracting performance in a single stream of instructions by maximizing instruction-level parallelism. Instead, multiple core architectures address the issue of execution speed by increasing throughput and executing multiple instruction streams simultaneously.

Aside from CPUs with multiple cores, the development of GPUs and their application to areas outside of graphics provides an alternative means to increasing execution speed. Pivotal to the engineering decisions in the design of the GPU is the allocation of the area on a GPU chip to hundreds of simple execution units as opposed to fewer complex execution units found in CPUs. Multiple execution units in the GPUs are also controlled by a single control unit, such that a single instruction is issued to many execution units simultaneously. Cache memory in a GPU is also limited when compared to cache memory on a CPU. The reason for these features in the design of GPUs is rooted in the nature of graphics where greater throughput is desired for

multiple highly independent operations that may be performed in parallel.

Both CPUs and GPUs rely on the concept of threads as a unit of parallel execution. Threads are a unit of execution that require fewer resources than processes as a large portion of execution state information is shared across all threads. Multiple threads and multiple processes in a parallel computing model are the units of execution to realize the full utility of computer architectures with multiple cores, which execute multiple instruction streams concurrently.

A study of applications of parallel computing provides an indication of the large body of research related to leveraging the computational capabilities of parallel computing architectures to enhance the performance of scientific applications. Such applications typically adopt parallel programming models to develop implementations on parallel computing architectures. Parallel execution models describe the method of execution and the program structure followed by algorithms implemented on parallel computing architectures such as multi-core CPUs and GPUs.

In terms of forms of parallelism, data-level parallelism is present in an algorithm if the computation of the algorithm for one set of inputs is independent of the computation of the same algorithm for another set of inputs [7]. Such an algorithm is deemed to exhibit data-level parallelism, which may be exploited with parallel computing models to reduce the execution time.

The purpose of the current work is to study the extent of performance enhancements drawn from parallel computing models on two types of computer architectures within the context of a highly data-parallel linear acoustic wave simulation algorithm. The two types of computer architectures include traditional CPUs and newer GPUs. The linear acoustic wave model that is simulated is developed by Ocheltree and Frizzell in [8] and is selected for two reasons: (i) the model is relatively simple to understand; and (ii) parallel computing models are hypothesized to produce large

speedups.

It is instructive to describe the simulation process of the linear acoustic wave model by first introducing some general terminology in the field of acoustics.

As waves propagate, they sometimes exhibit a phenomenon known as diffraction, which is the distortion of the wave pattern as the waves propagate past an obstacle. In the case of the current model, the waves are diffracted as they travel from an *acoustic source*.

The physical acoustic source is composed of piezoelectric material, which, when excited with an electric current, vibrates rapidly to generate mechanical waves [9]. Because an acoustic source converts electrical energy into mechanical energy, it is also referred to as a transducer, which generally includes any device that converts one form of energy into another [9]. Of course, in the context of the current work, the terms transducer and acoustic source are interchangeable and shall refer only to piezoelectric crystals that generate acoustic waves.

The function of the linear acoustic wave model is to describe the wave propagation patterns observed when an acoustic source of arbitrary shape vibrates and generates mechanical waves. The patterns are observed in a medium, such as water or tissue, as the mechanical waves propagate through that medium. The type of medium also affects the pattern of wave propagation that is observed.

A subset of the values provided as input to the linear acoustic wave model is a set of coordinates in three-dimensional space and the output produced is the intensity at those coordinates. When the intensity values at multiple points are computed within a three-dimensional rectangular prism in front of the acoustic source, the rectangular prism forms what is called the *acoustic field* or the *calculation volume* as depicted in Figure 1.1.

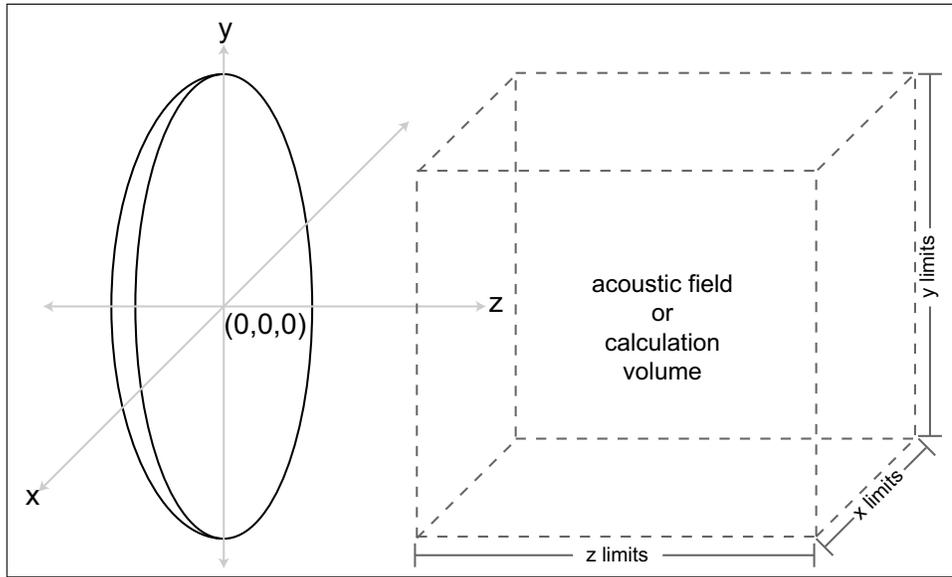


Figure 1.1: An illustration of the acoustic field or the calculation volume in front of the acoustic source.

Simulation of the linear acoustic wave model to produce a single intensity value involves an integration over the surface area of the acoustic source. The acoustic source may be considered as composed of several infinitesimal points. Consequently, two general factors contribute significantly to the spatio-temporal complexity of the linear acoustic wave model:

1. The first factor is the computational workload of integrating over the surface area of the acoustic source to obtain the intensity value at a *single* point in the calculation volume. This is because of the number of points in the composition of the acoustic source.
2. The second factor is due to multiple integrations over the surface area of the acoustic source to obtain multiple intensity values at *multiple* points in the calculation volume.

Both factors are adjustable and controlled. The number of points that compose the

acoustic source and the number of points in the calculation volume may be increased to improve the accuracy of the simulation results.

Increasing precision in this manner results in a greater computational workload and increased execution time. However, the numerical integration of a point in the acoustic field is independent of that same numerical integration for another point in the acoustic field. Thus, the algorithm exhibits data-level parallelism such that computations of intensity values at different points in the calculation volume can be performed independently.

1.1 Motivation

The linear acoustic wave simulation considered in the current work requires more than six hours to compute the acoustic field generated by an acoustic source if points are computed sequentially on an Intel Xeon E5462 2.8 GHz quad-core CPU. Evaluation of the model results in the identification of inherent data-level parallelism in the simulation process. It is hypothesized that speedups equal to the number of execution units in the parallel computing architecture are possible. There is minor synchronization overhead and no communication between threads or processes.

Due to the impractical length of execution time and the potentially large reductions in execution speed, parallel execution models and architectures are sought to enhance the performance of the linear acoustic wave simulation.

Furthermore, Linear Acoustic and Temperature Simulator (LATS) [10] is developed as a software package for the simulation and visualization of acoustic wave patterns. LATS will be useful in the development of transducers in ultrasound devices that find diagnostic and therapeutic applications in biomedical fields.

For example, High Intensity Focused Ultrasound (HIFU) is an ultrasound treat-

ment modality that is particularly useful for operating on tumours through non-invasive surgery [11, 12]. By creating a focused ultrasound beam with a region of high intensity at the focal spot, lesions may be created at the location of the tumour underneath the skin to essentially “cook” the tumour without damaging skin tissue through invasive surgery [11, 12].

1.2 Objectives and Scope

Primarily, the aim of the current work is to enhance the performance of the linear acoustic wave model presented by Ocheltree and Frizzell in [8] by developing parallel computing models that adopt different execution models on two types of parallel computing architectures. While it is known that parallel computation will definitely yield certain benefits, the extent of these benefits utilizing different parallel execution models and architectures is not known.

In the current work, synchronization of threads is present in the multithreaded and cluster computing implementations but automatically handled by the application programming interface (API) for multithreading. No inter-process communication (IPC) or inter-thread communication (ITC) is utilized in any of the implementations as sharing of information between threads and processes is not necessary due to the nature of the linear acoustic wave simulation. Because no communication is required and synchronization is handled by the API, the various communication and synchronization models are beyond the scope of the current work.

Memory models are mentioned briefly as each parallel computing architecture herein features a different memory model. However, the focus of the work remains the reduction of execution speed of linear acoustic wave simulation.

In the study of wave propagation, there are linear and nonlinear acoustic wave

models. Two main approaches may be described for linear acoustic wave models: direct numerical computation and convolution-based. The linear acoustic wave model in the current work fits in the category of direct numerical computation approaches.

Furthermore, while other linear acoustic wave models exist, the physical aspects of modelling wave propagation are limited to the model by Ocheltree and Frizzell in [8].

1.3 Thesis Contributions

To improve the execution speed of linear acoustic wave simulation, implementations relying on parallel computation are developed. The acoustic field is decomposed and the integrations of sets of points are executed in parallel to enhance the performance of the simulation. The major contributions of this work can be summarized as the following:

Reduced computations

For acoustic sources that produce axisymmetric acoustic fields, redundant computations are removed by computing only one quarter of the acoustic field and copying the intensity values at those points and filling the remaining three quarters of the acoustic field with those intensity values. The process is termed *quarter-field computation*.

Three parallel computing models

The problem of impractical execution times for linear acoustic wave simulation is addressed with three implementations that rely on parallel computing. As far as is known, the model by Ocheltree and Frizzell [8] has not been previously enhanced with parallel computing. Each implementation results in marked per-

formance improvement in terms of reductions in execution times.

Multi-layer simulation

Preliminary results are also developed for multiple layer simulations through an extension of the same linear acoustic wave model.

LATS software package

A Graphical User Interface (GUI) is also developed in MATLAB to configure and execute the linear acoustic wave simulation and visualize the results. The inclusion of the multithreaded implementation results in a complete software package termed LATS.

1.4 Outline

Chapter 2 develops the background for the current work beginning with an explanation of the physical concepts underlying linear acoustic wave models, which is followed by a description of the particular linear acoustic wave model considered for performance enhancement.

Delving into performance enhancement, the discussion continues in Chapter 2 with an introduction to basic computer architectures and data-level parallelism, which leads into research efforts undertaken by researchers to exploit parallelism utilizing multithreading, cluster computers and GPUs as part of the literature review.

The adoption of parallel computation models for linear acoustic wave simulation is described in Chapter 3. The single layer linear acoustic wave propagation model by Ocheltree Frizzell [8] is extended to multiple layers in this work. The multi-layer model is presented in Chapter 3.

Implementations and the associated results from the linear acoustic wave simula-

tion on different parallel computing architectures using different parallel programming models are presented in Chapter 4. Also in Chapter 4, is the implementation of the multi-layer linear acoustic wave propagation model and a description of the LATS software package. Chapter 5 concludes this work and presents possible avenues for future research.

Chapter 2

Background

Section 2.1.1 in this chapter presents a discussion of fundamental developments in the study of diffraction in linear acoustics. Subsequently, a detailed description of the linear acoustic wave propagation model considered for performance enhancement is presented in Section 2.1.2 and features of the model are considered for performance enhancement. Section 2.3 defines basic computer architectures and data-level parallelism. Open Multi-Processing (OpenMP) and the single process-multiple data (SPMD) parallel computing model are explained in Sections 2.4 and 2.5.

GPUs are discussed in Section 2.6. Subsections 2.6.1-2.6.5 describe a metric known as *computational intensity* [13], the Tesla GPU hardware architecture and finally the Compute Unified Device Architecture (CUDA) abstraction of the GPU hardware. The chapter is concluded with Section 2.7, which contains efforts by other researchers that relate to the current work.

2.1 An Introduction to Linear Acoustics

The field of acoustics pertains to the study of mechanical waves in various types of media, which may include solids, liquids or gases. In the present work, mechanical waves propagate as longitudinal waves and are confined to the case of fluid media. Furthermore, various types of wave phenomena, while applicable to other types of waves (electromagnetic, for instance), are henceforth considered in the context of mechanical waves. Therefore, in the current work, the terms wave and mechanical wave are interchangeable.

A particular phenomenon, known as diffraction, is observed when a wave continues to propagate through an aperture or encounters a change in media. The formation of the diffraction pattern is described by the Rayleigh diffraction integral [4]. Sommerfeld applied a Green's function approach to the Rayleigh diffraction integral, which results in the Rayleigh-Sommerfeld integral [4].

2.1.1 Diffraction in Linear Acoustics

The Laplace transform of the velocity potential at a point P situated in a homogeneous, isotropic, non-dissipative medium for a piston source with a uniform velocity distribution vibrating in an infinitely rigid baffle is described as [4]:

$$\Phi(r, s) = \int_S \frac{V_n(r_0, s)e^{-sR/c}}{2\pi R} dS, \quad (2.1)$$

where the integration is performed over the surface area S of the transducer, V_n is the particle velocity normal to the transducer surface, R is the distance from a spatial point P to an infinitesimal surface element dS on the transducer in the plane $z = 0$ and c is the speed of sound in the medium. r and r_0 describe the distance

from a point P to the origin and the distance from an infinitesimal surface element dS to the origin, respectively. The variable s refers to the complex angular frequency resulting from the application of the Laplace transform. A geometric illustration of this type of acoustic source model as adapted from [4] is presented in Figure 2.1.

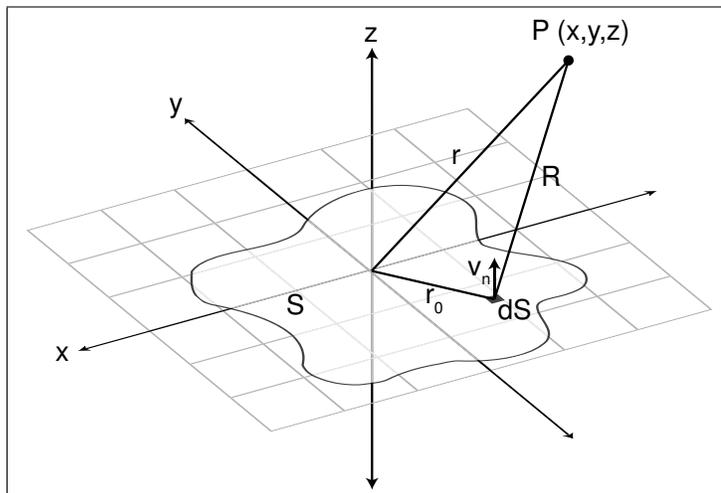


Figure 2.1: A geometric illustration of an arbitrarily-shaped acoustic source vibrating into a homogeneous, isotropic, non-dissipative medium. The figure is adapted from [4].

Equation (2.1) is in the frequency domain and is a commonly applied form of the Rayleigh-Sommerfeld diffraction integral. Noted by Harris in [4], Fresnel was the first to elaborate on the application of Huygens principle to approximate the Rayleigh-Sommerfeld diffraction integral. In consequence of the Fresnel-Huygens principle, a physical interpretation of the Rayleigh-Sommerfeld diffraction integral is that of a source composed of infinitely many points, each producing a wavelet shaped as a sphere. The field at any point may then be calculated by superimposing the waveform contribution from each point source [4]. Analysis by Helmholtz and Kirchoff led to the mathematical expression of the Helmholtz-Kirchoff integral, sometimes also known as the Fresnel-Kirchoff integral, which is a generalized form of the Rayleigh-Sommerfeld diffraction integral [4].

The Rayleigh-Sommerfeld diffraction integral is an analytical solution to model plane wave propagation and the associated diffractive effects in a given medium. Numerical expressions of the integral for square and rectangular sources [8, 14] and circular sources [15] are derived through various approximations.

2.1.2 The Linear Acoustic Wave Model by Ocheltree and Frizzell

The work presented by Stepanishen in [14] is a pulsed wave (PW), time-domain model for acoustic wave propagation. In contrast, the model developed by Ocheltree and Frizzell is a continuous wave (CW), frequency-domain model for acoustic wave propagation. In contrast to a PW model, waves in a CW model maintain a constant amplitude and frequency.

Ocheltree and Frizzell begin with the Rayleigh-Sommerfeld integral and arrive at an expression for the CW pressure amplitude p_0 at a point in three dimensions by enforcing appropriate boundary conditions and applying the Fraunhofer approximation [8]. The method followed by the authors is outlined in this section.

Equation (2.1) is approximated by Ocheltree and Frizzell in [8]. Simulation of the acoustic field is largely based on computation of the approximate expression for the sound pressure amplitude, p_0 , at a point P that is a certain distance from the transducer. The researchers divide the acoustic source into several incremental rectangular areas explaining that when the size of each element composing the acoustic source is small, relative to the distance R from the observation point P , the Fraunhofer approximation may be applied.

The transducer is then represented as an array of elements of width Δw and height Δh . The approximation holds true for each element in the array of elements that form

the transducer. Summation of the individual complex pressures results in the sound pressure amplitude for a given point. The sound pressure amplitude due to an array of N rectangular elements is represented by the following equation [8]:

$$p_0 = \frac{j\rho c\Delta w\Delta h}{\lambda} \sum_{n=1}^N \frac{u_n \exp(-(\alpha+jk)R)}{R} \text{sinc}\left(\frac{kx'_n\Delta w}{2R}\right) \text{sinc}\left(\frac{ky'_n\Delta h}{2R}\right), \quad (2.2)$$

where $j = \sqrt{-1}$, u_n is the velocity amplitude in meters per second (m/s), λ is the wavelength in meters (m), α is the attenuation coefficient in decibels per centimeter megahertz (dB/(cm·MHz)), k is the wave number ($2\pi/\lambda$), ρ is the density of the medium in kilograms per meters cubed (kg/m^3), c is the speed of sound in the medium in meters per second (m/s). The quantity ρc is known as the characteristic acoustic impedance and normally expressed in the units of Newton seconds per meter cubed ($(\text{N}\cdot\text{s})/\text{m}^3$) [9].

As mentioned previously, R is the distance from an element of the acoustic source to a point P in the acoustic field. According to Ocheltree and Frizzell, the Fraunhofer approximation permits the inclusion of only the first two terms of a binomial expansion of R , which is defined in [8] as:

$$R = \sqrt{z^2 + (x - x_n)^2 + (y - y_n)^2}. \quad (2.3)$$

The authors also define x'_n as $x - x_n$ and y'_n as $y - y_n$ and redefine R in [8] as:

$$R = \sqrt{z^2 + (x'_n)^2 + (y'_n)^2}. \quad (2.4)$$

Figure 2.2a depicts a geometrical interpretation of the simulation parameters for a convex circular source. Depicted in Figure 2.2b x_0 and y_0 form a secondary coordinate system, the origin of which, is situated at the center of an element n at (x_n, y_n) . The

direction of x_0 follows the edge of the element with width Δw , while the direction of y_0 lies along the edge of that element with height Δh . R is defined as the distance from the center of an element to the point of measurement. The radius of curvature (ROC) defines the location of the focal spot by adjusting the curvature of the transducer.

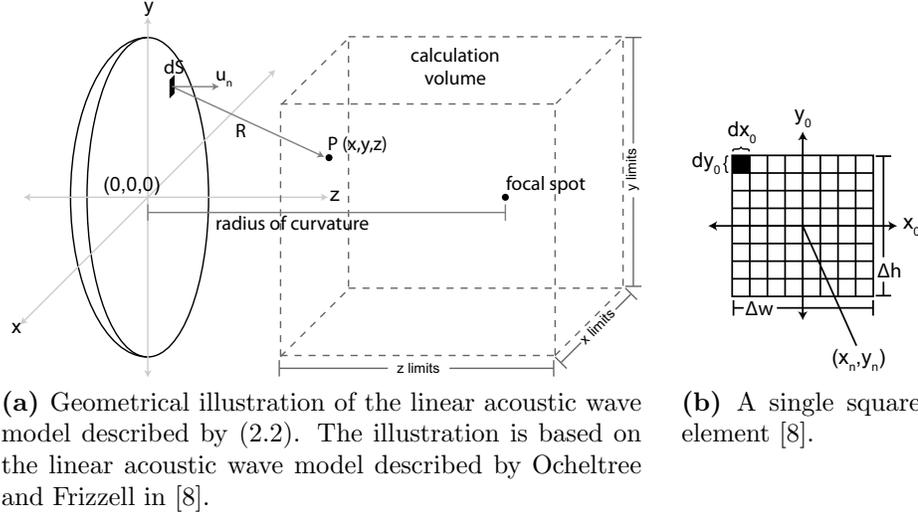


Figure 2.2: Figure 2.2a depicts the simulation parameters in Equation (2.2). Figure 2.2b describes the coordinate system for a single element in the acoustic source. In Figure 2.2b, a single element is composed of smaller sub-elements.

Since the desired metric is time-averaged intensity (W/m^2), a conversion from pressure (Pa) must be performed by applying the following equation:

$$\bar{I} = \frac{(p_0)^2}{(2\rho c)}. \quad (2.5)$$

The equation is valid for a plane CW acoustic wave, where p_0 is the pressure amplitude computed by Equation 2.2, ρ is the medium density in kilograms per meters cubed (kg/m^3), c is the speed of sound in the medium in meters per second (m/s) and ρc is known as the acoustic impedance in Newton seconds per meter cubed ($(N \cdot s)/m^3$) [9]. Rectangles of width Δw and height Δh become larger and fewer in number at further distances along the Z axis according to the following equation [8]:

$$\Delta w \leq \sqrt{\frac{4\lambda z}{F}}. \quad (2.6)$$

Here, F is an experimentally determined factor (set to 20, in most cases), z is the distance along the Z axis and λ is wavelength. An identical expression is applied for Δh .

Three other simplifications, highlighted by Equations (2.7) to (2.9), lead to double integrals that are evaluated by applying the Fourier transform to produce the approximate expression for the pressure amplitude in Equation (2.2) [8].

$$\cos\left(\frac{k(\Delta w^2 + \Delta h^2)}{8R}\right) \approx 1, \quad (2.7)$$

$$\frac{\sqrt{(x' + \Delta w/2)^2 + (y' + \Delta h/2)^2 + (z')^2}}{\sqrt{(x')^2 + (y')^2 + (z')^2}} \approx 1, \quad (2.8)$$

$$\exp(\alpha(x'\Delta w + y'\Delta h)/2R) \approx 1. \quad (2.9)$$

2.2 Bioheat Transfer Equation

A component of LATS is the capability to simulate the effects of heat in tissue media. The temperature simulation process relies heavily on the application of the numerical model developed by Pennes in [16].

Pennes introduces a numerical model which describes heat transport and temperature rise in biological media such as tissue through blood perfusion [16]. The differential equation developed by Pennes is known as the BHTE and is defined in [16] as:

$$\rho C \frac{\partial T}{\partial t} = K \nabla^2 T - W_b C_b (T - T_b) + Q. \quad (2.10)$$

The parameters of the BHTE are given as follows:

ρ - tissue mass density in (kg/m³)

C - tissue specific heat capacity in Joules per kilogram degree Celsius (J/(kg · °C))

T - tissue temperature in degrees Celsius (°C)

K - tissue thermal conductivity in Watts per meter degree Celsius (W/(m · °C))

∇ - spatial Laplacian operator

W_b - blood perfusion rate in kilograms per meter cubed second (kg/(m³ · s))

C_b - blood specific heat capacity in Joules per kilogram degree Celsius (J/(kg · °C))

T_b - blood temperature in degrees Celsius (°C)

Q - heat production rate per unit volume in Watts per meter cubed (W/m³)

2.3 An Introduction to Performance Enhancement Methods

There are many facets to performance enhancement. Depending on the features of a given algorithm, its performance may be improved in terms of memory consumption or execution time. In the case of a distributed algorithm, it may be enhanced by reducing network latency or increasing throughput. Increasing the scalability of such an algorithm may yield further benefits.

The direction undertaken in the present work is to reduce the execution time of the algorithm that computes the acoustic field generated by an arbitrarily-shaped

acoustic source. Pursuit of this goal begins with a discussion of a broad classification of computer architectures, which establishes the background for the parallel programming models presented thereafter.

2.3.1 Computer Architectures

Flynn discusses a taxonomy that broadly encompasses all computer architectures according to the number of data and instruction streams: single instruction-single data (SISD); single instruction-multiple data (SIMD); multiple instruction-single data (MISD); and multiple instruction-multiple data (MIMD) [17]. A single instruction stream needs only one CPU to process the stream, which, coupled with a single data stream proceeds with sequential execution. With multiple data streams, it is possible to access different areas of memory simultaneously. If multiple CPUs are available, multiple instruction streams can be processed simultaneously. SIMD is a technique to apply a single instruction across multiple data. For example, a SIMD capable processor can perform vector addition in one clock cycle, since each component of the vector is a different data stream and addition is the single instruction. A common exemplary implementation of a vector processor is the XP-M by Cray. It should be noted that SIMD processes instructions synchronously, in lockstep [7, p. 32].

None of the architectures are mutually exclusive. Varying levels of abstraction lead to a mixture of architectures. For instance, in a modern CPU with multiple cores, each core may execute instructions in an SIMD manner. If that core supports pipelined execution, multiple SIMD instructions may be executed simultaneously and at that level, it is following MIMD execution.

In terms of memory organization, modern CPUs are generally based on the Symmetric Multi-processing (SMP) architecture. In SMP, a number of processors share

a memory uniformly over a common front-side bus (FSB) and each CPU is identical [7, p. 45]. With the advent of multiple logical cores on one chip, expensive dual socket motherboards are no longer necessary to reap the benefits of multiprocessing. Memory access in SMP architectures is one advantage over of distributed memory architectures. Because each processor has access to a single global memory space, the cost of accessing a particular memory location is the same for all processors. This is representative of uniform memory access (UMA). The disadvantage, however, is that the scalability of such a system is adversely affected when memory contention increases as the number of processors is increased [7, p. 46].

Conversely, SMP systems that employ non-uniform memory access (NUMA) have local memory banks, which can be accessed more quickly than non-local or remote memory banks of the other CPUs [7, pp. 46-47]. This can be advantageous if the affinity of a process is set to one processor to exploit locality of reference. NUMA memory architectures also have the added benefit of better scalability when compared to UMA in UMA systems. Also, a distinction between SMP and MIMD is that while MIMD architectures can have a distributed or shared memory model, SMP generally refers to a system with a single shared memory space.

There also exist Asymmetric Multi-Processing (ASMP) systems, in which each CPU or core is confined to perform a specific set of tasks. For instance, one processor might be responsible only for graphics processing. An example of an ASMP system is the Cell microprocessor, designed by IBM and funded by Sony Corporation [18].

The type of computer architecture and the type of memory model affect the design and implementation of the parallel programming model in software. Consider, for instance, the operation of vector addition. On an SIMD system, the operation may be performed in parallel with SIMD instructions. Each SIMD instruction would add a number of vector components until all components are summed. In contrast,

on an MIMD system incapable of executing SIMD instructions, a large vector sum may be parallelized with multiple threads, where each thread is assigned a fixed number of components to add. Having identified several hardware architectures, it is important to identify the type of parallelism in the algorithm to match the hardware architectures with appropriate parallel computation models.

2.3.2 Data-level Parallelism

When a single task is executed repeatedly and depends largely on input data to obtain matching output data, *data-level parallelism* may be exploited to reduce execution time [7, p. 50]. The input data forms the workload for the task and is distributed over a number of execution units, which may include fibers, threads or processes or combinations of these methods of execution. In contrast, *task-level parallelism* may be identified as individual but different tasks that may be executed simultaneously [7, p. 50]. In particular, data-parallel applications concurrently operate on a distributed set of data, while task-parallel applications distribute multiple unique tasks for simultaneous execution. For instance, consider the following program listings:

Listing 2.3.1: Data-level parallelism

```

1      ⋮
2  for (i = 0; i < N; i++)
3  {
4      func(T[i]);
5  }
6      ⋮

```

Listing 2.3.2: Task-level parallelism

```

1      ⋮
2  independantFunc1 ();
3  independantFunc2 ();
4      ⋮

```

In Listing 2.3.2, privatization of the iteration variable `i` to multiple threads permits the allocation of *chunks* of iterations to different threads or processes based on their thread ID (TID) or process ID. Copies of the same variable `i` would exist in separate memory locations available to each thread.

For instance, a thread with TID 0 may be assigned the range from 0 to 24 of N , while the thread with TID 1 may be assigned the range from 25 to 49 of N . Executed concurrently, the 2 threads would complete 50 iterations in approximately the same time span of 25 iterations. Note the distinction from the task-level parallelism in Listing 2.3.2, where `independantFunc1()` and `independantFunc2()` may be executed simultaneously but are unique and different functions. However, unique but identical copies of `func()` are executed in Listing 2.3.2 and each copy operates on a separate chunk of the shared array T .

Parallelism in the linear acoustic wave model of Ocheltree and Frizzell [8], arises from the computation of the real and imaginary components of the complex pressure in Equation (2.2). The computation of intensity at a point P involves a summation of each pressure component, real and imaginary, over points that compose the surface area of the acoustic source. Identical computations are executed for another point P' and only the inputs are varied.

In essence, it is possible to simultaneously compute the intensity values of multiple points in the calculation volume. Or, numerous computations of the summation in Equation (2.2) may be performed concurrently, while only the coordinates of the point P , provided as input for each summation, is varied.

Because only the input data is varied and identical copies of the algorithm may be executed simultaneously, the algorithm exhibits data-level parallelism. Multiple existing parallel programming models may be adopted to exploit data-level parallelism in a given algorithm. Herein, the data-level parallelism in the algorithm is exploited with a fork-and-join model via OpenMP; with a SPMD model on a cluster and an SIMT model on a GPU.

2.4 Open Multi-Processing

OpenMP is an API developed to support parallel computing through multiple lightweight processes [19, 20]. Currently, it is deployed for a few base languages: FORTRAN, C and C++ [19] and consists of a set of compiler directives, runtime library routines and environment variables. Together, these components form a tool-set that is accessible to the programmer, provided an implementation exists for the selected base language and compiler support is available. Typically, OpenMP preprocessor directives are defined using the following syntactical convention in C [19]:

Listing 2.4.3: Syntax of OpenMP preprocessor directives

```
1 #pragma omp directive-name [clause [ [,] clause ]... ] new-line
```

Initially, an OpenMP program is enclosed completely inside an implicit parallel region. Only one thread—the *initial thread*—executes the sequential code of the program. When a parallel construct is encountered, the initial thread spawns a team of threads, each of which, will execute the code in the parallel construct, while the initial thread declares itself the *master thread*. An illustration of the OpenMP execution model is provided in Figure 2.3.

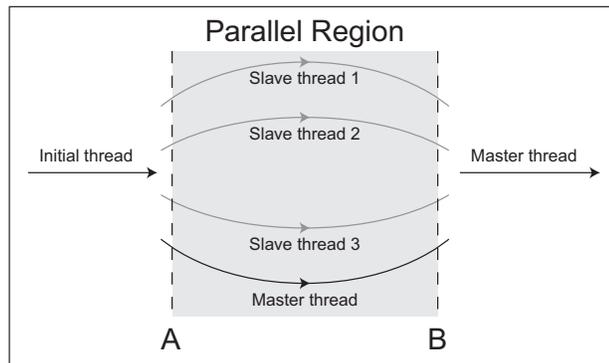


Figure 2.3: OpenMP thread creation and merging

An implicit barrier is placed at the end of a parallel construct that, when encoun-

tered, suspends parallel execution and returns control to the master thread, which continues execution from point B, which marks the end of the parallel construct.

Multithreading is achieved through utilization of the OpenMP API, which provides an implementation of threads as light-weight processes. On Linux systems, threads are implemented in OpenMP as Portable Operating System Interface for Unix (POSIX) threads [19].

By default, OpenMP creates one thread for each logical or physical CPU that it detects [19]. However, the affinity of the created threads is not guaranteed. In other words, thread migration may occur as the threads complete execution and spawn on a different physical core or if the operating system decides to balance workload by offloading a thread to another physical core.

Applications of OpenMP for performance enhancements are common in the literature and some examples are provided in Section 2.7.1.

2.5 Single Program Multiple Data Computation

First proposed by Frederica Darema, in a SPMD parallel computation model instructions execute simultaneously but asynchronously on multiple data [21].

In an early paper, Darema et al. describe the SPMD model with SPMD implementations of various numerical calculation algorithms [22]. The authors contrast the SPMD model to a fork-and-join model, where a single process begins execution and multiple processes are spawned as necessary thereafter. Execution in the SPMD model proceeds with all processes executing from the beginning, together coordinating to operate on one problem [22]. A benefit of this approach is the reduced overhead which the fork-and-join model suffers from when spawning and ending processes when encountering and leaving parallel regions [22].

The researchers implement SPMD parallel solutions for 2 problems. The first problem relates to fluid dynamics. Numerical computation is utilized to obtain solutions to three-dimensional Navier-Stokes equations with the ARC3D fluid dynamics program developed by Ames Research, NASA [22]. Specifically, the time-dependant flow at the surface of the intersection of a sphere and a cylinder is simulated for performance analysis [22]. The second problem is related to molecular dynamics. A program that simulates the physical behaviour of molecules is selected for performance enhancement for a particular problem. The problem consists of a simulation of a system of 1372 particles in an aluminum face-centered cube. After parallelization, the performance of the simulation is assessed [22].

For the simulations, Environment for Parallel EXecution (EPEX) runs in a virtual machine (VM) environment on a 2-way multi-processor (MP) IBM S/3081 machine. In terms of overhead, the authors note the problematic nature of short parallel sections in the fluid dynamics problem, which impede performance as processes are queued at synchronization points [22]. To reduce the overhead of parallelization, longer sections of parallel computations should consume a majority of the computation time. A combination of lengthy and short parallel sections may not necessarily have a large overhead, as is the case for the fluid dynamics problem studied by Darema et al., where 80% of the computation time is spent in the lengthy parallel loop section [22]. The speedup achieved by the researchers is nearly 2 with 2 processes and approximately 3.7 with 4 processes [22]. A 2-3% distance from the limit of 2 for the 2 process run is attributed to certain features of the VM operating system [22].

Though highly applicable to exploiting data-level parallelism, in a more recent piece, Darema stresses the generality of the model with the ability to execute multiple unique instruction streams [21]. Also, while SPMD is ordinarily implemented across a distributed memory system using message passing for synchronization and data-

sharing, examples of SPMD-type implementations on systems with a shared memory model also exist [23, 24, 25].

Barbara Chapman describes the SPMD approach with OpenMP as an explicit decomposition of computation and data [26, pp. 200-201]. In particular, a program which employs OpenMP to implement the SPMD model, features a large parallel region the length of the entire program. Thereafter, work-sharing constructs and synchronization clauses define boundaries within which the threads operate. Chapman notes that while the approach is more involved programmatically, it can also result in better scalability than simple for-loop parallelism [26, p. 200].

2.6 Graphics Processing Units

Significant research efforts were undertaken at the University of North Carolina, University of Utah and Xerox Palo Alto Research Institute (PARC) between the 1970s and 1980s particularly within the field of computer graphics, which fuelled the development of modern GPUs [27].

2.6.1 Computational Intensity

The demand for realistic 3D graphics manifests itself in GPUs theoretically capable of delivering greater than 1250 billion Floating point Operations per Second (FLOPS) for single precision operations in 2009 [28]. Additionally, GPUs feature high memory bandwidths exceeding 170 GB/s [28]. Comparatively, top-end CPUs are only capable of slightly greater than 125 GFLOPS for single precision operations and have memory bandwidths of approximately 35 GB/s [28].

The bandwidth of a Peripheral Component Interconnect Express (PCIe) (generation 3) slot—one type of interconnection between the CPU and GPU—is only 16

GB/s [29]. The interconnect bottleneck gives rise to a quantitative description of applications that are most suitable for GPU computations.

Buck et al. broaden the definition of arithmetic intensity defined by Dally et al. in [30] and develop the notion of computational intensity [13]. A general pattern is recognized for GPU applications, where a certain number of records are transferred to the GPU memory space, a number of computations are performed and the records are transferred back over the interconnect to the CPU memory space [13].

The authors describe computational intensity as a ratio, E/T , where E is the ratio between the execution time of an algorithm on a GPU operating on a single record and T is the transfer time associated with uploading that single record to the GPU and downloading it when computations are complete [13].

From this definition, it is observed that applications best suited for GPUs should enable a high computational intensity. Computations that perform few floating point operations but require a large number of transfers with a heavy bandwidth usage may not perform well when implemented on GPUs.

A potential bottleneck for GPU computations is the interconnect between the CPU and GPU memory spaces. Computational intensity, as defined by Buck et al. [13], emphasizes that the benefit of GPU computations should outweigh the drawback of transferring records to and from the GPU memory space over the interconnect [13].

2.6.2 Tesla Unified Computing Architecture

There are, of course, marked underlying differences between a GPU and a CPU that account for the ability of GPUs to provide greater throughput than CPUs [28]. Modern GPUs feature a hierarchical organization of processors that provide the means through which a GPU achieves high throughput with multiple threads. Lindholm et

al. provide an explanation of the Tesla unified computing architecture [31], which is utilized in the GPU experiments in the current work.

The following discussion about the Tesla architecture draws all important points from the work by Lindholm et al. in [31]. An illustration of the Tesla architecture is provided in Figure 2.4.

At the top of the hierarchy, the GPU consists of an array of independent execution units called Texture/Processor Clusters (TPCs) connected to Raster Operation Processors (ROPs), which perform memory operations on the Dynamic Random Access Memory (DRAM) chips. Separate work distribution units disseminate compute workloads to the TPCs.

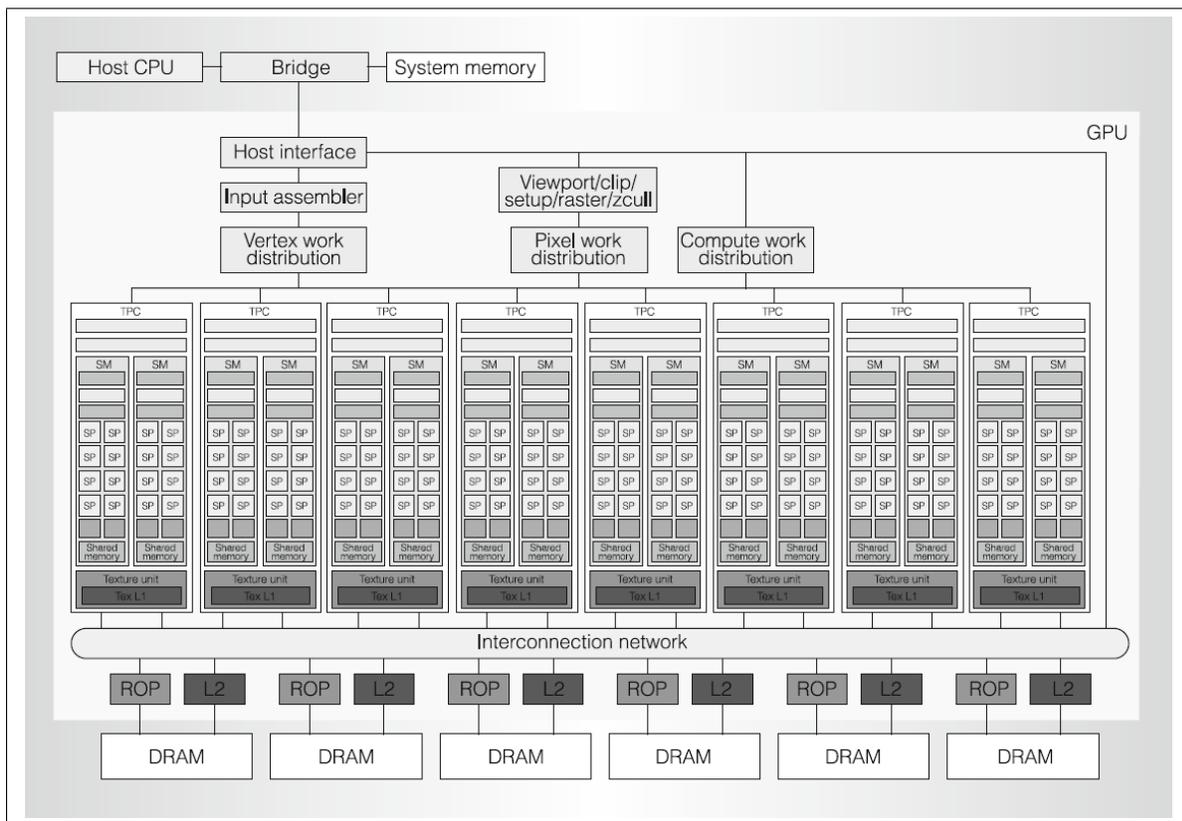


Figure 2.4: The Tesla GPU architecture with each TPCs containing two SMs. The figure is originally from [31].

The host interface unit bridges the communication gap between the GPU and the CPU. Part of its responsibilities include context switching, collecting data from system memory and also responding to commands from the CPU. Compute workloads are distributed via a round-robin scheme to the TPCs. A streaming processor controller (SMC), a texture unit and two SMs form a TPC.

Continuing down the logical hierarchy of processors, the main functional units in a TPC are the SMs. Each SM consists of an instruction cache, a multi-threaded issue unit (MT), a constant memory cache, eight Streaming Processors (SPs), two special function units (SFUs) and a 16 kB portion of shared memory that is accessible to all SPs in the SM.

Each SP contains one fused multiply-add unit (MAD) for floating-point operations such as addition, multiplication and addition-multiplication. Furthermore, each SM is also capable of performing operations with integers and also comparison and conversion operations. Eight MADs units are contained in one SM (one in each SP), which also contains two SFUs. Each SFU handles the computation of special functions such as transcendental functions. Four floating point multipliers are also contained in each SFU. The texture unit is also utilized as another execution unit, while the SMC unit and ROP unit support load and store operations from and to external memory.

2.6.3 Thread Warps

Each SM contains a SIMT instruction issue unit that performs administrative tasks on groups of 32 threads called *warps*, a term which originates from the practice of weaving [31]. A pool of 24 warps ($24 \text{ warps} \times 32 \text{ threads} = 768 \text{ threads}$) is managed by each SM.

Each thread is capable of branching and executing a separate code path. The

SIMT instruction issue unit first selects a warp from the pool that is ready to execute. Having identified such a warp, the SIMT instruction issue unit issues that instruction but only to those threads that are active within that warp. Each thread is distributed to each SP core where each thread has an instruction address and a register state.

When a warp of 32 threads follow the same code path during execution, full efficiency is realized. Divergence within a warp is handled by disabling threads that do not follow the same code path. Having completed execution of the divergent path, the execution converges by activating all threads and continuing on the same code path. Divergence of code paths only occurs between threads of the same warp. There are no inter-warp dependencies and, as such, threads of different warps execute independently.

Thread divergence and warps may be treated, for the most part, transparently by the programmers. It is not necessary to adhere to the limitation that warps of 32 threads should converge. To maximize performance, however, such considerations should be aligned with the design of GPU-based applications.

2.6.4 Parallel Computing on The Tesla Architecture

The researchers then proceed to provide a description of how parallel computing is handled in the Tesla architecture noting certain features such as synchronization, communication and cooperation that are not heavily relied upon in graphics programming models but often required in general purpose computing. Management of numerous parallel threads with cooperative capabilities led to the development of cooperative threads arrays (CTAs). In CUDA, CTAs are known as *thread blocks*.

Threads in a CTA all execute the same program concurrently but also cooperatively, synchronizing and communicating if necessary. In the Tesla architecture, each

CTA may consist of anywhere between 1 and 512 threads, each of which, has a unique TID. CTAs may also be “shaped” as desired into one-, two- or three-dimensional arrays.

The SMs in the GPU execute a maximum of eight CTA at a time. The number of CTA per SM varies and depends on the resources required by each CTA and the resources available in the SM. These resources include the number of threads, registers and the amount of shared memory required by a CTA. If the required resources are available, the SMC creates a CTA, assigning to each thread in the CTA a unique TID. Once a CTA is assigned to a SM, it executes that CTA in an SIMT fashion, processing one 32-thread warp at a time.

A collection of CTAs is organized into a grid where each CTA has a unique ID and also a grid ID. To enhance portability of executable binaries across GPUs with any number of processors, the execution of each CTA in a grid is independent of other CTAs in the same grid. Multiple grids constitute sequential steps of execution. For instance, two dependant instructions are represented as two grids composed of one or more CTAs. When each CTA independently completes processing the first grid, the second (dependant) grid is executed.

At the level of threads, the TID is utilized to selectively assign portions of the workload. Effectively, a summation of those portions is then processed by a single CTA, which is identified by the CTA’s ID. Sequential steps in the application are represented by a progression, over time, through multiple grids, which, together, are assigned the entire computational workload.

It should be noted that much of the complexity of the GPU architecture is hidden from the developer whose task is limited to creating an application for only one thread, the results from which are indexed by the TID, the CTA ID or a combination of both. The application is designed to transparently execute on an architecture with

any number of SMs and SPs. The mixture of inter-thread cooperation per CTA and independent execution of each CTA leads to a highly scalable parallel computing architecture.

2.6.5 Compute Unified Device Architecture

Nvidia introduced CUDA in November 2006 [28]. It extends the C programming language with syntactical conventions designed specifically for stream computing on GPUs. Support for Open Computing Language (OpenCL) and DirectCompute is also available for programming CUDA-enabled GPUs and a specialized implementation for Fortran is provided with CUDA Fortran. Further discussion of semantic conventions in CUDA is presented next as CUDA is utilized for the development of the GPU implementation of the linear acoustic wave model in the present work.

In general, the system with the CPU is referred to as the *host*, since the *device* containing the GPU is hosted by that system [28]. The kernel may only be called by the host and executed on the device. Furthermore, the terms *block* and CTA are inter-changeable, both referring to a collection of threads, the dimensions of which, are under developer control.

2.6.5.1 Kernels

At the center of the programming model for GPUs is the concept of kernels. The invocation of a kernel is not too different from the invocation of a function in C. The subtle differences lie in the syntactical and semantic conventions. Syntactically, a kernel *launch* involves wrapping the *execution parameters* in triple angle brackets and any remaining parameters are specified in a manner not unlike a standard C function call. Code comprising a kernel resembles that of an ordinary single-threaded

program. As an example, consider the following segment of code:

Listing 2.6.4: Kernel launch syntax

```
1 int otherParameter = 19;
2 dim3 blocks(64, 64, 1);
3 dim3 threads(32, 4, 4);
4
5 testKernel<<<blocks, threads>>>(otherParameter);
```

In Listing 2.6.4, the kernel identifier is `testKernel`. The execution parameters are specified within the triple angle brackets and establish the logical layout of threads prior to launching the kernel. `otherParameter` is a simple integer parameter passed to the kernel function `testKernel` and loaded into the global memory of the device.

2.6.5.2 Threads

Threads on a GPU are organized hierarchically with logical formations of multiple threads that aid in development but also scalability. At the bottom of the hierarchy is a single lightweight thread. A composition of multiple threads forms a logical thread block that may be either 2-dimensional or 3-dimensional. That is, a block of threads may have a certain length, a width and also a height. Illustrations of this hierarchy are provided in Figures 2.5a and 2.5b.

Special built-in variables provide access to the logical hierarchy of threads. The `threadIdx` is a device-only built-in variable and contains the TID of the current thread. For instance, in a block of dimensions 16×8 , inspecting the value of `threadIdx.x` from the first thread would yield a value of zero. Subsequent threads would yield value of 1, 2, 3 until the last thread, where the value of `threadIdx.x` would be 15. In this particular case, every block would contain threads with IDs in the X-dimension in the range of 0 and 15 inclusive. Note that `threadIdx.y` would yield values in the range of 0 and 7 inclusive such that each thread in a block may

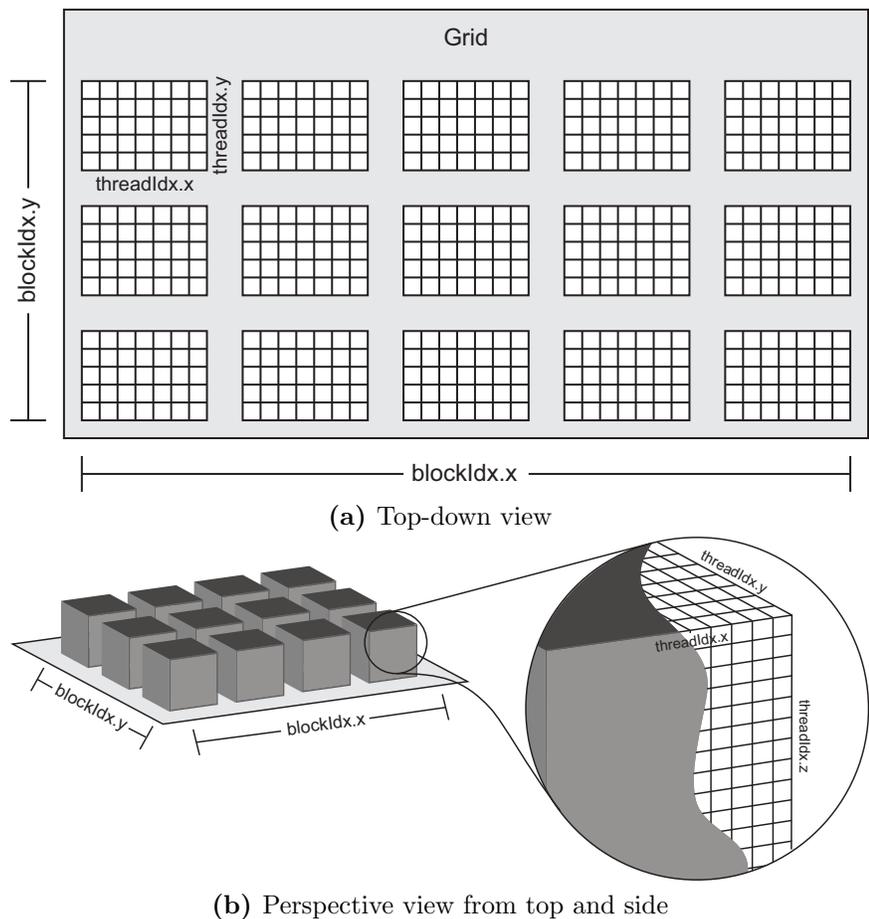


Figure 2.5: Two-dimensional and three-dimensional views of the logical organization of threads, blocks and grids in the CUDA programming model. The built-in variables `blockIdx` and `threadIdx` are provided by CUDA to address blocks and threads, respectively.

be uniquely identified with two-dimensional Cartesian coordinates (x, y) . An additional observation is that there is no Z -coordinate, since the block was declared as two-dimensional with 16×8 threads.

At the grid level, each block is identified by a unique block ID (BID), the value of which, is stored in the device-only built-in variable `blockIdx`. Suppose a grid with dimensions of 8×4 blocks. In the X-dimension, the values of `blockIdx.x` would lie in the range of 0 to 7 inclusive. For the Y-dimension, the values of `blockIdx.y` would be between 0 and 3 inclusive. A block of threads in a block with BID coordinates

(1, 1) querying the value of the `blockIdx.x` variable would each observe a value of 1.

2.7 Literature Review

The following subsections contain applications that leverage the parallel computing capabilities of multiple threads, cluster computers and GPUs similar to the work performed here. Section 2.7.1 contains works related to multithreading and cluster computing, while Section 2.7.3 contains applications that rely on GPUs.

2.7.1 Applications of Multithreading and Cluster Computing

The linear acoustic wave propagation model by Odegaard et al. is computationally intensive [32]. Therefore, Epasinghe develops a parallelized computation model in [33] that implements the same model to improve its performance.

In Epasinghe's approach, the computation domain consists of the volume of points at which intensity values are computed. The throughput of the computations is improved by decomposing the computation domain over a number of processes using the Parallel Virtual Machine (PVM) software. Behaving as middleware, PVM resides on each node and, to the application, provides a set of library routines for message passing and process spawning while transparently integrating heterogeneous architectures. Communication between nodes is supported with a message passing library of functions, which is linked to the executable at compile-time. The simulation software is also interfaced with MATLAB through a GUI to simplify usage for non-programmers.

The algorithm developed by Epasinghe is similar to an SPMD model, though technically, it is not exactly an SPMD model because not all processes are independently executed on a number of nodes. Instead, in Epasinghe's implementation, a master process begins execution and several child processes are spawned, which then handle

the workload assigned to them by the master process. In this regard, the execution model followed by Epasinghe is closer to a fork-and-join model.

For the simulations, the parallel algorithm is tested on IBM RS6000 supercomputers. Furthermore, the load distribution is assumed to be perfect with each node receiving an equal number of points to compute. If the speedup achieved when the algorithm is executed on a certain number of nodes n is defined as S_n , Epasinghe defines efficiency as S_n/n . The results from Epasinghe's study indicate near-perfect efficiency when the number of nodes is small. For instance, the efficiency achieved with 2 nodes is approximately 96%, whereas with 30 nodes it is only slightly greater than 68%. Since the efficiency of the algorithm is affected by the number of nodes and worsens when the number of nodes is increased, the algorithm does not scale well.

Near the same approximate time period and independent of the work by Odegaard et al., Jensen proposes a model that utilizes the impulse response method but which can also handle inhomogeneous media [34].

Though the spatial impulse response method is generally less computationally intensive than direct numerical solutions of the Rayleigh diffraction integral, it still benefits from cluster computing approaches to increase efficiency. Jensen presents exactly this approach in [35], though details of the computational model are vague. For the simulations, the pulse-echo response for 2000 emissions is computed and the emissions are partitioned by assigning each to a separate file [35]. The computations are performed on a 32-node Linux cluster with MATLAB 6.5 and the total time for the computations is 391 hours [35]. Jensen remarks that ordinarily such computations would complete in approximately 12512 hours, indicating a speedup of exactly 32 from the original computation model.

Aside from applications within the field of acoustics and ultrasonics, cluster com-

puting and high performance computing (HPC) are also widely applicable in other areas.

Henry, Xia and Stevens discuss the utility of HPC approaches to simulate various genome-scale metabolic models [36]. Comprehensive data pertaining to enzymatic processes are used to arrive at conclusions about whole-cell behaviour through genome-scale metabolic models [36].

Flux balance analysis (FBA) is a pivotal technique in a genome-scale metabolic model, which simulates the organism's metabolism within set boundary conditions [36]. As described by Henry, Xia and Stevens, FBA consists of 3 main components: a list of reactions; a set of gene-protein-reaction (GPR) mappings; and an objective function describing the growth behaviour of small molecules.

As an application of HPC, Henry, Xia and Stevens implement parallel gene knockout simulation algorithms on the BlueGene/P supercomputer situated at Argonne National Laboratory (ANL), which has a total of 163,840 processors [36]. Generally, gene knockout simulation involves multiple parameter sweeps on the order of 10^{10} [36]. Given the complexity of the problem space, HPC is particularly attractive in reducing the computation time to perform the parameter sweeps.

Two types of parallelism are identified resulting in 2 different algorithms. The first algorithm exploits fine-grained parallelism with high communication overhead, whereas the second algorithm applies coarse-grained parallelism with minimal or no communication overhead. The large number of CPUs in the BlueGene/P supercomputer also provides a means to test the scalability of their algorithms [36]. It is shown that the scalability of the coarse-grained algorithm is better in relation to the scalability of the fine-grained algorithm [36]. As a test of the coarse-grained approach, the authors performed a simulation on 65,536 processors, which completed in 2.7 hours simulating 18,243,776,054 quadruple knockouts [36].

In another application of HPC, Fortmeier and Bücken discuss the parallelization of an algorithm for re-initializing level-set functions [37]. Level-set functions are useful in dividing a domain into sub-domains and in the work by Fortmeier Bücken, they are applied to study the 2-phase nature of an oil drop in water in an HPC setting [37].

Instead, the authors propose discretizing the flow and the level-set function utilizing an unstructured grid composed of a set of vertices. The process is parallelized by dividing the set of vertices into multiple subsets [37]. The domain of vertices is thus decomposed into multiple sub-domains.

Division of the workload then involves submitting a subset of numerous vertices to a process [37]. Each process then leverages the multiplicity of cores by dividing that subset into further subsets to be processed by multiple threads. Three separate double-nested for-loops are parallelized in this manner. The Open MPI (OpenMPI) is utilized to disperse chunks of the iterations to processes, while the OpenMP API handles further division of those chunks to multiple threads. In this sense, a hybrid approach is followed, exploiting multiple levels of parallelism [37].

The results, as noted by Fortmeier and Bücken, highlight the importance of the interconnect speed and bandwidth for OpenMPI applications, where processors connected by an older interconnect technology produced longer runtimes than those connected with faster interconnects [37]. Furthermore, the authors note greater speedups with OpenMP enabled to decompose the previously sectioned domain. Contrary to expectations however, enabling Symmetric Multi-threading (SMT) on the processors and doubling the number of threads from 8 to 16, actually results in decreased performance [37]. A shortage of hardware execution contexts, increased contention for resources over a shared memory bus are two possible reasons that may attributed to for the decreased performance.

Reumann et al. find another application of HPC in their simulation of the electro-

mechanical behaviour of ventricles in the human heart [38]. Applying a model developed by Tusscher et al. in [39], Reumann et al. follow a domain decomposition strategy to enhance the performance of the simulations on the BlueGene/L supercomputer. The three-dimensional dataset consisting of over 32 million active ventricular elements is decomposed into binary trees [38] by utilizing an orthogonal recursive bisection (ORB) algorithm described in [40].

Sub-volumes, of number 2^n , are created and each sub-volume is mapped to a processor. Since the BlueGene/L supercomputer is structured in a binary arrangement consisting of 2^n partitions, the mapping process is simplified [38]. The number of processors ranges from 512 to 16,384. Standard non-blocking MPI functions serve the role of facilitating communication between processes [38].

A load balance metric is also described by the authors as the number of tissue elements versus the number of non-tissue elements, bearing in mind that non-tissue elements are not computed. It is this metric that supports the domain decomposition based on the computational load imposed by tissue elements, since only tissue elements require computation and non-tissue elements do not require any computation [38].

The lowest load balance tested is 1:1, where the number of tissue elements equals the number of non-tissue elements, while the highest load balance ratio tested is 1:100, which indicates that for one non-tissue element there are one hundred tissue elements which are computed [38].

The results from their study indicate near-linear speedup in all cases [38]. That is, as the number of processors is doubled, the speedup is nearly proportional to the number of processors. However, as the number of processors is increased, the load balance metric indicates a decrease in the total runtime of the algorithm but also a decrease in the load balance, where a smaller portion of the total runtime elapses

for actual computation time [38]. Furthermore, the load balance for the average communication time remains somewhat constant for all processor settings [38]. This indicates that while the speedup is nearly-linear, the computational load is not evenly spread across all processors.

2.7.2 General-Purpose Computing on Graphics Processing Units

Application of GPUs to a problem domain outside of graphics requires a mapping of that problem domain to a graphics domain. The inputs also have to be transformed into a graphics representation such as vertices or fragments. An example is the general problem of matrix multiplication that may be mapped to the graphics-related operation of a dot product between multiple vectors.

Peercy et al. present an abstraction of the OpenGL architecture as an SIMD system [41]. OpenGL is treated as a form of assembly language, on top of which a customized shading language is implemented. Though the work by the authors is far removed programmable hardware, they note the PixelFlow architecture [42] as an early example of interactive programmable shading [41].

Certain instructions in the Interactive Shading Language (ISL) are transformed to OpenGL commands that operate in an SIMD manner at the pixel level. Utilizing this approach, the researchers develop a complete RenderMan shading language implementation atop their custom ISL. Peercy et al. demonstrate the degree to which fragment and vertex shaders may be customized and the possibility of generalizing processing on GPUs.

Another example of general-purpose computing on GPUs is provided by Krüger and Westermann [43]. The authors discuss an implementation of linear algebra op-

erators on the GPU [43]. An important development in the work by Krüger and Westermann is the mapping of the problem domain of matrix operations and representations to that of graphics operations and representations. Matrices are represented by texture maps in the GPU, while shader programs are designed to handle matrix arithmetic operations.

Thompson, Hahn and Oskin also utilize GPUs for general purpose applications involving large vectors but also develop a framework to simplify the utilization of GPUs for general purpose computing [44]. For performance analysis, the authors also compute matrix multiplications and solve the 3-satisfiability problem.

Realization of these and other general applications have led to further development of languages and runtime environments to facilitate the programmability of GPUs enlarging their scope from graphics processing to stream processing. Today, several frameworks, supporting a variety of languages, exist that permit developers to access the computational facilities provided by GPUs but also other stream processing devices. Notable frameworks and APIs include BrookGPU [13], Nvidia's CUDA [28], Microsoft's DirectCompute [45] and AMD's Accelerated Parallel Processing (APP) Software Development Kit (SDK) [46].

In general, variations of the C language are implemented in each of the frameworks and APIs, with features tailored specifically for stream computing applications. CUDA C, for instance, introduces a triple-angle bracket syntax for kernel calls in C. In 2008, the OpenCL [47] was introduced as a means to standardize software that executes on architecturally varying systems composed of CPUs, GPUs and other processing units. Support for OpenCL is included in the APP SDK and CUDA, though CUDA C is utilized for the implementations in this work.

2.7.3 Applications of Graphics Processing Units

As opposed to linear wave propagation models, the authors in [48] describe the computational complexity of modelling nonlinear pressure fields citing the work in [49], which was computed on a CRAY Y-MP4D/464 system. With reference to the nonlinear numerical computation model developed by Lee in [50], the underlying reasons for the increased computation complexity may be attributed to three general factors:

(i) additional terms associated with physical phenomena must be computed. In particular, the model developed by Lee contains additional terms for nonlinearity, diffraction, and absorption; (ii) minute changes in pressure created during wave propagation cause compression and rarefaction. This results in an increase in temperature and introduces a region in which the wave propagates at a faster speed. In regions of rarefaction, the wave propagates at a slower speed. Variations in the speed of propagating waves, introduces harmonics or frequencies other than the driving frequency. A unique field must be generated for each harmonic that is considered requiring multiple computations of the entire calculation volume; and (iii) spatial complexity, which is increased when computations are performed in three-dimensional space.

The nonlinear numerical computation technique developed by Lee in [50] solves the Khokhlov-Zabolotskaya-Kuznetsov (KZK) equation, which is discussed in [51]. Zemp, Tavakkoli and Cobbold discuss another numerical computation method in [52], which relies on a second-order operator splitting technique previously developed by Tavakkoli et al. in [53].

Further discussion of nonlinear acoustics is beyond the scope of this work and is only presented here briefly to emphasize the complexity associated with modelling nonlinear wave propagation. A benefit of the nonlinear approach is that the precision of wave propagation is higher than that of linear wave propagation models [51]. An-

other characteristic of certain nonlinear models is that they also exhibit a high degree of data-level parallelism. Given the added complexity and data-level parallelism inherent in these nonlinear models, the capabilities of GPUs are particularly attractive for improving their performance.

An example of a GPU implementation of a nonlinear ultrasound model is provided by Karamalis, Wein and Navab in [54]. The algorithms developed by Karamalis, Wein and Navab synthesize B-mode ultrasound images [54].

The authors utilize finite difference numerical solutions for the Westervelt Partial Differential Equation (PDE) to model wave propagation. Included in the equation are terms to account for thermal attenuation and nonlinearity [54]. The general concept underlying the finite difference method is the evaluation of the Westervelt PDE at various sampling points in a two-dimensional grid.

The computation of the finite difference equations is performed on the GPU with code written in C++, OpenGL and OpenGL Shading Language (GLSL). The GPU utilized for the experiments is an Nvidia GeForce GTX 280 and the authors note potential performance improvements that may be possible with another implementation written in CUDA C or OpenCL.

The remaining steps to process the image are performed on the CPU as they are less computationally intensive [54]. Two image datasets are simulated. The fetus dataset and the phantom dataset required 55 minutes and 78 minutes, respectively, to complete generation of radio frequency data on the GPU with 2048^2 points and 192 scan lines [54]. Comparatively, image formation on CPU only required 19 seconds for the fetus dataset and 24 seconds for the phantom dataset [54]. The authors note that similar computations required 32 hours to complete utilizing the simulation framework developed by Pinton et al. in [55].

Another example is provided by Michéa and Komatitsch in the evaluation of the

performance of solving the seismic wave equation using finite difference methods in the time domain on GPUs [56]. Single-GPU testing is performed on an Nvidia GeForce 8800 GTX GPU, while tests on multiple GPUs are performed on a GPU cluster consisting of multiple Tesla S1070 servers.

The researchers perform two experiments to test the scalability of their approach on multiple GPUs. In the first experiment, the total workload across all GPU is increased but the workload per GPU remains equal. In the second experiment, the total workload remains constant and as the number of GPUs is increased, the workload handled by each GPU decreases. The first experiment is a test of “weak scaling” and the second experiment is a test of “strong scaling” as defined by the authors [56].

Results from the study by Michéa and Komatitsch indicate that speedup remains constant in the case of weak scaling since the workload per GPU is constant and high enough to hide the effects of high latency operations on the GPU [56]. Conversely, the effect of strong scaling is that speedup no longer remains constant as the number of GPUs is increased [56]. As the number of GPUs is increased, the workload per GPU is decreased and the effects of high latency operations and parallelization overhead become more apparent. The overall speedup from their work ranges between 20 and 60 when utilizing either a single GPU or multiple GPUs as opposed to a CPU.

Chapter 3

Parallel Computing Models, Algorithms and a Multi-layer Linear Acoustic Model

As part of the contribution of this work, the focus now shifts to enhancing the performance of the algorithm developed by Ocheltree and Frizzell [8] with particular emphasis on reducing the total execution time of linear acoustic wave simulation.

Where a trade-off between memory and execution speed is required, execution speed is favoured. Exchanging higher spatial complexity for a reduction in temporal complexity is considered viable given that the spatial requirements are low and costs of both volatile and non-volatile storage are not expensive. Usage of faster but more limited volatile memory (for instance, DRAM) is also favoured as memory chips of these type continue to increase in read and write speeds and transfer bandwidth.

This chapter first describes the sequential method of computation in Section 3.1. Section 3.2 presents a method for reducing redundant computations for axisymmetric acoustic fields. The workload that is subjected to parallel computation is defined in Section 3.3. Subsequent sections present details pertaining to three parallel computing models that are developed in this work to enhance the performance of linear acoustic wave simulation.

The Fork-and-join (FJ) model for parallel computation of the workload is described in Section 3.4, while the extension of that same model manifests itself in the SPMD model, which is described in Section 3.5.

Section 3.6 contains a description of the last parallel computation model, which is designed for GPUs. Subsections 3.6.1-3.7 describe the separation of device and host code and the method by which CUDA entities are mapped to the acoustic field in the numerical model.

Additionally, in this work, the linear acoustic wave propagation model by Ocheltree and Frizzell [8] is augmented to model wave propagation across multiple media or multiple layers. The multi-layer model is described in Section 3.7 and a summary of the chapter is presented in Section 3.8.

3.1 Sequential Computation

The general form of the sequential algorithm for field computation is presented in Algorithm 1. Algorithm 1 is a means to compute the intensity values at multiple points in the acoustic field utilizing Equation (2.2) in Section 2.1.2.

Computation of the acoustic field proceeds along the X-axis of the calculation volume, followed by the Y-axis before proceeding to the next X-Y plane in the Z-axis. The computation sequence is summarized in Algorithm 1 with 3 major for-loops that process data for each point in the acoustic field.

Three factors may be identified as having a direct effect on the temporal complexity of Algorithm 1: the number of points in the calculation volume, the number of elements in the composition of the acoustic source and the number of sub-elements in each element. Each of these factors is directly mapped to one of the 5 for-loops in Algorithm 1, where an increase or decrease in execution speed is related to the

Algorithm 1: Sequential field computation

```
1 for  $k \leftarrow start_z$  to  $end_z$  do
2   for  $l \leftarrow start_y$  to  $end_y$  do
3     for  $i \leftarrow start_x$  to  $end_x$  do
4       for  $element \leftarrow 1$  to  $N_e$  do
5         collect element parameters;
6         calculate number of sub-elements;
7         for  $subelement \leftarrow 1$  to  $N_{subelements}$  do
8           calculate real field contribution  $F_r$ ;
9           calculate imaginary field contribution  $F_i$ ;
10           $C_r \leftarrow C_r + F_r$ ;
11           $C_i \leftarrow C_i + F_i$ ;
12        end
13      end
14       $field[i,j,k] \leftarrow C_r^2 + C_i^2$ ;
15    end
16  end
17 end
```

number of iterations of each for-loop. N_x , N_y and N_z increase as the dimensions of the calculation volume are increased or as the step sizes S_n in the X, Y or Z directions are decreased.

The number of elements, N_e , increases as the element size is decreased or, as the size of the acoustic source is increased and N_e remains constant. Note that while N_x , N_y and N_z are static parameters along with the number of elements (N_e), the number of sub-elements ($N_{subelements}$) is dynamic and changes as the angle and distance from the observation point varies. This effect manifests itself in Equation (2.6) where the width of a single sub-element is affected by the the Z-distance indicated by the variable z .

Each parallel computing model described in this chapter essentially “peels” the three for-loops in Algorithm 1 that start between lines 1 and 3. The core of Algorithm 1 is between lines 4 and 14. In the SIMT parallel computing model, the core of

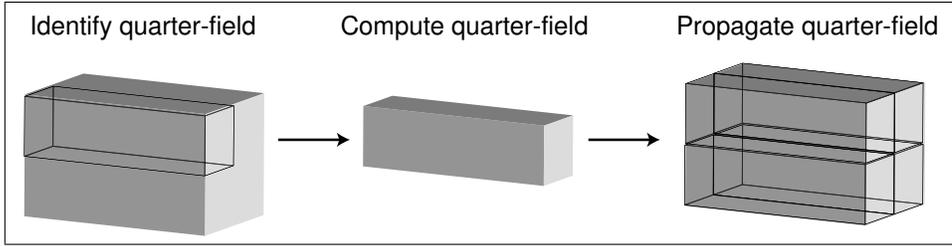


Figure 3.1: Axial-symmetric workload division

Algorithm 1 is exposed and computed directly by each unit of execution.

3.2 Reducing Redundancies

Further speedup in computation may be realized by reducing the number of redundant calculations in the acoustic field, which is axisymmetric around the Z-axis. That is, the intensity value at (x, y, z) is equal to the intensity values at $(-x, -y, z)$, $(-x, y, z)$ and $(x, -y, z)$. The equality of these values asserts that computation of only one of these values is sufficient to obtain the other three.

Consequently, only one quarter of the calculation volume requires computation and is aptly termed *quarter-field computation*. The process is illustrated in Figure 3.1. Both circular and rectangular acoustic sources may benefit from this optimization, as both these source shapes produce acoustic fields with reflective symmetry along the X and Y axes. However, only acoustic fields emanating from circular sources exhibit rotational symmetry about the Z axis. That is, at a certain radial distance r from the origin, all intensity values are equal in an acoustic field generated by a circular source.

A restriction with quarter-field computation is that the calculation volume lie directly in front of the acoustic source, where $start_n$ and end_n values for X and Y axes cover an equal number of points around the origin. Or, $start_n = -(end_n)$ for

the X-axis and also the Y-axis, though the ratio N_x/N_y need not be 1. That is, the volume may be a rectangular prism situated directly in front of the acoustic source.

Redundancies can still be reduced with off-axis calculations by sectioning that part of the calculation volume that is directly in front of the transducer. Quarterfield-computation is then performed on only that portion and the leftover portion is computed normally to complete computation of the calculation volume. The results provided here however, impose the former restrictions and quarter-field computation is only performed on an on-axis calculation volume.

3.3 Workload Definition and Decomposition

To distribute the workload, the workload decomposition technique applied in the present work is similar to the technique followed by Reumann et al. [38] to the extent that the three-dimensional workload is mapped to multiple processors.

Contrary to the ORB algorithm in the work by Reumann et al., no binary trees are formed in the workload decomposition process in the current work and no IPC is required to compute points within the same neighbourhood. Rather, the workload decomposition herein is a one-step process for the multithreaded implementation and a two-step process in the cluster implementation owing to the hybrid nature of that approach. In the multithreaded approach, the workload is simply divided over the number of threads, while in the cluster approach the workload is first divided over the number of processors and, in the second step, segmented further over the number of threads per processor.

The model detailed by Ocheltree and Frizzell [8] involves point-by-point computation of a certain calculation volume, which consists of varying intensity values and forms the acoustic field. The acoustic source is composed of several rectangular ele-

ments, which are, themselves, composed of smaller sub-elements. Each point in the finite calculation volume involves integration over the surface of an arbitrarily-shaped source. Thus, the computation of the entire calculation volume or acoustic field is considered the workload and the volume of the workload is described by the number of points in the calculation volume. The number of points over a certain interval is defined as:

$$N_n = \left\lfloor \frac{(start_n - end_n)}{S_n} + 1 \right\rfloor, \quad (3.1)$$

where n specifies the direction (X, Y or Z), $start_n$ is the starting point of the interval, end_n is the end point of the interval and S_n is the step size in the direction of X, Y or Z. The size of the workload for one plane parallel to the X-Y axes is then $W_{xy} = N_x \cdot N_y$. Similar calculations may be performed for the X-Z and Y-Z axes, while the total size of the workload is defined by $T = N_x \cdot N_y \cdot N_z$.

Given the workload characterization of the linear acoustic simulation model, the data-level parallelism inherent in the computations is exploited via a workload decomposition strategy.

In the descriptions that follow, three models for parallel computing are outlined. The first model utilizes multiple threads to efficiently harness CPU resources. The utility of this approach is particularly apparent when the physical CPU consists of multiple logical or physical cores in an SMP arrangement.

An extension of the FJ model is realized in the SPMD model where multiple CPUs in a distributed memory architecture may be utilized with multiple processes that each execute multiple threads to further divide the workload across multiple cores in a shared memory architecture.

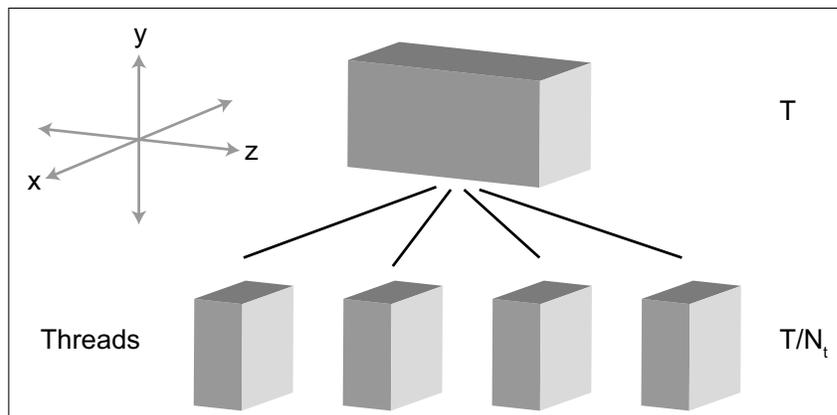


Figure 3.2: Distribution of workload in the FJ model.

3.4 Fork-and-join Model and Algorithm

The coarse-grained approach by Henry, Xia and Stevens [36] is adapted in the present work to develop the FJ and SPMD models. Their fine-grained algorithm follows a master-slave model and requires communication between processes, while in their coarse-grained algorithm, each slave determines its own workload and no IPC is required. The coarse-grained algorithm is found to be highly scalable up to 65,536 processors and a similar approach is applied in the current FJ and SPMD models, which do not require any IPC and so, have no communication or synchronization overhead.

The FJ model utilizes as threads as the units of execution. The total workload T is divided by the number of available threads at different points on the Z -axis. Selection of the Z -axis for workload distribution is to elicit the largest possible portion of the workload and assign it to a thread. An example of the workload decomposition in the FJ model is provided in Figure 3.2, where four threads are assigned to multiple contiguous X - Y planes or a *chunk* of the total workload T .

The left side of Figure 3.2 indicates the units of execution, which in this case

includes only threads. The right side of Figure 3.2 displays the workload per unit of execution for that level of workload decomposition. In this case, the total workload T is divided over the number of threads N_t .

In Algorithm 1, for-loop parallelism is exploited utilizing OpenMP by assigning to each available thread a portion of the number of points along the Z-axis. Hence, the range of the iterative variables $start_z$ and end_z is shortened to N_z/N_t , where N_t is the number of threads. It is worth noting that efficient utilization of all logical or physical CPU cores should ensure that $N_t \mid N_z$, which means that N_t divides N_z or $N_t \cot k = N_z$ for some integer k where the symbol \cdot always denotes multiplication.

Naturally, as threads share the same address space, the iterative variable is private to each thread. Algorithm 2 describes the parallel computation in the FJ model. Note that the for-loop on line 1 is parallelized.

Algorithm 2: Field computation in the FJ model

```

1 for  $k \leftarrow start_z$  to  $end_z$  do in parallel
2   for  $l \leftarrow start_y$  to  $end_y$  do
3     for  $i \leftarrow start_x$  to  $end_x$  do
4       for  $element \leftarrow 1$  to  $N_e$  do
5         collect element parameters;
6         calculate number of sub-elements;
7         for  $subelement \leftarrow 1$  to  $N_{subelements}$  do
8           calculate real field contribution  $F_r$ ;
9           calculate imaginary field contribution  $F_i$ ;
10           $C_r \leftarrow C_r + F_r$ ;
11           $C_i \leftarrow C_i + F_i$ ;
12        end
13      end
14       $field[i,j,k] \leftarrow C_r^2 + C_i^2$ ;
15    end
16  end
17 end

```

The FJ model is simple and with multi-core SMP provides a viable technique to

achieve improved performance by exploiting data-level parallelism. However, since SMP systems share a common memory bus in a shared memory architecture, the scalability of such systems enforces a limit to the maximum performance that can be gained on such systems.

To overcome this performance barrier the FJ model is augmented with a multi-process approach in another parallel computing model designed for better performance on clusters.

3.5 SPMD Model and Algorithm

Fortmeier and Bücken apply a second level of parallelism providing an extension to the coarse-grained parallelism with multiple OpenMP threads through OpenMPI [37]. A similar hybrid approach is followed in the present work with the exception of utilizing OpenMPI for IPC. Instead, each process is assigned a fixed portion of the total workload, which is segmented and dispersed to multiple threads through OpenMP and requires no IPC.

The lack of any dependencies between the calculation of any two points in the calculation volume permits the computation of the acoustic field to occur over separate processes and memory spaces without any IPC. Without any IPC, the scheduling of each process is simplified and restrictions in terms of scalability are limited to the space in which each process executes.

The units of execution in the SPMD model are threads and processes. Multiple levels of parallelism are applied by first dividing the total workload along the Z-axis as in the FJ model. In the case of the SPMD model however, each sectioned workload is assigned to a process instead of a thread. An illustration of the distribution of the workload is provided in Figure 3.3, where the number of processors and threads is

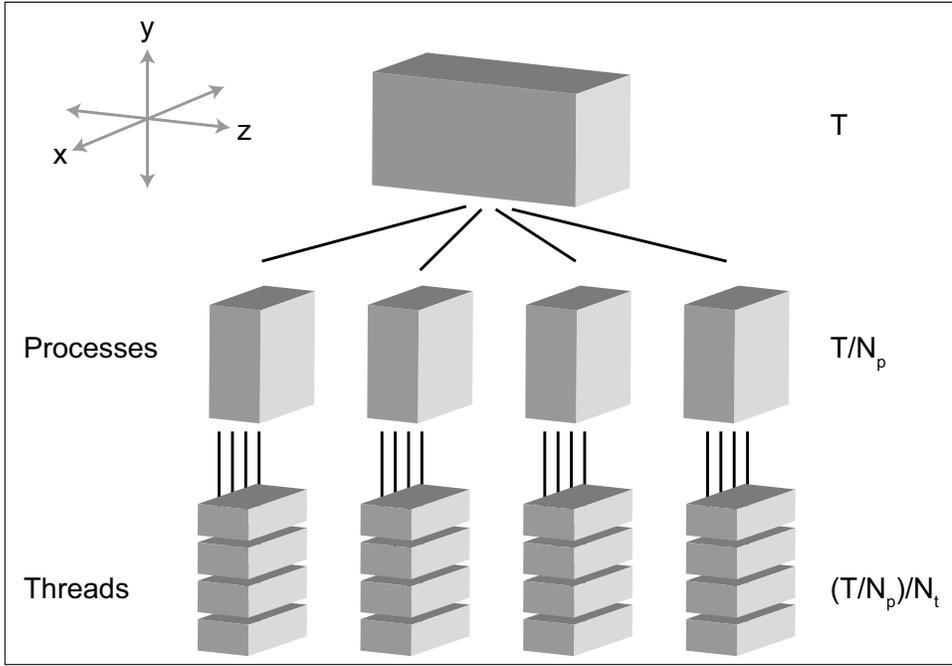


Figure 3.3: Distribution of workload through multi-processing and multithreading in the SPMD model.

four.

The left side of Figure 3.3 indicates the units of execution, which in this case includes threads and processes. The right side of Figure 3.3 displays the workload per unit of execution for that level of workload decomposition. In this case, the total workload T is divided over the number of processes N_p , which is then further divided by the number of threads N_t .

Thus, the assigned workload for a process is $W_p = T/N_p$, where the N_p is the number of processors. The utility of each CPU is enhanced by further dividing the assigned workload at different points along the Y-axis instead of the Z-axis. Each thread is then assigned a smaller workload, the size of which, is calculated as below:

$$W_t = \frac{T}{N_p \cdot N_t}. \quad (3.2)$$

Note, once again, that while any total number of points T would compute with this approach, an even distribution of the workload requires that $N_p \mid N_z$ and $N_t \mid N_y$.

The parallel computation in the SPMD model is exemplified in Algorithm 3. The for-loops on lines 1 and 2 are computed in parallel by first assigning a range of iterations of the for-loop on line 1 to a process. Secondly, the process spawns multiple threads and assigns each thread a range of iterations of the for-loop on line 2.

Algorithm 3: Field computation in the SPMD model

```

1 for  $k \leftarrow start_z$  to  $end_z$  do in parallel
2   for  $j \leftarrow start_y$  to  $end_y$  do in parallel
3     for  $i \leftarrow start_x$  to  $end_x$  do
4       for  $element \leftarrow 1$  to  $N_e$  do
5         collect element parameters;
6         calculate number of sub-elements;
7         for  $subelement \leftarrow 1$  to  $N_{subelements}$  do
8           calculate real field contribution  $F_r$ ;
9           calculate imaginary field contribution  $F_i$ ;
10           $C_r \leftarrow C_r + F_r$ ;
11           $C_i \leftarrow C_i + F_i$ ;
12        end
13      end
14       $field[i,j,k] \leftarrow C_r^2 + C_i^2$ ;
15    end
16  end
17 end

```

In a cluster computer, allocation for jobs is usually performed based on the number of nodes that are required for a particular job. A node consists of a number of processor cores, a certain amount of Random Access Memory (RAM) and some hard disk drive (HDD) space. In a distributed memory architecture, the memory space of each node is independent of the memory space of any other node.

Since the type of parallelism in the cluster computing model is coarse-grained—that is, without any IPC—an immediate allocation of a large number of resources is

not required.

Consider a case where N_p is set to 8. That is, the number of processors or cores is set to 8. In a system with 2 processors or cores per node, this requires an allocation of 4 nodes. Each process is submitted as a separate job and computation proceeds asynchronously and independent of other processes on separate nodes. The results are then collected after each job is completed.

3.6 SIMT Model and Algorithm

The GPU-based SIMT model for parallelized simulation of linear acoustic wave propagation is a culmination of the previous multithreaded and cluster models for exploiting parallelism in the sequential algorithm. The SIMT model also extends the work presented by myself and others in [57]. As mentioned at the start of the chapter, various factors affect the temporal complexity of the sequential algorithm. Of these factors, the for-loops that iterate over the X, Y and Z axes are major contributors to the computation workload.

A realization that is not difficult to arrive at is that, with enough resources, the computation workload may be continuously divided and assigned to multiple execution units until the computation of each point in the calculation volume occurs simultaneously. With that level of parallelization, the only limiting factor is the time required for sequential processing of a single point. It is precisely this concept that underlies the SIMT programming model.

In the SIMT model, extraneous for-loops for each of the X, Y and Z dimensions are removed and only the code pertaining to the computation of a single point is exposed. The core code is all that is necessary to compute the entire acoustic field. Due to the design of GPUs and CUDA, modifications to the sequential algorithm

code are required as part of the modification of the sequential algorithm to an SIMT parallel computing model.

3.6.1 Device and Host Code Separation

The SIMT model of the sequential algorithm divides the sequential algorithm code into host and device components. Code designated for the host executes on the CPU and occupies the main memory space accessible to the CPU. Device code includes all code that executes on the GPU within the confines of GPU memory. Though such a division of code exists, there exists only one executable, which begins execution on the host and prepares the GPU prior to device code execution on the GPU.

Initially, the sequential algorithm code executes completely on a host system. As part of the SIMT model, all variables required exclusively for the computation of a single point in the calculation volume are identified. These variables are removed from the host code and declared in device code since they are required for acoustic field computations. The separation between device and host code is presented in Figure 3.4

With variable storage implemented in the device code, the core execution code of the sequential algorithm is identified. The core execution code is only that code that is necessary to obtain a single intensity value given certain X, Y and Z coordinates of one point. In the sequential algorithm, the three for-loops that iterate over coordinates in the X, Y and Z axes are not necessary in the SIMT model because the core execution code outputs only a single intensity value at one point in the acoustic field.

A single kernel consists of a single stream of instructions and with multiple copies of that kernel executing simultaneously, multiple instruction streams exist. In this sense, the SIMT model benefits from the paradigm of stream computing exploiting

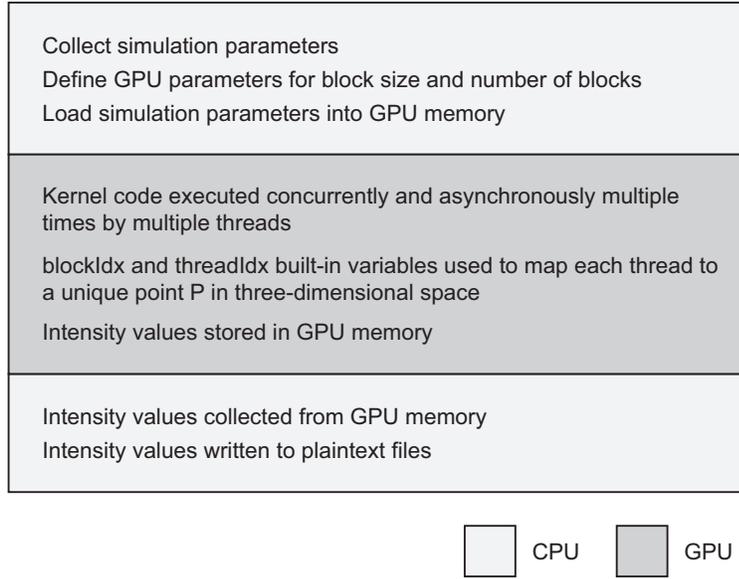


Figure 3.4: Device-host or GPU-CPU code separation in the SIMT model.

Algorithm 4: Field computation kernel in the SIMT model

```

1  xBlocksPerPlane  $\leftarrow N_x / \text{blockDim.x}$ ;
2   $i \leftarrow \text{blockDim.x} * (\text{blockIdx.x} \% \text{xBlocksPerPlane}) + \text{threadIdx.x}$ ;
3   $j \leftarrow \text{blockDim.x} * \text{blockIdx.y} + \text{threadIdx.y}$ ;
4   $k \leftarrow (\text{int})(\text{blockIdx.x} / \text{xBlocksPerPlane})$ ;
5  for element  $\leftarrow 1$  to  $N_e$  do
6    collect element parameters;
7    calculate number of sub-elements;
8    for subelement  $\leftarrow 1$  to  $N_{\text{subelements}}$  do
9      calculate real field contribution  $F_r$ ;
10     calculate imaginary field contribution  $F_i$ ;
11      $C_r \leftarrow C_r + F_r$ ;
12      $C_i \leftarrow C_i + F_i$ ;
13   end
14 end
15  $\text{field}[i,j,k] \leftarrow C_r^2 + C_i^2$ ;

```

data-level parallelism inherent in the original sequential algorithm.

When the core execution code is provided a different set of X, Y and Z coordinates, it outputs the intensity value at a point with exactly those coordinates. In this manner, when multiple copies of identical core execution code or multiple copies of

the kernel are executed simultaneously and each copy operates on a different point, intensity values composing the entire acoustic field may be computed. The kernel algorithm for the implementation of the sequential algorithm on the GPU is presented in Algorithm 4.

The variables `i`, `j` and `k` are dependant on the built-in variables `blockIdx` and `threadIdx`. Due to this dependency, the combination of the variables results in a unique three-dimensional index in the `field` array on line 15.

3.6.2 Calculation Volume Mapping

Another modification involves the transformation of the calculation volume in a manner such that points in the calculation volume become functions of the built-in variables in CUDA. Because the sequential algorithm is initially implemented such that computations are performed for a three-dimensional acoustic field and stored in a one-dimensional array, a similar strategy is applied in the implementation of the sequential algorithm on the GPU.

In host code, memory for the acoustic field is first allocated as a one-dimensional array in DRAM on the GPU, the scope of which, is global. For an acoustic field with 64 points in X, Y and Z dimensions, this requires the dynamic allocation of 262144 floating point values in GPU memory. Pivotal to the SIMT model is then mapping the input values and output locations to TIDs and BIDs such that each thread is assigned a unique input value and output location. This mapping is explained in multiple steps.

Because CUDA CTAs or simply, blocks, are either two-dimensional planes or three-dimensional cuboids, a natural mapping to the input points in the calculation volume is formed. A geometrical division of the workload is developed to establish a

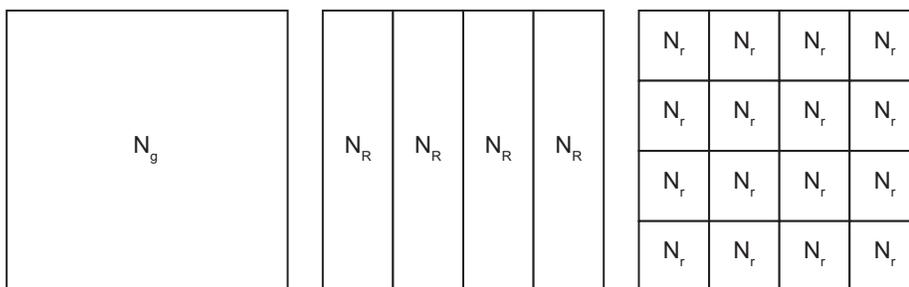


Figure 3.5: Division of points in a single plane of the calculation volume

relationship between input points in the calculation volume and CUDA blocks and threads.

There are, of course, several methods of dividing points in the calculation volume. Herein, a single plane of the calculation is first considered. A single plane is composed of a certain number of points, defined by N_g , that are arranged in a large rectangular grid with equal spacing between all points. Suppose the large grid of points is then divided with horizontal lines with equal spacing between each line.

A hierarchy is then formed with the large grid composed of a number of smaller rectangles or rectangular strips, which are themselves composed of a certain number of points N_R . The large grid is then further divided with vertical lines and equal horizontal spacing between each line. Instead of rectangles, each with a certain number of points N_R , smaller rectangles now compose the grid. Each of these smaller rectangles, or tiles, which are not necessarily squares, contains a certain number of points N_r . Also, note that $N_r < N_R < N_g$. Figure 3.5 illustrates the division of a single plane with the described method.

With the representation of a single plane of points depicted in Figure 3.5, the logical formation of blocks of threads is simplified when a single tile is considered equivalent to a single block in CUDA. Each block has a unique ID accessed with the `blockIdx.x` and `blockIdx.y` variables. Note that the block in the upper-left has a

`blockIdx.x` value of 0 and a `blockIdx.y` value of 0. Index values increase to the right and to the bottom for the `blockIdx.x` and `blockIdx.y` variables, respectively. Index values are also always positive and never negative but may be zero.

Contained in each block is a set of points, and, similar to the equivalence between tiles and blocks, a relationship is formed between points and threads. Each thread is addressed with a unique combination of the `threadIdx.x` and `threadIdx.y` variables. The combination of both variables describes a point in a coordinate system where numbering begins at $(0,0)$ in the upper-left corner of a block and increases to the right and to the bottom. Like block index values, thread index values may be zero and are always positive.

3.7 Multi-layer Linear Acoustic Model

The model described by Ocheltree and Frizzell simulates wave propagation in a single homogeneous medium. A final development in the current work includes an extension of the linear acoustic model to multiple layers, a technique that becomes viable following the development of the previous HPC models.

Two different homogeneous media with varying medium properties are separated by an interface. Consider, for instance, a tissue sample submerged in water. As acoustic waves propagate through the water medium, they encounter an interface, a plane at which the water and tissue media are separated. At this point the waves exhibit phenomena that include diffraction, refraction and reflection. The pattern of wave propagation in the second tissue medium is changed after propagating past the interface between two different media due to variations in the physical properties of the medium through which the wave is propagating. That is, differences in the speed of sound in the medium, the density of the medium and attenuation in the medium

affect the pattern of wave propagation that is observed as the wave travels beyond the interface.

Imaginary and real components of sound pressure, p_0 , are computed prior to computation of the intensity values in the acoustic field. Subsequently, the imaginary and real sound pressure components of pressure are converted to time-averaged intensity at a point P as described in Section 2.1.2.

The extension of the linear acoustic model to multiple layers relies on collecting the particle velocity phase values and particle velocity amplitude values at each point in a single X-Y plane located at the interface between two media. Instead of calculating time-averaged intensity at each point in the interface plane, the real and imaginary components are utilized to compute the particle velocity phase and particle velocity amplitude of the waves at those points. Particle velocity phase, in radians, is calculated with the following equation:

$$p_v = \arctan(i/R) \quad (3.3)$$

and particle velocity amplitude, in Pascals, is obtained with the following equation:

$$v_a = \sqrt{R^2 + i^2}. \quad (3.4)$$

In both equations, i is the imaginary and R is the real component of the complex pressure quantity p_0 defined by Equation (2.2) in Section 2.1.2.

With reference to the elements that compose an acoustic source described in Section 2.1.2, the output of each element is a wave with a certain particle velocity phase and a certain particle velocity amplitude.

Initially, the particle velocity phase of all elements in the acoustic source is zero radians, an indication that, given a completely flat acoustic source, waves produced

by each element are in-phase. A curved acoustic source naturally produces waves that are not necessarily in-phase despite zero values for particle velocity phase for each element in the acoustic source. The phase alignment is affected by the curvature of the acoustic source. Particle velocity amplitudes for all elements are equal reflecting a uniform velocity distribution over the surface of the acoustic source. The output power of the acoustic source is also related to the particle velocity amplitude of each element. Higher particle velocity amplitudes result in greater intensity values in the acoustic field.

A virtual acoustic source is introduced at exactly that location where the interface between two media exists. It is important to distinguish between the physical and virtual acoustic sources. The physical acoustic source is an actual acoustic source in that it is equivalent to a physical transducer with certain physical properties and marks the actual source for the propagation of acoustic waves. In contrast, a virtual acoustic source does not physically exist. The notion of a virtual source is introduced to account for certain effects that result due to a transition between two media. The effects of acoustic waves propagating through another medium are accounted for beginning at the location of the interface between the two media.

The geometric setup for multi-layer linear acoustic simulation is illustrated in Figure 3.6. The development of the virtual source begins with the identification of the last X-Y plane in the first medium. The last X-Y plane is the X-Y plane with the highest Z-coordinates that are still within the confine of the first medium. Or, the last X-Y plane is furthest from the physical acoustic source but belongs within the first medium. Particle velocity phases and particle velocity amplitudes are collected for each point in the last X-Y plane.

Depending on the difference between acoustic impedances of two media at the

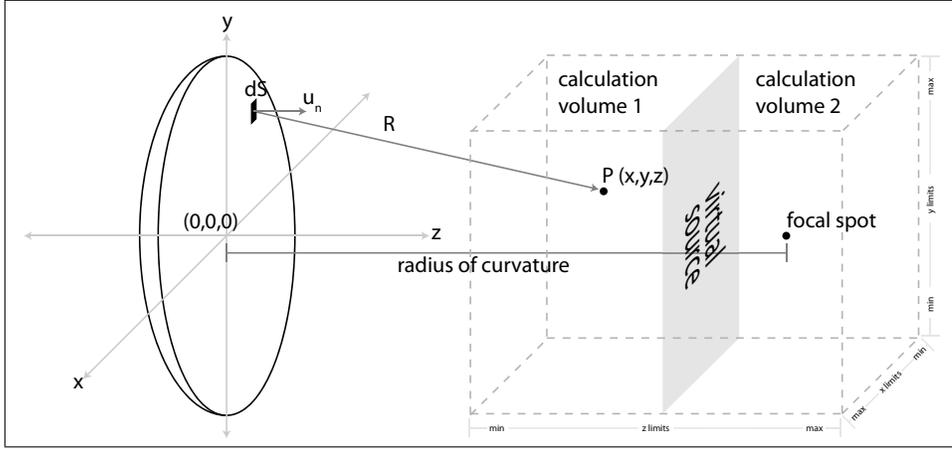


Figure 3.6: Geometric depiction of simulation with two adjacent media

interface, a certain portion of the wave is reflected back into the originating medium while the remaining portion is transmitted into the next medium. To account for the portion of the wave that is transmitted, the following correction factor is applied to all particle velocity amplitude values obtained from the last X-Y plane [9, p. 54]:

$$\text{Corrected } v_a = v_a \cdot \frac{2Z_{02} \cos \theta_i}{Z_{02} \cos \theta_i + Z_{01} \cos \theta_t} \quad (3.5)$$

where Z_{01} and Z_{02} are the acoustic impedances of the first and second media, respectively, θ_i is the angle of the incident wave and θ_t is the angle of the transmitted wave. v_a refers to the particle velocity amplitude at a certain point in the last X-Y plane. In the multi-layer experiments, the angle of the incident and transmitted waves are assumed to be 0 radians, which removes any effects of the angle of incidence and transmission at the cost of simplifying the simulation process. It is clear that only a difference between the acoustic impedances of two different media will produce a correction factor with a value other than 1. When multi-layer simulation is performed for two identical media, the correction factor is always 1.

A flat rectangular virtual source geometry is then generated such that the number

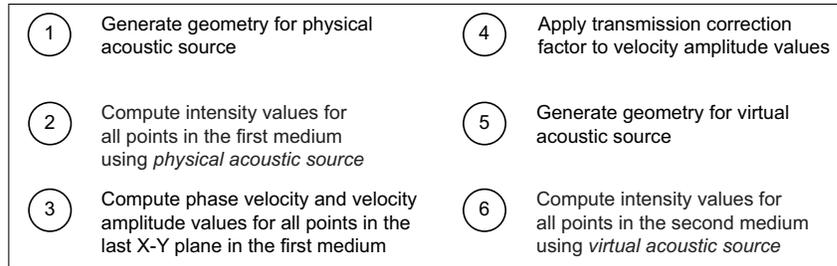


Figure 3.7: Steps involved in multi-layer computations with the considered linear acoustic model

of elements in the virtual source equals the number of points in the last X-Y plane. Each point in the last X-Y plane is associated with an element in the virtual source. For an element associated with a certain point in the last X-Y plane, the particle velocity phase and corrected particle velocity amplitude at that point are stored as part of the 16 properties that describe that element.

The general steps involved in the computation of wave propagation across multiple layers are summarized in Figure 3.7. The virtual source, the last X-Y plane and the interface are all situated at identical locations. The simulation is then repeated with waves propagating from the virtual source into another medium or a secondary layer with attributes that are potentially different than those of the first layer.

3.8 Summary

An important aspect of developing any parallel computing model is the definition and division of the workload over a number of execution units. Thus, it is presented first in this chapter for each parallel programming model.

The workload is defined as the number of points in the acoustic field that must be computed. In the multithreading model, the total number of points is divided over a number of threads. The cluster computing model extends this workload division

by utilizing multiple processes with multiple threads. The total number of points is divided over a number of processes and each process divides the workload further over a number of threads to maximize the utility of each multi-core CPU. In this sense, the workload division in the cluster computing model is hierarchical.

Workload division in the SIMT model is performed by assigning a single block of threads to a section of points in a single plane in the calculation volume. Each thread handles the computation of a single point. Mapping between threads and points is achieved by utilizing the `blockIdx` and `threadIdx` built-in variables in CUDA. Multiple blocks combined are assigned all the points in the calculation volume. This type of mapping provides the flexibility of mapping calculation volumes of different sizes to threads on the GPU.

Finally, a model for multi-layer linear acoustic wave propagation is presented. Multi-layer wave propagation involves the collection of information related to the state of waves at multiple points in a single X-Y plane that divides two homogeneous media. The state of the waves at this point is utilized to create a virtual acoustic source, which essentially “continues” the pattern of wave propagation into a second layer by repeating the simulation described by Algorithm 1.

Chapter 4

Parallelized Implementations, Multi-layer Simulation and The LATS Software Package

Following an explanation of the metrics utilized to assess performance in Section 4.1, the results achieved with the HPC implementations are presented in this chapter.

The first multithreaded implementation is based on the fork-and-join model presented in Section 3.4 and is targeted towards a single multithreaded SMP processor and operates in a shared memory environment. The second implementation leverages the capabilities of clusters as a HPC approach utilizing multiple threads and processes. The SPMD model described in Section 3.5 is implemented on cluster computers by extending the first multithreaded implementation with a cluster of multiple CPUs in a distributed memory environment. Results from the multithreaded and cluster computing implementation are presented and discussed in Section 4.2. The implementations and results are also given in [57].

The third and last GPU implementation follows the SIMT parallel programming model discussed in Section 3.6 to exploit the massively parallel design of GPUs. Results of the GPU-based implementation are presented in Section 4.3.

Furthermore, an extension from a single medium to multiple media of the linear

Table 4.1: Relevant simulation parameters and their respective values* for each of the three experiments.

Parameter	Experiment number		
	One	Two	Three
$start_x$	-17.5	-12.7	-12.75
end_x	17.5	12.7	12.75
$start_y$	-15.75	-12.7	-12.75
end_y	15.75	12.7	12.75
$start_z$	4.5	5.0	5.0
end_z	36	30.4	30.5
$S_{x,y,z}$	0.5	0.2	0.1
$esize$	0.5	0.2	0.1
$espace$	0.5	0.2	0.1
$tsize$	10	10	10
F	20	20	20

* Aside from the parameter F , which is unitless, all parameter values are in millimeters.

acoustic wave propagation model by Ocheltree and Frizzell [8] is described in Section 4.4. Preliminary results from the multi-layer linear acoustic wave propagation model are presented in Section 4.4.

For the experiments, all physical parameters such as frequency, the type of medium and the size of the acoustic source are identical, except for the dimensions of the calculation volume, which are adjusted to simplify performance assessment. The varied parameters are illustrated in Table 4.1 and defined in Sections 3.3 and 4.1.

For instance, N_x, N_y, N_z are kept equal to produce a cubic volume. Furthermore, to ensure that in Equation (3.2), $W_t \in \mathbb{Z}$, the number of points in X, Y and Z are set to 64 each for the first experiment; 128 each for the second experiment; and 256 each for the third experiment. The step sizes in the X, Y and Z directions are set to 0.5 mm for the first experiment, 0.2 mm for the second experiment and 0.1 mm for

the third experiment. Similarly, the elements sizes for the acoustic source are set to 0.5, 0.2 and 0.1 mm for the first, second and third experiments, respectively. Note, however, that there is no relationship between the step sizes and the element sizes. The equivalence of the numbers is simply a matter of convenience and an indication of the progressive increase in the precision and efficiency of the computations.

The number of threads and processors (N_t and N_p , respectively) are both set to 8 for all experiments. For example, in the first experiment with the cluster implementation, $T = 64 \cdot 64 \cdot 64 = 262144$ points and $N_p \cdot N_t = 8 \cdot 8 = 64$ threads. Hence, W_t from Equation (3.2) is equal to 512 points. Subsequent to the computation of the total acoustic field, each process stores only that portion of the intensity field assigned to it in a separate file.

4.1 Performance Metrics

A certain metric is defined as efficiency to reflect a relative improvement (or decline) in the precision of the linear acoustic wave simulation over a certain length of computation time:

$$E = \frac{P_p/t_p}{P_s/t_s}, \quad (4.1)$$

where P indicates the precision and $P = 1/(S_n + \textit{esize})$. The variable *esize* refers to the width and height in millimeters of each square element, while t is the computation time. The subscripts p and s indicate respective values for a parallel implementation and the sequential implementation. For fixed values of precision, Equation (4.1) is essentially a measure of the relative computation times of a parallel implementation and the sequential implementation. That is, $E = t_s/t_p$, which is

simply the speedup (as defined in [58]) in computation time achieved when parallel computation is applied compared to the execution time of the sequential implementation of the algorithm.

Tables 4.2 and 4.3 also contain values for the Karp-Flatt (K-F) metric, defined in [59] as:

$$e = \frac{1/E - 1/N_p}{1 - 1/N_p}. \quad (4.2)$$

The K-F metric provides a measure of parallelization where problem sizes or datasets may increase in size [59], marking it as an amenable metric for algorithms that exhibit data-level parallelism. The value of e provides an indication of the effect of different workload or input sizes in a parallelized implementation. Note that if efficiency, E , exceeds the number of processors, N_p , the K-F metric, e , becomes negative. When $E = N_p$, the value of e is zero and when $E < N_p$, the value of e is positive.

4.2 Multithreaded and Cluster Computing

RQCHP is a network for HPC solutions for researchers in Quebec, Canada. It encompasses five different institutions of higher learning within Quebec: Université de Montréal, Université de Sherbrooke, Concordia University, École polytechnique de Montréal and Bishop’s University. RQCHP also collaborates with Compute Canada, an umbrella organization that enables access to several HPC initiatives for researchers throughout Canada.

The cluster utilized for the purposes of conducting the experiments with the multithreaded and cluster computing implementations is named Mammoth-Serial II and

features 308 nodes, each with 2 Intel Xeon E5462 quad-core CPUs. It is accessed through the RQCHP [2].

In total, there are 2464 cores, 5.6 TB of RAM and 75.2 TB of disk space. The operating system installed on the cluster is a variant of Linux known as CentOS version 5. The compiler utilized for the compilation and optimization of the sequential algorithm is the Intel C/C++ Compiler (ICC) version 10.1. The simulation parameters utilized for each of the three experiments are outlined in Table 4.1.

The results for the computation times utilizing multiple threads and multiple processes with multiple threads in a cluster are presented in Table 4.2 and Table 4.3 and Figures 4.1 and 4.2. Table 4.2 summarizes the total computation times for the sequential, multithreaded and cluster implementations of the model without the application of quarter-field computation. With a precision of 1 mm^{-1} , the multithreaded implementation has a speedup that nearly equals the number of threads, N_t , which is 8. In the remaining two cases, however, the speedup slightly exceeds the total number of threads. There are multiple reasons that may have given rise to this phenomenon.

An explanation for this behaviour may stem from the nature of the CPUs and their caches. Particularly, the CPUs utilized for the simulations belong to the family of server processors featuring 32 kB of L1 instruction caches and 32 kB of L1 data caches for each physical core [60]. The Intel Xeon E5462 also contains two 6 MB L2 caches, a total of 12 MB of L2 cache available to each physical core [60]. Given the repetitive nature of the algorithm, the availability of large and fast L1 and L2 caches can provide the added performance benefits necessary for speedup beyond the number of available threads.

Since there is no thread affinity, a thread may complete execution on one core and spawn on another core. In many cases, this is detrimental as thread creation adds

Table 4.2: Sequential, multithreaded and cluster computation times with their respective Karp-Flatt metric values for 8 and 64* threads without quarter-field computation.

Sequential		Multithreaded		
P (mm^{-1})	Time (s)	Time (s)	E	K-F metric
1	14.96	2.09	7.16	1.68E-02
2.5	705.76	83.84	8.42	-7.09E-03
5	22356.91	2657.78	8.41	-6.99E-03

Sequential		Cluster		
P (mm^{-1})	Time (s)	Time (s)	E	K-F metric
1	14.96	0.43	34.79	1.33E-02
2.5	705.76	10.70	65.96	-4.71E-04
5	22356.91	332.95	67.15	-7.44E-04

* 8 processes (on separate CPUs) with 8 threads each produces a total number of 64 threads, though they are executing asynchronously.

unnecessary overhead and the locality of the cache is changed. However, data-level parallelism suggests that any thread may benefit from the instruction cache of any other thread when identical instructions are being executed by all threads. Thus, a secondary indication of the non-linear speedup may be a direct product of the inherent nature of data-level parallelism, where identical instructions are executed over a varied dataset on the CPU.

For the cluster implementation, the speedup exceeding the total number of threads is likely due to the reasons described previously but also perhaps because of the averaged calculations. To explain further, 8 processors are utilized for the cluster implementation and 8 jobs were submitted to the cluster. Each job subsequently runs asynchronously, perhaps finishing at different times dependant on when each job is executed.

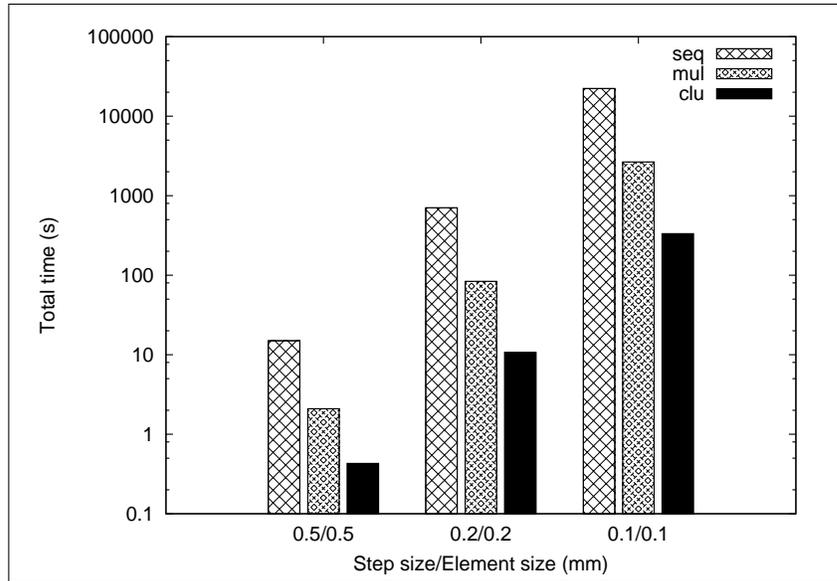


Figure 4.1: Sequential, multithreaded and cluster computation times for step sizes and element sizes of 0.5, 0.2 and 0.1 without quarter-field computation enabled.

However, the computation time for a single job is an indication of the total computation time for 8 jobs, were they executed synchronously on 8 processors. Thus, an average of the computation times for each job provides a metric for estimating the runtime of the algorithm executed synchronously on 8 processors. This averaging process may then introduce marginal differences in the computation times for the cluster implementation.

From Table 4.2, a few important points may be discerned regarding the K-F metric values. Firstly, all values are nearly zero, which indicates that the algorithm exhibits a high degree of parallelism. A second point to note is that in all cases, the values for the K-F metric decrease as the workload is increased by increasing the precision of the simulations. This indicates that the workload for the first experiment is small enough that the overhead of parallelization becomes noticeable. In the second and third experiments, the workload is high enough that benefit of parallelization outweighs the costs associated with overhead.

Table 4.3: Sequential, multithreaded and cluster computation times with their respective Karp-Flatt metric values for 8 and 64* threads with quarter-field computation.

Sequential		Multithreaded		
P (mm^{-1})	Time (s)	Time (s)	E	K-F metric
1	3.9	0.8	4.88	9.16E-02
2.5	175.61	21.1	8.32	-5.54E-03
5	5919.85	663.71	8.92	-1.47E-02

Sequential		Cluster		
P (mm^{-1})	Time (s)	Time (s)	E	K-F metric
1	3.9	0.27	14.44	5.45E-02
2.5	175.61	2.82	62.27	4.40E-04
5	5919.85	83.20	71.15	-1.60E-03

* 8 processes (on separate CPUs) with 8 threads each produces a total number of 64 threads, though they are executing asynchronously.

Table 4.3 signifies that applying quarter-field computation results in a further enhancement to the degree of optimization, where computation times are nearly 4 times shorter in most cases, while the same degree of precision and accuracy is maintained. Note however, that for a precision of $1\text{ }mm^{-1}$ the threads complete execution too quickly and the speedup is below the theoretical threshold. This indicates that the overhead due to thread creation, thread migration and context switching is greater than the resultant performance benefits.

4.3 Graphics Processing Units

The simulation parameters for the GPU experiments are identical to those utilized for the previous experiments. Experimentation begins with a low precision calculation volume with N_x , N_y and N_z all equal to 64 points. For all experiments with the GPU

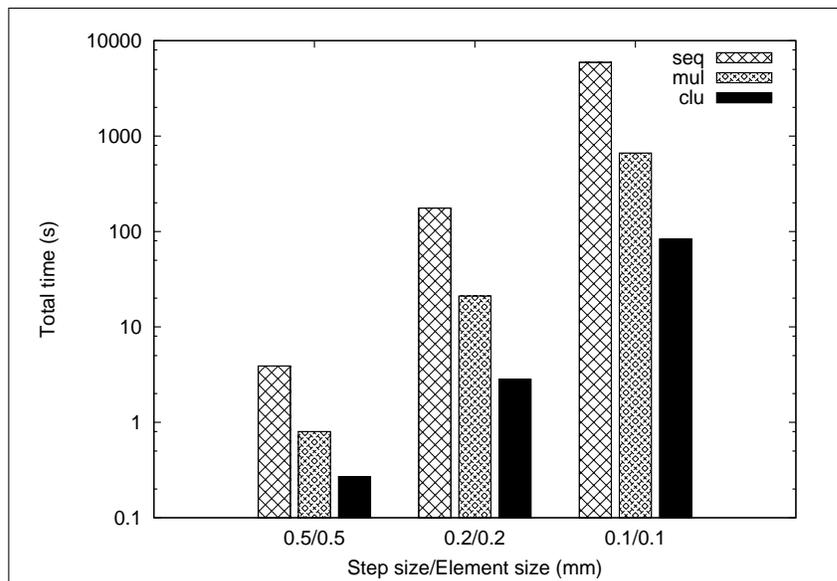


Figure 4.2: Sequential, multithreaded and cluster computation times for step sizes and element sizes of 0.5, 0.2 and 0.1 with quarter-field computation enabled.

implementation, quarter-field computation is enabled, halving the number of points that lie along the X and Y axes.

It should be noted that while the number of points in the calculation volume is progressively increased, the physical dimensions of the calculation volume remain approximately the same as indicated in Table 4.1. While maintaining approximately the same physical dimensions, the number of points is adjusted by increasing the precision, P_p and P_s , of the sequential and GPU-based implementations, respectively. The element size, $esize$, and the step sizes, S_n , are progressively decreased.

The GPU utilized for the experiment is a Tesla T10 processor by Nvidia with CUDA compute capability version 1.3. The GPU consists of 30 SMs, each with 8 SPs for a total of 240 scalar processors or CUDA cores. The memory consists of 4 GB of 512-bit Graphics Double Data Rate 3 (GDDR3) DRAM capable of a maximum bandwidth of 102 GB/s. The device-host interface is a PCIe x16 connection with a

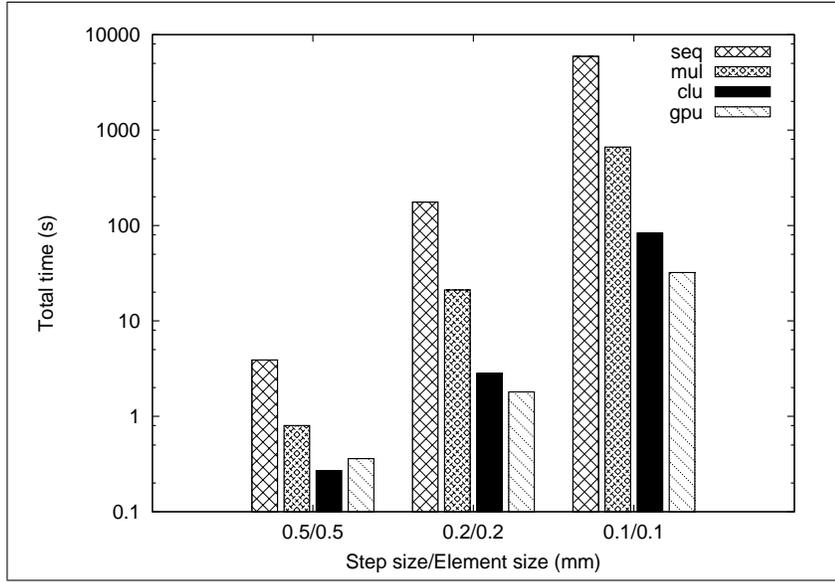


Figure 4.3: A comparison between the GPU execution times and the execution times of the other implementations.

maximum bandwidth of 16 GB/s.

The GPU is located on a single rack along with three other GPUs of identical type, which is itself a part of a larger server consisting of multiple racks designed for GPU computing. The GPU server is accessed through an Secure Shell (SSH) client and is part of SHARCNET. SHARCNET is a consortium of 19 universities and affiliates in southern Ontario that provide access to multiple clusters of HPC servers, which are interconnected with a high-speed wide-area network [1]. A primary goal of SHARCNET is to enable and support academic research. SHARCNET, with RQCHP, are two of seven consortia that provide access to HPC infrastructure across Canada. The seven consortia collectively operate under the umbrella organization known as Compute Canada.

In terms of the CUDA parameters, the length and width of each block is set to 16 threads. Each block contains 256 threads, which provides a natural mapping between points in the calculation volume and the threads in each block. The total

Table 4.4: Sequential and GPU computation times with their respective Karp-Flatt metric values.

P (mm^{-1})	Sequential		GPU	
	Time (s)	Time (s)	E	K-F metric
1	3.9	0.36	10.83	8.85E-02
2.5	175.61	1.80	97.56	6.11E-03
5	5919.85	32.12	184.30	1.26E-03

number of blocks in the grid varies per experiment. As described in Section 3.6.2, the GPU representation of the calculation volume involves the adjacent placement of all planes from left to right. Each plane is divided horizontally and vertically into blocks to create an equal number of points within each block. The number of blocks along the X direction of the grid is computed with the following formula: $npx / \text{blockDim.x} * npz$. For the number of blocks in the Y direction of the grid, the following formula is applied: $npz / \text{blockDim.y}$.

Since the execution times when parameters values are stored in either constant or shared memory are identical, either type of memory storage is suitable for comparison to the other implementations.

Because shared memory is located on-chip as opposed to being located off-chip in DRAM, parameter values are stored in shared memory to produce the execution times depicted in Figure 4.3 and Table 4.4.

Note that in the case where step sizes and element size equal 0.5 mm, the performance of the GPU implementation is worse than the cluster implementation. With that level of precision, the overhead of configuring the GPU and the inability to hide the effects of high latency operations outweighs the benefits of GPU computation. Despite an execution time marginally slower than the cluster implementation in the

case when the precision is set to 1 mm^{-1} , the GPU implementation still outperforms all implementations in the remaining two cases.

The speedup achieved with the GPU implementation is higher than that of the other implementations. The Tesla GPU utilized for the experiments contains 30 streaming multi-processors with 8 scalar processors each resulting in a total of 240 scalar processors. Given that switching between threads has zero overhead in the GPU, a perfect speedup would equal the number of scalar processors or, the maximum speedup is 240. The actual speedup achieved is below this limit in all experiments.

It may be observed, however, that speedup increases as the precision of the acoustic source and the calculation volume is increased. Two sets of factors may be attributed to the progressive increase in speedup observed in the GPU implementation.

The first set is the association between a decrease in the element size of the acoustic source and the increase in the arithmetic intensity of the field computation for a single point in the calculation volume. As the size of each element is decreased, the number of elements increases and therefore, a greater number of elements, each with multiple sub-elements, must be processed to compute to the intensity value at a point in the calculation volume. Consequently, the arithmetic intensity, or, the execution time of a single thread on the GPU is increased.

The second set is the relationship between the number of points in the calculation volume and the number of threads spawned to compute the intensity values at those points. Another reason for an increased speedup in the third experiment is due to a high number of points in the calculation volume. This is due to the design of GPUs as high throughput devices.

In the first experiment, the total number of threads equals: $N_z \cdot N_x/2 \cdot N_y/2 = 64 \cdot 32 \cdot 32 = 65536$ threads. The total number of threads in the third experiment is calculated similarly and equals: $256 \cdot 128 \cdot 128 = 4194304$ threads. The number of

threads in the third experiment is 64 times greater than that in the first experiment, which provides a workload great enough to hide the time required for high latency operations.

Recall from Section 2.6.3 that a warp of 32 threads is executed at once by any single SM. When a high latency operation, such as a fetch from global memory, is issued, the SM switches to another warp of 32 threads that are ready for execution. The ability of the GPU to hide high latency operations is thus dependant on the availability of a high volume of threads, which is exactly the case in the third experiment where 4194304 threads are spawned, one for each point in the calculation volume.

A third factor that affects performance on the GPU is known as *occupancy* [61, pp. 44-46], a metric that defines the number of thread blocks that occupy each SM. The number of threads per block, registers used by each thread and the amount of shared memory required by each block limit the maximum number of blocks that may occupy each SM. As a result, the number of active warps per SM is also limited. Physically, the maximum number of active warps per SM is 32. The number of registers required by each thread in the GPU implementation is 32. With 256 threads per block, the number of active warps is limited to 16 active warps per SM in the following manner:

$$\frac{256 \text{ threads / block}}{32 \text{ threads / warp}} = 8 \text{ active warps / SM} \quad (4.3)$$

$$\frac{16384 \text{ registers available / SM}}{256 \text{ threads / block} \cdot 32 \text{ registers / thread}} = 2 \text{ active thread blocks / SM} \quad (4.4)$$

$$8 \text{ active warps / SM} \cdot 2 \text{ active thread blocks} = 16 \text{ active warps / SM.} \quad (4.5)$$

Increasing the number of threads per block to 512 only reduces the number of

active thread blocks to one due to the requirement of 32 registers per thread. A block with 512 threads thus results in an identical number of 16 active warps per SM. Given that the maximum number of active warps supported by the Tesla architecture is 32, the occupancy in the current GPU implementation is limited to 50% according to the following equation:

$$\frac{16 \text{ active warps per SM}}{32 \text{ maximum active warps per SM}} \times 100\% = 50\%. \quad (4.6)$$

A low occupancy is not necessarily indicative of poor performance as it results in a greater number of registers available to each thread block assigned to a SM [61, pg. 47]. However, in the GPU implementation, the effects of low occupancy are more apparent in the first and second experiments than the the third experiment. This is because the number of threads in the first and second experiments is insufficient to hide high latency operations in the kernel.

4.4 Multi-layer Simulation

The speedups achieved with any of the HPC implementations lend themselves to the possibility of increasing the complexity of the simulation process. Such a possibility is explored when the simulation process for a single homogeneous layer is extended to multiple homogeneous layers by collecting information related to certain properties of the propagating wave at an interface plane separating two layers or two media.

It should be noted that the results presented and discussed are preliminary but support the plausibility of a multi-layer extension of the single layer model. To test the validity of the multi-layer model, the acoustic fields generated with the single layer and multi-layer models are compared. For the single layer medium, a material

with properties similar to that of tissue is utilized. Two layers are utilized for the multi-layer model, the properties of which, are identical to the properties of the single layer tissue medium. It is instructive to imagine a block of tissue for the single layer model and introducing a slice parallel to the X-Y plane at a certain location along the Z-axis. The X, Y and Z limits are -5 to +5 mm, -5 to +5 mm and +15 to +45 mm respectively.

A concave acoustic source is focused such that the geometrical focal spot is located in the second layer of the multi-layer model such that waves propagate through the first layer and through the interface into the second medium to create a region of high intensity near the focal spot. The highest intensity value that is observed is called the peak intensity value. Figure 3.6 illustrates the experimental setup. The location of the interface plane is fixed at 18.6 mm, transducer geometry remains constant and only the center frequency is adjusted for each test.

The effects of multi-layer wave propagation on the peak intensity values at the acoustic focal spot are studied and the results, for different center frequencies, are depicted in Figure 4.4. The disparity between peak intensity values continues to increase until a frequency of 6 MHz, at which point the difference stabilizes and begins to steadily decrease. With minor exceptions for frequencies at and below 0.25 MHz, the peak intensity values in the multi-layer model are generally less than the peak intensity values produced with the single layer model.

Despite the inequalities between the peak intensity values produced with the single layer and multi-layer models, the shape of the resultant line plots in Figure 4.4 indicates a strong correlation between the two sets of values.

The location of the interface plane also affects the peak intensity value that is then

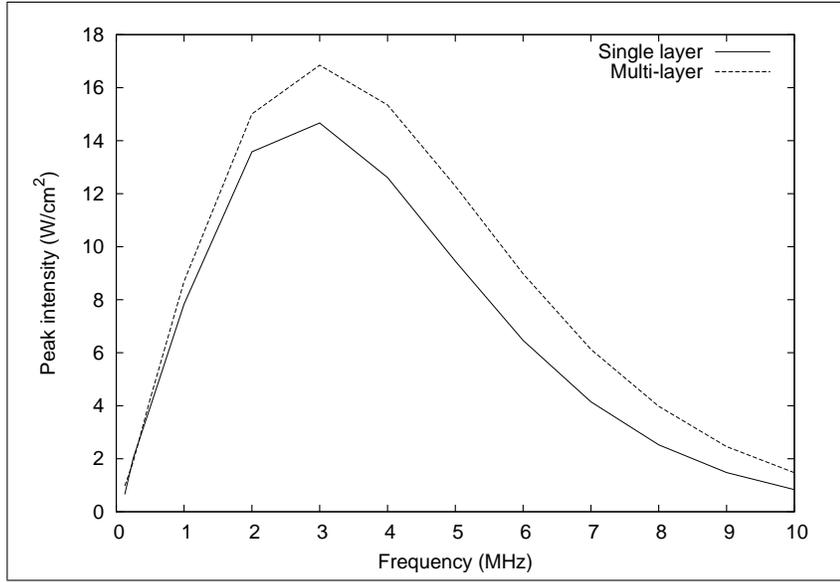
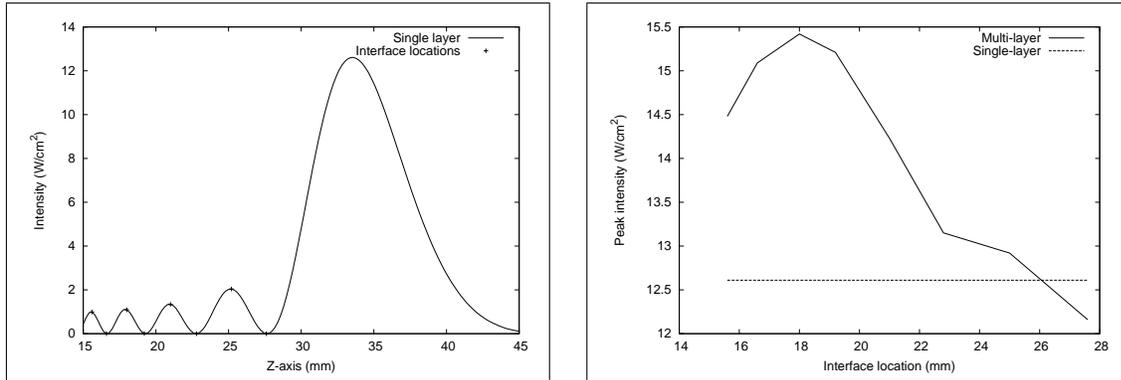


Figure 4.4: A comparison of peak intensity values observed with the single layer model and the multi-layer model at varying frequencies and a fixed interface location.



(a) Axial profile along the Z-axis of intensity values produced with a 4 MHz center frequency. **(b)** Peak intensity values observed in the second layer with varied interface locations.

Figure 4.5: Plots depicting interface locations and the peak intensity values observed at those interface locations. The interface locations in 4.5a are selected based on the presence or lack of intensity at a particular point on the Z-axis.

observed in the medium into which the wave propagates. In the second experiment, multiple tests are performed by placing the interface at 8 locations along the Z-axis. The interface locations are determined based on whether or not a crest or a trough is observed at that point in the axial profile along the Z-axis, as illustrated in Figure

4.5a. The axial profile is produced utilizing the single-layer model with X, Y and Z limits identical to those defined in the previous experiment.

The interface locations corresponding to the first 8 crests and troughs in Figure 4.5a are: 15.6, 16.6, 18, 19.2, 21, 22.8, 25.2 and 27.6 mm. The results, depicted in Figure 4.5b, indicate that the location of the interface produces discernible effects on the peak intensity value observed in the second layer.

Multiple reasons may account for this phenomenon, which include the increasing effects of attenuation as distance is increased, the angle phase velocity of the waves at certain locations along the Z-axis and also the power of the signal at those locations. In actual physical scenarios, the location of the interface between two identical media should not produce any differences in the peak intensity observed in the second layer. Thus, further study is required to assess and improve the current multi-layer model.

4.5 Linear Acoustic and Temperature Simulator

Features of LATS are presented here. Though the cluster computing and GPU implementations are not accessible through the LATS software package, the multithreaded implementation is accessible and the number of threads may be configured prior to executing any simulations.

LATS also contains functionality for viewing the acoustic fields generated following the simulation process. The features of LATS are organized in the following sections in the sequence that a user would proceed through the GUI.

For instance, the user must first configure LATS, then input simulation parameters for acoustic field simulation and may only then proceed to view the acoustic field. The additional step of applying the BHTE [16] described in Section 2.2 is optional but may be utilized to study the effects of heat absorption and temperature rise in

tissue media.

4.6 Configuration

First-time users of the simulation GUI must begin at the configuration page as depicted in Figure 4.6a. The functionality of this page is outlined below:

- **Location of simulation programs** must point to the folder where the supporting programs for the simulation are situated. This is where programs, which the simulation is dependent upon, are contained. These include *field_2008*, *bhte_2008*, etc.
- **Location of saved output files** points to the directory which will be used for writing to any output files that are produced as a result of the simulations.
- **Output filenames' prefix** is a string of characters that will act as a prefix to any of files that are created as output. For instance, a prefix of *crystal* will produce *crystal.mat* and *crystal_field.mat* in the output directory. Note that the prefix string must contain only valid filename characters, which is dependent on the filesystem.
- **Computation configuration** selects a method for the computations. **Sequential** utilizes one thread and is comparably slower than the **Multi-threaded** computations. In multi-threaded computations, multiple threads are initiated to compute the calculation volume or the acoustic field, thus yielding faster results on architectures that support multi-threading. The number of threads are defined in the **Number of threads** field. Note that, in general, this value should be equal to the number of logical or physical cores available on the CPU.

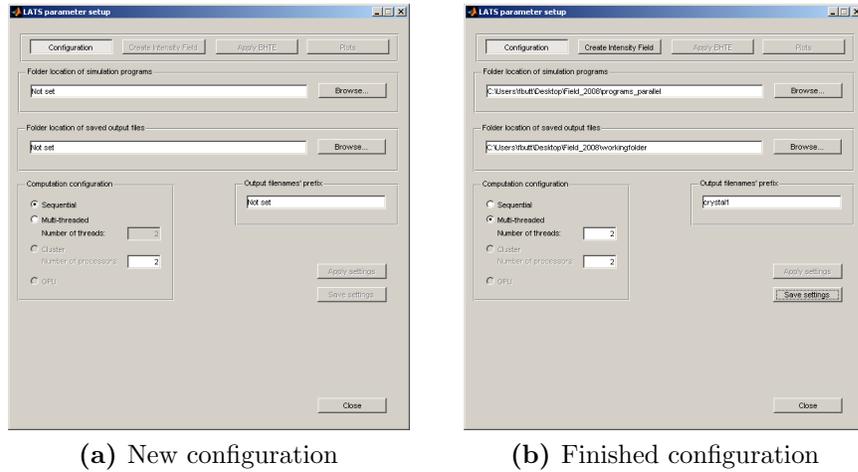


Figure 4.6: An empty (4.6a) and complete (4.6b) configuration page in the simulation GUI.

The disabled options of **Cluster** and **GPU** are included for completeness and to provide an indication of features that may be included in the future.

The *Browse...* buttons permit navigation to the desired directories for each of the parameters. Clicking on the *Apply settings* button loads the values from each parameter into the program and enables the *Save settings* and *Create intensity field* buttons.

Clicking on *Save settings* saves the configuration settings to a file named *sim-config.mat* and subsequent runs of the simulation GUI will no longer require re-configuration of the settings.

With configuration completed, the page should appear as in Figure 4.6b and clicking on the *Create intensity field* button moves to the next page in the simulation GUI.

4.7 Create Intensity Field

The parameters on this page describe properties of the medium and the transducer along with dimensions of the calculation volume. The following is an explanation of each parameter on *Create intensity field* page.

- **Total acoustic power** is the rate at which the transducer does work and is measured in Watts (W). For computations, this field is always set to 1 W. A linear relationship between intensity (W/cm^2) and power (W) is applied through the **Acoustic output power** field on the *Plots* page. For example, if the **Acoustic output power** is set to 2 W, the plotted intensity values will be twice as high when compared with intensity values plotted at 1 W.
- **Center frequency** is measured in megahertz (MHz) and describes the frequency of the ultrasonic waves' propagation.
- **Integration factor** is a unitless constant utilized in the calculation of the Rayleigh diffraction integral.
- **Element size** is a measurement, in millimeters (mm), of the size of each element in the group of elements that form the transducer geometry.
- **Element spacing** describes, in millimeters (mm), the amount of space to include between each element in the group of elements that form the transducer geometry.
- **Transducer size in x** describes, in millimeters (mm), the physical size of the transducer in the x-direction.
- **Transducer size in y** describes, in millimeters (mm), the physical size of the transducer in the y-direction.

- **Curvature radius** describes, in millimeters (mm), the location of the focal spot relative to the center of the transducer. The location is varied by manipulating the curvature of the transducer.
- **Speed of sound** in meters per second (m/s), is the speed at which sound travels in the given medium.
- **Density** in kilograms per cubic meter (kg/m³), is the density of the medium.
- **Attenuation coefficient** is input in decibels per centimeter megahertz (dB/cm·MHz) and is a given value describing the loss in the medium due to attenuation. The default value of 0.1560 is ideal for skin. Note that the values for the attenuation coefficient under the *Create intensity field* tab and the *Apply BHTE* tab are independent. The attenuation coefficient used for intensity field calculations must have a correction factor applied to it prior to running the field simulation if the waves propagate through 2 different mediums. The proper attenuation coefficient for intensity field simulations may be arrived at by utilizing the following weighted formulation:

$$\alpha = \frac{L_1}{L_1 + L_2} \alpha_{ac1} + \frac{L_2}{L_1 + L_2} \alpha_{ac2}$$

where L_1 and L_2 denote the lengths along the Z-axis of medium 1 and medium 2, respectively. α_{ac1} is the actual attenuation coefficient of medium 1 and α_{ac2} is the actual attenuation coefficient of medium 2. For simulations with only 1 layer, it is apparent that $\alpha = \alpha_{ac1} = \alpha_{ac2}$.

- **Attenuation frequency dependency** is a fixed unitless value that must be between 1 and 2 (inclusive). The default value of 1 is ideal for skin. With n defined as the frequency dependency, the calculation of attenuation is based on

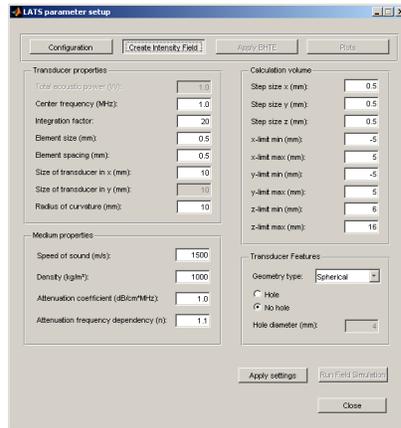
the following equation:

$$\text{Attenuation} = \alpha \cdot f^n$$

where α is the **attenuation coefficient** described previously and f is the **center frequency**.

Note that the value for attenuation under the *Create Intensity Field* tab will be different from the value for attenuation calculated under the *Apply BHTE* tab only if intensity computations are performed for a two layer medium. This is due to the utilization of the weighted average equation for obtaining α for intensity field computations and not utilizing a weighted average for α for BHTE computations. When intensity field computations are performed for a single layer, values for the attenuation coefficients under both the *Create Intensity Field* and *Apply BHTE* tabs are equal, since no weighted average is applied for either attenuation coefficient.

- **Step sizes x, y and z** describe the level of detail with which the computation proceeds over the given volume. Each step size is specified in millimeters (mm).
- **x, y and z limits** construct the extents of the calculation volume, which is a rectangular cuboid or a box shape. The minimum and maximum limits are relative to the center of the transducer, which is placed at (0,0,0). The limits are specified in millimeters (mm). If it is desired to include the focal spot in the calculation volume, it is recommended to set the minimum and maximum z-limits such that they include the curvature radius. For example, the default curvature radius of 10 mm is contained within the default minimum and maximum z-limits of 6 mm and 10 mm, respectively.
- **Hole or no hole** describes a characteristic of the transducer to be created with



(a) Intensity field parameters



(b) Notification of simulation completion

Figure 4.7: Default parameter values on the *Create intensity field* page and notification when simulation is complete.

a certain circular region that has no power. The radius of the hole, created from the center of transducer, is specified in millimeters (mm).

Clicking on the *Apply settings* button, produces a plot of the inputted transducer geometry and clicking on *Run simulation* begins computation of the intensity field. Partial results can be viewed in the Matlab command window and a notification dialog box pops up when the computation is complete (Figure 4.7b).

4.7.1 Setup annular array

Under the *Create Intensity Field* tab, an annular geometry must be selected for the **Geometry type** parameter to enable the simulation of a phased array transducer. Clicking the *Setup annular array...* button displays the annular array setup dialog box depicted in Figure 4.8, where the phase angles are set to 0 and the amplitudes are distributed evenly to represent a non-phased transducer with uniform velocity distribution.

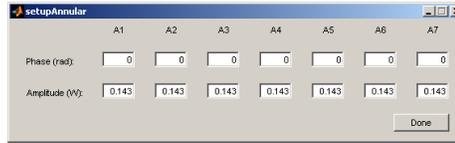


Figure 4.8: Annular array setup dialog box with sample values for the phase angles and amplitudes of 7 annuli.

Examples of values for phase angles and amplitudes in radians and Watts, respectively, are also indicated and may be adjusted as desired for each annulus. The annuli are numbered from 1 to n , where n is the number of annuli set initially in the **Number of annuli** parameter in the *Create Intensity Field* tab. *A1* refers to the inner-most annulus and the labelling proceeds outwards from there. The maximum number of annuli is 7 and the minimum is 2.

Upon completion of phase angle and amplitude entry, clicking on the button labelled *Done*, completes the annular array setup process.

4.8 Apply BHTE

Following the creation of the intensity field, the *Apply BHTE* and *Plots* tabs are enabled. The application of BHTE to the generated intensity field is optional but enables viewing of temperature maps and thermal dosage plots under the *Plots* tab. The parameters for BHTE are outlined below and depicted visually in Figure 4.9a:

- **Skin temperature** is the temperature of the skin and is measured in degrees Celsius ($^{\circ}\text{C}$).
- **Initial body temperature** is the temperature of the body and is measured in degrees Celsius ($^{\circ}\text{C}$).
- **Blood temperature** is the temperature of the blood before heat is applied. It

is measured in degrees Celsius ($^{\circ}\text{C}$).

- **Tissue thermal conductivity** expresses the how well the tissue conducts heat and is measured in watts per meter per degrees Celsius ($\text{W}/\text{m}/^{\circ}\text{C}$).
- **Blood specific heat capacity** is the heat required to create a difference in temperature in a certain quantity of blood. The unit for this quantity is joules per kilogram per degrees Celsius ($\text{J}/\text{kg}/^{\circ}\text{C}$)
- **Blood perfusion rate** is the rate of blood flow through a certain volume over a period of time. It is described by the unit kilograms per cubic meter per second ($\text{kg}/\text{m}^3/\text{s}$).
- **Attenuation coefficient** is input in decibels per centimeter megahertz ($\text{dB}/\text{cm}\cdot\text{MHz}$) and is a given value describing the loss in the medium due to attenuation. The default value of 0.1560 is ideal for skin. Note that the values for the attenuation coefficient under the *Create intensity field* tab and the *Apply BHTE* tab are independent. The attenuation coefficient used for BHTE calculations must not have a correction factor applied to it.
- **Attenuation frequency dependency** is a fixed unitless value that must be between 1 and 2 (inclusive). The default value of 1 is ideal for skin. With n defined as the frequency dependency, the calculation of attenuation is based on the following equation:

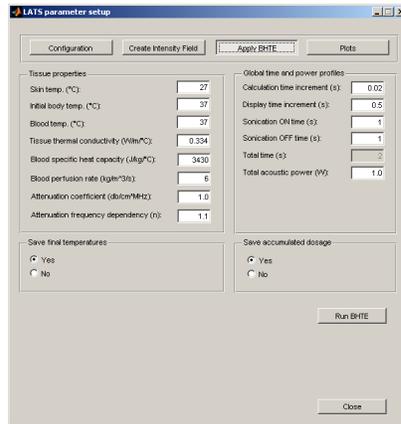
$$\text{Attenuation} = \alpha \cdot f^n$$

where α is the **attenuation coefficient** described previously and f is the **center frequency** utilized for intensity field computations.

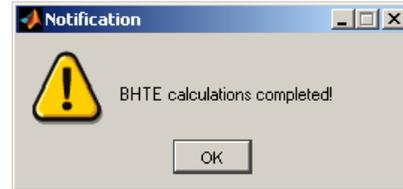
Note that the value for attenuation under the *Create Intensity Field* tab will

be different from the value for attenuation calculated under the *Apply BHTE* tab only if intensity computations are performed for a two layer medium. This is due to the utilization of the weighted average equation for obtaining α for intensity field computations and not utilizing a weighted average for α for BHTE computations. When intensity field computations are performed for a single layer, values for the attenuation coefficients under both the *Create Intensity Field* and *Apply BHTE* tabs are equal, since no weighted average is applied for either attenuation coefficient.

- **Calculation time increment (s)** describes the time interval (in seconds (s)) at which the BHTE sampling is performed.
- **Display time increment (s)** specifies the time interval (in seconds (s)) at which to produce the temperature maps and thermal dosage plots.
- **Sonication ON time** is the amount of time that heat is applied and is indicated in seconds (s).
- **Sonication OFF time** is the amount of time that heat is not applied and is indicated in seconds (s).
- **Total time** is the total of *Sonication ON* and *Sonication OFF* times in seconds (s). This value is automatically updated as a convenience to the user. It cannot be set manually.
- **Total acoustic power** affects the intensity of the heat application and is measured in Watts (W).
- **Save final temperatures** indicates whether or not the final temperature matrix is saved for each BHTE sample point.



(a) BHTe parameters



(b) Notification of simulation completion

Figure 4.9: Default parameter values on the *Apply BHTe* page and notification when simulation is complete.

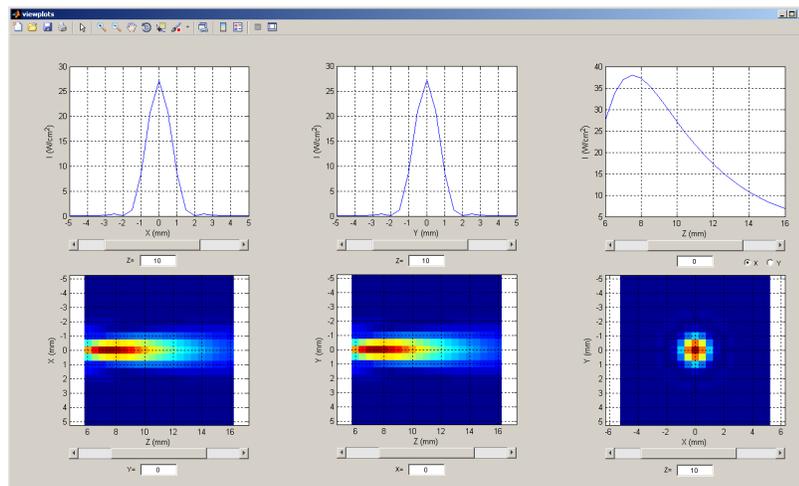
- **Save accumulated dosage** indicates whether or not the accumulated dosage matrix is saved for each BHTe sample point.

4.9 Plots

The *Plots* tab is enabled after an intensity field is created under the *Create Intensity Field* tab. However, the BHTe plots can be viewed only after BHTe simulation is performed under the *Apply BHTe* tab.

The **Acoustic output power** parameter is measured in Watts (W). Through multiplication by the intensity values in the intensity field, this parameter scales the intensity values higher or lower. That is, a peak intensity of 40 W/cm^2 would become 400 W/cm^2 , if the **Acoustic output power** was set to 10 W.

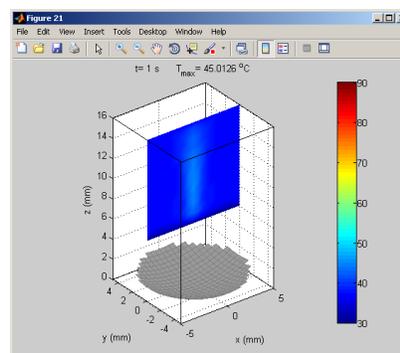
The intensity field plots can be viewed by clicking on the *View intensity field* button. Clicking on the *View temperature map* and/or *View thermal dosage* buttons produces temperature maps and/or thermal dosage plots, respectively. The sampling



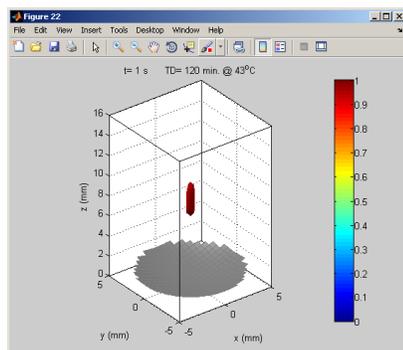
(a) Intensity field plots



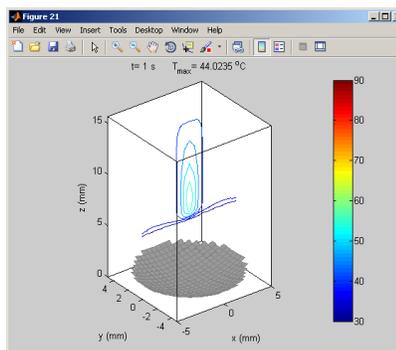
(b) Intensity field parameters



(c) Single temperature map after 1 time interval



(d) Thermal dosage at 1 time interval



(e) Contour plot at 1 time interval

Figure 4.10: Plotting features

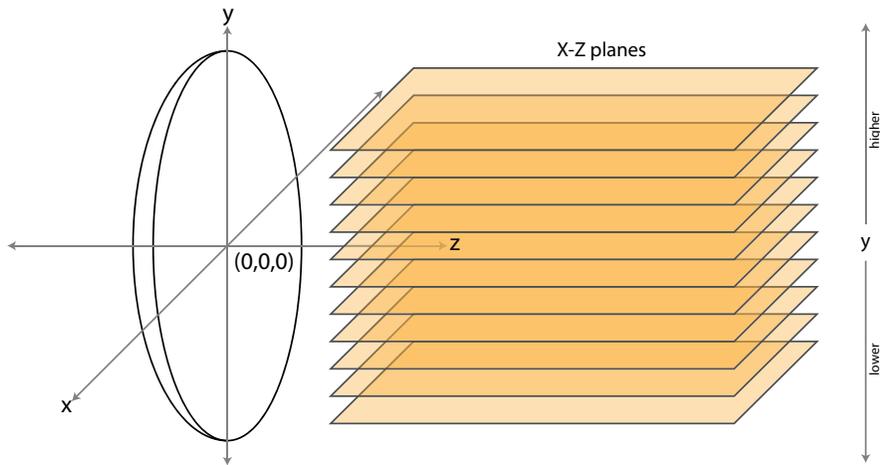


Figure 4.11: X-Z plane colour map plotting explanation

interval for these plots is set through the **Display time increment (s)** parameter under the *Apply BHTE* tab.

Manipulation of the sliders under each plot (Figure 4.10a) adjusts the plane from which the plotting data is extracted. As an example, consider the colour-mapped plot in the second row and first column from the top-left. Adjusting the slider to the right increases the Y-axis value and the plot is updated with a slice of the X-Z plane at that point on the Y-axis. Adjusting the slider to the left decreases the Y-axis value and updates the plot accordingly. An illustration of this is indicated in Figure 4.11. Note that adjusting the Y-axis in this plot affects the intensity profile of the X-axis along the Z-axis in the plot situated at the first row and first column.

Having performed BHTE computations, there are 3 types of plots available for viewing the effects: contour, temperature map and thermal dosage and each type may be viewed by clicking on the appropriate button under the *Plots* tab. The contour plots present outlines that mark a major change in temperature in the field, while the temperature maps provide a smoothed display of temperature variations across a

plane in the calculation volume. Lastly, if the region is heated to a high temperature over an interval of time, the thermal dosage plots depict, in 3D, a lesion that would result from HIFU treatment.

4.10 Summary

This chapter presented the results achieved through the implementation of the parallel computing models described in Chapter 3. Performance enhancements equal to the total number of threads are realized in the first two implementations on CPUs.

In the third implementation on GPUs, the importance of a sizeable workload is highlighted when speedups far below the number of cores in the GPU are observed in the first two experiments. In the third experiment, the number of points in calculation volume are 64 times greater than in the first experiment and 8 times greater than in the second experiment. Because the number of points is equal to the number of threads in the GPU implementation, the computational workload in the third experiment is high and sufficient to hide high latency operations on the GPU.

The HPC implementations of the linear acoustic wave model described by Ocheltree and Frizzell [8] lead to the realization of a multi-layer extension to the same linear acoustic wave model. Though the results do not quite match what is expected based on single layer simulations, the preliminary findings are promising in the development of a more accurate multi-layer linear acoustic wave simulation model.

Chapter 5

Concluding Remarks

In this work, the performance of a linear acoustic simulation model is enhanced in terms of precision and efficiency.

The sequential algorithm is studied and data-level parallelism inherent in the algorithm is exploited with multiple HPC approaches that each offer certain advantages and disadvantages.

With multiple OpenMP threads, the linear acoustic simulation exploits the full capabilities of the multi-core architectures prevalent today. Provided a C compiler supporting OpenMP threads exists for that platform, an added benefit of this approach is its portability as it is capable of executing on both single and multiple core CPUs. However, OpenMP's limitation to SMP architectures restrains the coarse-grained data-level parallelism inherent in the model in terms of the efficiency and precision that may be realized.

A hybrid distributed approach is thus developed with multiple threads but also multiple processes. With local memory available to each CPU that runs one process, a particular advantage of this approach is its scalability. As the number of points is increased in the calculation volume, so too is the spatial and temporal complexity of

the algorithm as more points in the calculation volume must be computed.

By reducing the number of points that each process must compute, a distributed memory approach reduces the spatial complexity of the algorithm per process. This is precisely the case when, for a fixed number of points, the number of processes to compute those points is increased in implementation that relies on cluster computing. Each process computes fewer points and the local spatial complexity of the algorithm per process is reduced. Multiple threads spawned by each process further exploit the data-level parallelism. However, the threads on one physical CPU operate with SMP and all threads share a local memory.

In the final approach, the sequential algorithm is implemented to exploit the parallel computation capabilities of GPUs. Owing to the design of GPU architectures, millions of lightweight threads may be spawned. The performance achieved with the GPU implementation provides an indication of the specificity of their application to parallel computing. The number of registers, the amount of shared memory and other physical hardware limitations impose restrictions on the types of problems that will benefit most from implementation on a GPU.

While the lack of any dependencies between calculations of points in the acoustic model is highly amenable to GPU implementation, the algorithm to compute a single point requires 32 registers, which limits the occupancy in each SM as the registers in each SM must be shared by all active threads. Thus, a high workload, in terms of the number of points, is required to spawn millions of threads to hide the effects of high latency operations. The highest speedup of 185.45 is achieved with the heaviest workload on the GPU and lower speedups are realized with smaller workloads.

With super-scalar architectures coupled with large and fast CPU caches, the speedups realized in certain cases are beyond theoretical limits, which suggests that the speedup is at most equal to the number of processors. The axisymmetric prop-

erties of the acoustic field lead to quarter-field computation, which results in further performance benefits through a 75% reduction in the number of computations. Compared to the baseline sequential algorithm with quarter-field computation, the multithreaded, cluster and GPU implementations with quarter-field computation are, in cases with the maximum workload, 9, 71 and 185 times faster with 8, 64 and 240 cores, respectively.

The enhanced performance of linear acoustic field simulation provides an avenue for further increasing the complexity of the linear acoustic model. A multi-layer linear acoustic model is developed as an extension of the single layer linear acoustic model. Peak intensity values observed at varied frequencies with the single layer model and the multi-layer model, coupled with the effects of interface location on the peak intensity values indicate disparities between the validated single layer model and the multi-layer model. However, the preliminary results demonstrate the plausibility of a realistic multi-layer model.

5.1 Future Work

The execution speed of the algorithm may be improved further through the usage of multiple GPUs without requiring any synchronization between GPUs. Also, connectivity between LATS and the cluster computing and GPU implementations is possible through the development of a separate module that handles communication between a client that is running LATS and a server that will execute the linear acoustic wave simulation.

Certain improvements in the parallel implementations may potentially lead to further enhancements in certain aspects of performance that were not considered in the current work. Memory usage, for instance, may be improved with an increased

usage of HDD space as opposed to DRAM in both the multithreaded and cluster implementations. Disk usage was minimized in the current work since accesses to the disk are slower than accesses to DRAM.

The scalability of all HPC approaches are limited in terms of the number of elements that compose the acoustic source since, prior to computation, all elements are first stored into memory. Potential improvements in the scalability of the algorithm are possible but would likely reduce execution speed if HDD storage is utilized instead of system memory.

The results from the multi-layer model implementation indicate that, while disparities exist when compared to the single layer model, subject to further study, the multi-layer model may produce results that reflect the reality of wave propagation across multiple media with better accuracy.

The current work also naturally lends itself to applications of HPC approaches to nonlinear acoustic wave propagation models such as NLP, which is developed in [52] and the KZK equation which is described in [51]. Such approaches are greatly warranted given the additional complexity of such models as mentioned in Section 2.7.3.

GPU approaches, in particular, may present challenges given the resource intensive nature of nonlinear acoustic wave models and the restrictions on resources in the GPU. With technological developments, future GPU devices will be more flexible though even with current devices parallel computing models for nonlinear acoustic wave propagation are likely possible.

As the scalability of grid computing platforms continues to improve [62], such platforms provide additional means of realizing performance gains on a greater scale.

Bibliography

- [1] SHARCNET. Shared Hierarchical Academic Research Computing Network. <http://www.sharcnet.ca>. [Online; accessed 19-August-2011].
- [2] RQCHP. Réseau Québec de Calcul de Haute Performance. <http://www.rqchp.ca>. [Online; accessed 19-August-2011].
- [3] Compute/Calcul Canada. Compute/Calcul Canada. <https://computecanada.org/>. [Online; accessed 19-August-2011].
- [4] G. R. Harris. Review of Transient Field Theory for a Baffled Planar Piston. *Journal of the Acoustical Society of America*, 70(1):10–20, 1981.
- [5] R. Merritt. Cpu designers debate multi-core future. <http://www.eetimes.com/showArticle.jhtml?articleID=206105179>, February 2008. [Online; accessed 19-August-2011].
- [6] P. Dvorak. The processor future is multicore. *Machine Design*, 80(5):74–80, 2008.
- [7] W. Gropp, K. Kennedy, L. Torczon, A. White, J. Dongarra, I. Foster, and G. C. Fox. *The Sourcebook of Parallel Computing (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, November 2002.
- [8] K. B. Ocheltree and L. A. Frizzell. Sound field calculation for rectangular sources. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, 36(2):242–248, 1989.
- [9] R. S. C. Cobbold. *Foundations of Biomedical Ultrasound*. Oxford University Press, Toronto, 2006.
- [10] F. Butt. *Linear Acoustic and Temperature Simulator (LATS) User Guide*, 1.4 edition, 2011.
- [11] G. ter Haar and C. Coussios. High intensity focused ultrasound: Physical principles and devices. *International Journal of Hyperthermia*, 23(2):89–104, 2007.

- [12] J. Tavakkoli and N. T. Sanghvi. Ultrasound-guided HIFU and Thermal Ablation. In V. Frenkel, editor, *Therapeutic Ultrasound: Mechanisms to Applications*, pages 137–161. Nova Science Publishers, Hauppauge, NY, January 2011.
- [13] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *ACM Transactions on Graphics*, volume 23, pages 777–786, 2004.
- [14] P. R. Stepanishen. The Time-Dependent Force and Radiation Impedance on a Piston in a Rigid Infinite Planar Baffle. *The Journal of the Acoustical Society of America*, 49(3B):841–849, March 1971 1971.
- [15] J. Zemanek. Beam Behavior within the Nearfield of a Vibrating Piston. *Journal of the Acoustical Society of America*, 49(1 pt 2):181–191, 1971.
- [16] H. H. Pennes. Analysis of tissue and arterial blood temperatures in the resting human forearm. *J.Appl.Physiol.*, 1(2):93–122, 1948.
- [17] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21; C-21(9):948–960, 1972. ID: 1.
- [18] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to The Cell Multiprocessor. *IBM J.Res.Dev.*, 2005.
- [19] OpenMP C and C++ Application Program Interface Version 3.0. <http://www.openmp.org/mp-documents/spec30.pdf>, 2008. [Online; accessed 19-August-2011].
- [20] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. ID: 1.
- [21] F. Darema. *The SPMD Model : Past, Present and Future*, volume 2131 of *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 1–1. Springer Berlin / Heidelberg, 2001.
- [22] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.
- [23] T. H Weng, S. W Huang, W. W. Ro, and K. C Li. Implementing FFT using SPMD style of OpenMP. In *Proceeding - 6th International Conference on Networked Computing and Advanced Information Management, NCM 2010*, pages 91–96, 2010.

- [24] G. Krawezik, G. Alléon, and F. Cappello. *SPMD OpenMP versus MPI on a IBM SMP for 3 kernels of the NAS benchmarks*, volume 2327 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2002.
- [25] Z. Liu, B. Chapman, T. H Weng, and O. Hernandez. *Improving the performance of OpenMP by array privatization*, volume 2716 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2003.
- [26] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.
- [27] D. Blythe. Rise of the graphics processor. *Proceedings of the IEEE*, 96(5):761–778, 2008.
- [28] Nvidia Corporation. NVIDIA CUDA Programming Guide. <http://developer.nvidia.com/nvidia-gpu-computing-documentation>, 2010. [Online; accessed 19-August-2011].
- [29] J. Ajanovic. PCI Express* (PCIe*) 3.0 Accelerator Features, 2008. <http://www.intel.com/technology/pciexpress/devnet/resources.htm>.
- [30] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with Streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, page 35, New York, NY, USA, 2003. ACM.
- [31] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.
- [32] L. Odegaard, S. Holm, F. Teigen, and T. Kleveland. Acoustic field simulation for arbitrarily shaped transducers in a stratified medium. In *Proceedings of the IEEE Ultrasonics Symposium*, volume 3, pages 1535–1538, 1994.
- [33] K. Epasinghe. Simulation and Visualization of Ultrasound Fields. Master’s thesis, University of Oslo, 1997.
- [34] J. A. Jensen. A model for the propagation and scattering of ultrasound in tissue. *Journal of the Acoustical Society of America*, 89(1):182–190, 1991.
- [35] J. A. Jensen. Simulation of advanced ultrasound systems using field II. In *2004 2nd IEEE International Symposium on Biomedical Imaging: Macro to Nano*, volume 1, pages 636–639, 2004.

- [36] C. S. Henry, F. Xia, and R. Stevens. Application of high-performance computing to the reconstruction, analysis, and optimization of genome-scale metabolic models. *Journal of Physics: Conference Series*, 180(1), 2009.
- [37] O. Fortmeier and H. M. Bückner. Hybrid distributed-/shared-memory parallelization for re-initializing level set functions. pages 114–121, 2010.
- [38] M. Reumann, B. G. Fitch, A. Rayshubskiy, D. U. Keller, G. Seemann, O. Dossel, M. C. Pitman, and J. J. Rice. Strong scaling and speedup to 16,384 processors in cardiac electro-mechanical simulations. *Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 2009.
- [39] K. H. W. J. Ten Tusscher, D. Noble, P. J. Noble, and A. V. Panfilov. A model for human ventricular tissue. *American Journal of Physiology - Heart and Circulatory Physiology*, 286(4 55-4):H1573–H1589, 2004.
- [40] L. Nyland, J. Prins, R. H. Yun, J. Hermans, H. C Kum, and L. Wang. Achieving scalable parallel molecular dynamics using dynamic spatial domain decomposition techniques. *Journal of Parallel and Distributed Computing*, 47(2):125–138, 1997.
- [41] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, pages 425–432, 2000.
- [42] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: high-speed rendering using image composition. *Computer Graphics (ACM)*, 26(2):231–240, 1992.
- [43] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM Transactions on Graphics*, volume 22, pages 908–916, 2003.
- [44] C. J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, MICRO 35*, pages 306–317, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.
- [45] C. Walbourn. DirectCompute, 14 July 2010 2010. <http://blogs.msdn.com/b/chuckw/archive/2010/07/14/directcompute.aspx>.
- [46] AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide (v1.3c), 2011. <http://developer.amd.com/sdks/AMDAPPSDK/documentation/Pages/default.aspx>.
- [47] Khronos OpenCL Working Group. The OpenCL Specification Version 1.0, 2009. <http://www.khronos.org/registry/cl/>.

- [48] O. A. Kaya, A. Şahin, and D. Kaleci. Pressure field of rectangular transducers at finite amplitude in three dimensions. *Ultrasound in Medicine and Biology*, 32(2):271–280, 2006.
- [49] A. C. Baker, A. M. Berg, A. Sahin, and J. N. Tjotta. The nonlinear pressure field of plane, rectangular apertures: Experimental and theoretical results. *Journal of the Acoustical Society of America*, 97(6):3510–3517, 1995.
- [50] Y. Lee. *Numerical Solution of the KZK Equation for Pulsed Finite Sound Beams in Thermoviscous Fluids*. PhD thesis, University of Texas at Austin, 1993.
- [51] M. F. Hamilton and D. T. Blackstock. *Nonlinear Acoustics*. Academic Press, United States of America, 1st edition, 1998.
- [52] R. J. Zemp, J. Tavakkoli, and R. S. C. Cobbold. Modeling of nonlinear ultrasound propagation in tissue from array transducers. *Journal of the Acoustical Society of America*, 113(1):139–152, 2003.
- [53] J. Tavakkoli, D. Cathignol, R. Souchon, and O. A. Sapozhnikov. Modeling of pulsed finite-amplitude focused sound beams in time domain. *Journal of the Acoustical Society of America*, 104(4):2061–2072, 1998.
- [54] A. Karamalis, W. Wein, and N. Navab. *Fast ultrasound image simulation using the Westervelt equation*, volume 6361 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2010.
- [55] G. F. Pinton, J. Dahl, S. Rosenzweig, and G. E. Trahey. A heterogeneous nonlinear attenuating full-wave model of ultrasound. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, 56(3):474–488, 2009.
- [56] D. Michéa and D. Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. *Geophysical Journal International*, 182(1):389–402, 2010.
- [57] F. Butt, A. Abhari, and J. Tavakkoli. An Application of High Performance Computing to Improve Linear Acoustic Simulation. In *14th Communications and Networking Simulation Symposium (CNS11), Proceedings of the 2011 Spring Simulation Multiconference, SpringSim’11*, pages 71–78, Boston, MA, USA, April 4-7 2011. SCS/ACM.
- [58] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, 30:483–485, 1967.
- [59] A. H. Karp and H. P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, 1990.

- [60] Intel Corporation. Quad-Core Intel Xeon Processor 5400 Series: Datasheet. http://www.intel.com/Assets/en_US/PDF/datasheet/318589.pdf, 2008. [Online; accessed 19-August-2011].
- [61] Nvidia Corporation. CUDA C Best Practices Guide. <http://developer.nvidia.com/nvidia-gpu-computing-documentation>, 2010. [Online; accessed 19-August-2011].
- [62] F. Butt, S. S. Bokhari, A. Abhari, and A. Ferworn. Scalable Resource Discovery Through Distributed Search. *International Journal of Distributed and Parallel Systems*, September 2011. In Press.