UNIVERSITY OF OSLO
Department of Informatics

A Parallel Front Propagation Method

Simulating geological folds on parallel architectures

Master thesis

Mohammed Sourouri

Spring 2012

A Parallel Front Propagation Method

Mohammed Sourouri

Spring 2012

ABSTRACT

Static non-linear Hamilton-Jacobi equations are often used to describe a propagating front. Advanced numerical algorithms are needed to efficiently compute solutions to these non-linear equations. In geological modelling, layers of rocks can be described as the position of a propagating front at different times. A fast simulation of such layers is a key component in exploration software developed by Kalkulo AS for Statoil AS. Developing fast algorithms and solvers is essential in this application, since faster solvers enables users to test more geological scenarios, leading to a better understanding of the inner earth. Front propagation is also used in other applications, such as reservoir simulation, seismic processing and medical imaging, making a fast algorithm highly versatile.

The recent years rise of parallel architectures has made substantial computational resources available. One way to originate faster solvers is therefore to develop algorithms that are able to exploit the increasing parallelism that these architectures offer. In this thesis, a novel three-dimensional anisotropic front propagation algorithm for simulation of geological folds on parallel architecture is presented. The algorithm's abundant parallelism is demonstrated on multi-core CPUs and GPU architectures. Implementation on multi-core architectures is achieved by using the OpenMP API, while the Mint programming model is used to facilitate with the GPU programming.

We demonstrate 7x to 2x speedups running on the Nvidia GeForce GTX 590 GPU, compared with a multi-threaded implementation on a NUMA-machine using two interconnected 12 core AMD Opteron processors. These results point to enormous potential performance advances of our algorithm on parallel architectures.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API	Application Programming Interface
APU	Accelerated Processor Unit
ccNUMA	Cache-Coherent NUMA
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FDM	Finite Difference Method
FEM	Finite Element Method
FIM	Fast Iterative Method
FLOP	Floating-Point Operation
FMM	Fast Marching Method
FSM	Fast Sweeping Method
FVM	Finite Volume Method
GCC	GNU Compiler Collection
GPU	Graphics Processing Unit
ICC	Intel C/C++ Compiler
MIC	Many-Integrated Cores
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
nvcc	Nvidia C/C++ Compiler
OpenCL	Open Computing Language

OpenMP	Open Multi-Processing
PCIe	PCI Express
PDE	Partial Differential Equation
PGI	The Portland Group, Inc
PMM	Parallel Marching Method
Pthread	POSIX thread
PTX	Parallel Thread Execution
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
SMX	Next Generation SM
SOFI	Semi-Ordered Fast Iterative Method
SPMD	Single Program Multiple Data

ACKNOWLEDGMENTS

This thesis would never have materialised without my advisor Tor Gillberg. Despite being very busy, Tor has been extremely generous with his time and helped me whenever the chips were down. His dedication and hard work have inspired me since the day we met. I appreciate his leadership along with that of my mentor Professor Hans Petter Langtangen. Many thanks also to Professor Xing Cai whom I enjoyed working with on the ICCS 2012 paper. Discussions with Xing have been a tremendous help in shaping my research.

I am very grateful for professor Are Magnus Bruaset at Simula School of Research and Innovation AS for the generous funding of my stay at University of California, San Diego (UCSD). Further thanks goes to Dr. Stuart Clark and my fellow colleagues at the Computational Geoscience group at Simula Research Laboratory.

I would also like to thank collaborators from other institutions. Professor Scott Baden (UCSD) and Dr. Didem Unat (Lawrence Berkeley National Laboratory). My sincere thanks goes to the following persons who made my stay at UCSD a memorable one: Alden King, Nguyen Thanh Nhat Tan, Yajaira Gonzalez, Pardis Miri and Alexander Breslow.

I would never have survived without the companionship of my dog, Tussie. I am also truly indebted to my brother for his encouragement and moral support. I owe sincere thankfulness to Karoline for caring about me and keeping me with company. Many thanks for all the memories.

Finally, I would like to thank my parents whom I dedicate this thesis to. Thanks for letting me travel halfway across the world to pursue my dreams and of course, taking care of Tussie while I was away.

Chapter 1

INTRODUCTION

1.1 Motivation

Kalkulo AS, a subsidiary of Simula Research Laboratory, has been developing a geological modelling application in collaboration with Statoil for the past several years. An essential module of this application is the simulation of folded geological structures of Earth's crust. The simulations are done repeatedly in large 2D and 3D models, and for the application to be used in an interactive manner, the simulations must be computed rapidly. Unfortunately, the computational load for simulating a fold is high, and in 3D the lengthy computational times becomes an issue for the user of the application. An impediment for both Kalkulo and Statoil is that most available numerical algorithms for simulating folds are serial.

About a decade ago, processor manufacturers realized that due to energy constraints, serial Central Processing Unit (CPU) architecture had reached a dead end. Instead, processor manufacturers turned their attention to parallel architectures. By simply multiplying the number of processor cores, multi-core CPUs were born. Theoretically speaking, multi-core CPUs can deliver *number of cores* times better performance of a single CPU core. Running serial code on a parallel architecture will unfortunately not automatically lead to increased application performance. Developers must instead rewrite or write new programs that can take advantage of the increased parallelism.

In addition to multi-core CPUss, another computing platform that has gained a lot of traction lately are Graphics Processing Units (GPUs). It is thought that due to their highly parallel architecture, GPUs are better optimized for throughput computing. As we will show in section 6.8, GPUs possess a higher performance per watt ratio than multi-core CPUs. It is expected that this ratio will only increase as GPU manufacturers move to improved chip lithography [27].

Today, GPUs live on cards that are connected to the CPU via the

PCI Express Bus (PCIe). Another distinctive GPU feature is that they have limited memory. Depending on manufacturer and configuration, a typical device has only a couple of Gigabytes of memory. Furthermore, programming GPUs is associated with a low level of programming. For instance, due to the limited memory size, programmers need to divide data in smaller chunks and manually transfer the data from the CPU to the GPU using the PCIe bus. Another characteristic feature of GPUs is that they consist of many (usually hundreds to thousands) simpler and slower cores than traditional CPUs. Hence, a GPU's performance is tightly coupled to the parallelism of the code they execute.

Taking the overall goal of creating an interactive simulator and parallel architectures in account, it is clear that a parallel algorithm needs to be developed. Due to the increasing popularity of heterogeneous computing, an algorithm should not only target multi-core CPUs, but also GPUs. Thanks to the generic nature of the problem, it is thought that other applications such as reservoir simulation, seismic processing and medical imaging, might also benefit from a new and more parallel algorithm, adding further motivation for the research in focus of this thesis.

The main findings of this thesis have been published as a refereed paper in Elsevier's Procedia Computer Science. The paper will be presented by me at the International Conference on Computational Science 2012 in Omaha, Nebraska. The acceptance rate for ICCS is 30 percent [32].

1.2 Contributions

- We present a novel 3D front propagation algorithm for simulating folds by solving a static Hamilton-Jacobi equation.

- We implement the algorithm for two of the most common parallel architectures available today, multi-core CPUs and many-core GPUs. The multi-core implementation is performed using the OpenMP API, while the many-core implementation is done using an automated C-to-CUDA source code translator called Mint.

- We have demonstrated the effectiveness of our algorithm running on a 24-core multi-core CPU and a Nvidia GeForce GTX 590 GPU. Specific optimizations for NUMA-machines are also presented.

- Finally, we discuss the possibility of running the algorithm on future parallel architectures. It is hypothesized that today's heterogeneous computing model will be further extended and increase in popularity. Due to the embarrassingly parallel nature of the algorithm, we believe the 3D PMM can be easily adapted to future parallel architectures.

1.3 Document organization

- Chapter 2 provides the mathematical framework which our algorithm will later be based on. A short introduction to partial differential equations, wavefront propagation and the Eikonal equation is given.
- Chapter 3 provides an overview of some of the latest trajectories in parallel computer architecture today. The architectures of a typical multi-core CPU and a GPU are discussed in detail. Some of the challenges facing programmers who wish to develop parallel applications are also discussed. The closing part of the chapter briefs the reader about the multi-core programming interface OpenMP, and the directives-based GPU programming model, Mint.
- Chapter 4 discusses two groups of numerical methods for anisotropic propagation, namely front tracking methods and sweeping methods. One method from each of the two groups is discussed in detail. Parallel possibilities of several algorithms are also discussed.
- Chapter 5 gives the reader an introduction to geological fold modelling, and the role front propagation plays with respect to fold modelling. For motivation purposes, various front propagation algorithms are compared with each other, before the 3D PMM algorithm is presented and discussed in more detail. The chapter is rounded off with implementation details from OpenMP and Mint.
- Chapter 6 presents the results obtained by running the algorithm using OpenMP on two interconnected 12-core AMD Opteron processors, and on a Nvidia GeForce GTX 590 GPU using Mint. In addition, specific optimizations for NUMA-machines is presented and discussed.
- Chapter 7 discusses some of the current limitations of the 3D PMM algorithm and how they can possibly be addressed in the future. A sneak peek into forthcoming parallel architectures is also presented before the thesis is brought to a conclusion.
- Appendix A is a conference paper for the International Conference on Computational Science, ICCS 2012 that I have co-authored with Tor Gillberg and professor Xing Cai. Portions of this thesis is based on this paper. This thesis contains new information not published in the original paper. I will present the paper at the ICCS 2012 conference on June 6th 2012 in Omaha, Nebraska.
- Appendix B contains results from experiments conducted with different parameters.

Chapter 2

BACKGROUND AND OVERVIEW

2.1 Differential Equations and Solution Approximation Methods

Calculus of infinitesimal changes was most likely independently developed by Newton and Leibniz in the late 17th century. Compared to earlier mathematics, the notation of Leibniz allowed a much more compact formulation of mathematically described observations. The calculus of infinitesimal changes is a mathematical framework for describing relations of functions and their derivatives. One example is the relationship between position p and velocity v,

$$v = \frac{d}{dt}p, \tag{2.1}$$

$$p(0) = 0. \tag{2.2}$$

equation (2.1) is an *ordinary differential equation*, an expression relating changes in time of one property to another. With a constant velocity and the *initial condition* (2.2), the position at later times can be computed by solving equation (2.1).

If the equation instead relates small changes in space, the equation is simply called a *differential equation*. Mixed expression of both time and space derivatives are known as Partial Differential Equations (PDEs).

2.2 Partial Differential Equations

PDEs are often used to track the evolution of a system such as the rate of change of a wave in space and time. A typical example PDE is the wave equation in 1D as shown in equation (2.3),

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial^2 x^2}, \quad x \in (0,1), \quad t > 0 \tag{2.3}$$

where u describes the shape of the travelling wave, while c is the wave's velocity.

Depending on the problem to be solved, a PDE usually require specification of both boundary conditions *and* initial conditions. A PDE can usually have many solutions and it is not uncommon that a PDE has infinitely many solutions. Moreover, most PDEs do not have analytical solutions, hence, they need to be solved by numerical approximation methods using computers.

2.3 Solution methods

Three popular numerical methods for approximating PDEs are the Finite Difference Method (FDM), Finite Element Method (FEM) and the Finite Volume Method (FVM). In all of these methods, the first step involves a *discretization* of the equation to be approximated. How the actual discretization is performed, depends on the method. In this thesis we will look at how discretization is performed for the FDM and the FEM. The discussions of the FVM discretization are outside the scope of this thesis, and will therefore not be discussed further. A thorough introduction to numerical methods can be found in [28].

2.4 The Finite Difference Method

Discretization in the FDM is done by introducing a grid with discrete values on a set of nodes in space and time. The next step consists of replacing the derivatives of the equation with a finite difference approximation, often referred to as a numerical *scheme*. In short, the basic idea behind FDM is to sample the sought entity at a set of points, and approximate the mathematical equation at those points with the aid of a numerical scheme. At some points values are given from the initial condition, and values for all other unknown nodes are solved for through the discretized equation.

One area where FDM is of practical interest is wave propagation problems. Simulations of wave propagations are needed in a variety of fields, including medical imaging and geophysics. The FDM can be used to simulate how waves diffuse in space and time.

2.5 The Finite Element Method

The core idea of the Finite Element Method is to approximate the unknown function u, as a linear combination of a set of predefined functions. That is, u, is approximated as $\hat{u}$ such that

$$\hat{u} = \sum_{j=1}^{M} u_j N_j(x). \tag{2.4}$$

$N_j(x)$ are referred to as the *basis function*s, while u_j are the unknown coefficients, as shown in equation (2.4). As with almost every numerical approach, the ultimate goal for FEM is to minimize the difference between the exact solution and the numerical solution.

Before FEM can be applied to a PDE, the PDE must be restated into an integral form, referred to as the *weak* form. A weak form of the problem is obtained by multiplying the PDE with a test function, that is a function with a set of assumed properties. After the multiplication, the resulting equation is reformulated, most commonly by integrating by parts.

It is worth noticing that in the FDM domain, there is no need to break the PDE into a weak form, because the equation is directly converted to a set of discrete equations. Some might argue that the need for a weak form accompanied with other subtle points such as the difference between different boundary conditions, contributes to the overall complexity of the FEM, making the method more challenging to understand and use.

The FEM might be a more complex method but once mastered, the method can deal more readily with complicated geometric shapes. It is especially this property that is much used in engineering design. There are many applications that rely on the use of the FEM. Some typical fields are: stress and thermal analysis of industrial parts, crash analysis of aircrafts and cars, fluid flow and surgical procedures in medicine.

2.6 Wavefront Propagation

This section describes a methodology for modelling a high frequency approximation of an expanding wave. That is, the wave that is modelled is assumed to have a high frequency, and thus a small wavelength. Instead of describing the full waveform, including amplitudes and frequencies, only the first arrival of the wave is modelled, resulting in a simplified mathematical problem.

A wave front Γ in two dimensions propagates with velocity $F(x)$ at point x. The front moves in a direction normal to itself, meaning the direction is oriented with respect to an inside and an outside [41]. We wish to track the motion of the wave front as it evolves. Let Γ_t denote the position of the front at time t. Since F is assumed to be everywhere

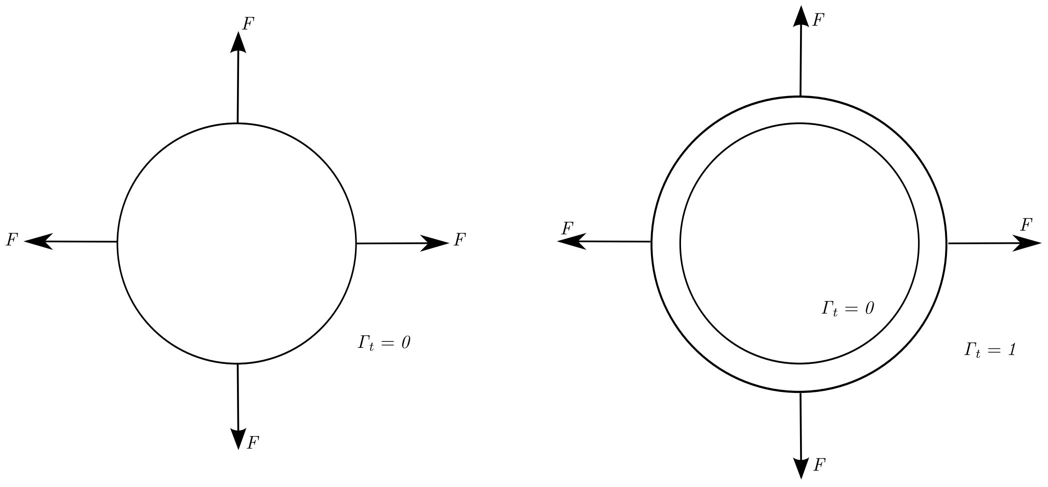


Figure 2.1: An expanding wave front with $F > 0$ in its normal direction.

positive, $F > 0$, the front expands monotonically from Γ and in an outward direction. The front's velocity function F may depend on several factors and can be written as:

$$F = F(L, G, I) \tag{2.5}$$

where:

- L = Local factors (curvature, normal direction, geometric formation)
- G = Global properties of the front (integrals of the front, associated differential equations)
- I = Independent properties (independent of the shape of the front, for instance an underlying fluid velocity that passively transports the front, gravity, magnetic fields.)

We wish to model the position of the front as it expands. The front position can be characterized by its time of arrival, $T(x, y)$, as it reaches point (x, y). Due to the monotonic motion of the front, it reaches each point only once, that is, we search for only the first time of arrival of the front. This can be done by formulating an ordinary differential equation, using the fact that *distance = speed · rate*. We then get equation (2.6),

$$F(x)\frac{dT(x)}{dx} = 1. \tag{2.6}$$

This equation can be extended to higher dimensions [42], resulting in a formulation of the following form,

$$|\nabla T|F = 1, T = 0 \text{ on } \Gamma_0 \tag{2.7}$$

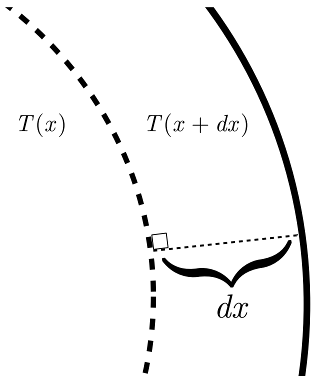


Figure 2.2: A differential equation for the time of arrival is formulated using the fact that *distance = speed · rate*.

where Γ_0 is the initial location of the interface. Thus, the first time of arrival of the wave front is a solution to a boundary value problem. If we let the velocity be a function solely of the location and a scalar unity, then the equation is reduced to what is known as the Eikonal equation.

2.7 The Eikonal Equation

The Eikonal equation is a non-linear PDE, which is a special case of the static Hamilton-Jacobi equations discussed in the previous section. The Eikonal equation is given as

$$\begin{aligned} |\nabla T(x)| &= \tfrac{1}{F(x)}, \quad x \in \Omega \\ T(x) &= g(x), \quad x \in \Gamma, \end{aligned} \tag{2.8}$$

where Ω is a domain in R^n and Γ a set of nodes with initially given arrival times, given by g. $F(x)$ is a scalar function representing the velocity, and $T(x)$ is the unknown first time of arrival of the front.

The equation in its simplest form is when the local velocity function F(x) is set to be constant. When $F = 1$ and $g(x) = 0$, $T(x)$ represents the shortest Euclidean distance from x to the given starting position, Γ.

The Eikonal equation is a versatile equations it has applications in several different fields such as seismology, path planning, multiphase flow, and computational geometry (computer graphics). In seismic imaging the Eikonal equation is considered to be a fundamental equation [31]. The many applications in imaging might not come as a surprise to those who speak Greek, given that the word "Eikon" means image in Greek.

In seismology, a pressure wave is sent through the medium of investigation, and the time of arrival of the reflected waves are analyzed. In seismic forward modeling, a model of the medium is given, and the

Eikonal equation is solved to receive estimates of the first arrival of the pressure wave in the medium.

Compared to other mathematical equations, the Eikonal equation may look simple, but one must not be fooled by its looks. The Eikonal equation is quite challenging to solve due to its non-linearity [22]. Another challenge facing those who wish to solve the Eikonal equation is efficiency. Many numerical schemes and algorithms have been proposed in order to solve this equation as efficiently as possible. Over the next several chapters we will look at some of the most common algorithms developed for solving the Eikonal equation numerically.

2.8 Summary

In this chapter we have looked at how PDEs are used to track the evolution of a system. We have presented two common solution methods for PDEs, FEM and FDM. The former is regarded as a more complex method, but that can readily handle complicated geometries. The latter on the other hand, is regarded as a method with a simpler structure.

Wavefront propagation describes the motion of an expanding wave. The wave front will expand monotonicity when it moves in its normal direction from the initial front, given that the velocity function is larger than zero.

The Eikonal equation is a non-linear PDE, which is a special case of the static Hamilton-Jacobi equation that can be used to describe how waves moves through a medium. Different numerical methods for solving the Eikonal equation exists, but the challenge lies in creating an efficient solver.

Chapter 3

PARALLEL COMPUTING

3.1 The Shift Towards Parallel Computing

For more than two decades, the performance of microprocessors based on a single Central Processing Unit (CPU) has been increasing on average 50 percent per year [34]. Thanks to this unprecedented drive of performance, software developers and users has been able to wait for architectural advances to obtain increased speed for their applications.

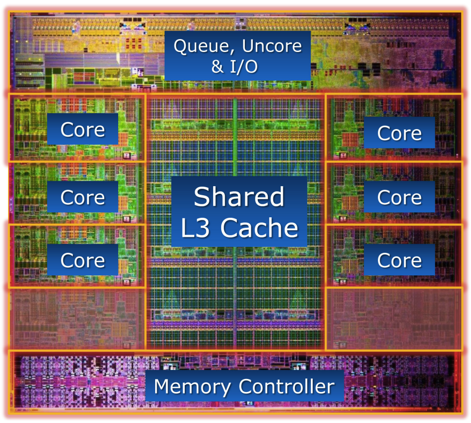
In 2002, however, this drive took a dramatic turn. Since then, the increase in single-CPU performance has slowed to 15-20 percent per year [34]. Due to energy-consumption and heat-dissipation challenges, processor manufacturers realized that they simply could not continue to increase the clock frequency. A dramatic change in processor design was needed, and by 2005, most of the largest manufacturers had decided that this change was in the direction of parallelism.

Today, virtually all of the biggest processor manufacturers has adopted a model of increasing the performance by placing multiple processors, referred to as processor *cores*, in a single package. The processors that employ this design principle are said to have a *multi-core* architecture.

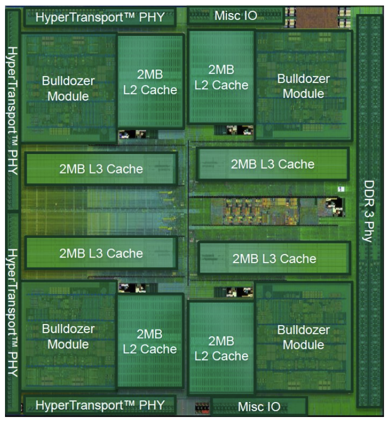
This radical change in processor design has a tremendous impact on software developers. Unfortunately, increasing the number of processor cores will not automatically increase the performance of a serial application, that is, an application that targets one single processor. For applications to take advantage of the increasing number of cores, software developers must specifically change their code to target this type of parallel architecture, in other words, write parallel programs.

Although the parallel computing approach might appear new, both parallel computers and parallel programs is by no means new. As a matter of fact, modern parallel computing has a long history that can be traced back to the late 1950s or early 1960s [33]. For decades, programmers in the high-performance community has been writing parallel programs to exploit the parallelism found in large-scale computers known as supercomputers. Supercomputers has often been built and used for demanding scientific applications such as climate modelling, protein folding, energy research and data analysis.

3.2 Modern CPU Architecture



(a) Intel "Sandy Bridge-E" architecture



(b) AMD "Bulldozer" architecture

Figure 3.1: Two multi-core CPUs from Intel and AMD respectively. (a) has six cores (hex-core), while (b) has eight cores (octo-core). Each Bulldozer-module consist of two cores. The HyperTransport connections can also be seen in (b). Image (a) courtesy of Intel Corporation. Image (b) courtesy of AMD.

Currently, the latest trend among processor manufacturers is to increase the number of processor cores. This trend however, comes at a great cost. The increasing number of cores in today's multi-core processors is done at the expense of poorer memory bandwidth per core. As a remedy, manufacturers have turned their attention to hierarchical organizations of cores and memory, better known as NUMA (Non-Uniform Memory Access).

The history of NUMA goes back to the 1960s when processor speed was starting to outperform memory speed. A new direction was needed in order to get memory speed on par with processor speed. Among many suggestions was NUMA. The fundamental idea behind the technology is to increase memory bandwidth per core by organizing processor cores and memory hierarchically so that each core gets access to its own memory. This hierarchical organization reduces possible contention when multiple cores are trying to read or write to the same memory bank. This is especially important nowadays as manufacturers continue to increase the number of processor cores at the expense of memory bandwidth.

Although a technology from the 1960s, the first physical implementations of NUMA did not take place until the 1990s, when parallel supercomputers were starting to be limited by poor memory performance. NUMA is implemented in hardware using a separate system bus to connect each "isle" of cores to the memory. These small isles are referred to as *NUMA-domains* or *NUMA-nodes*. Each core in a NUMA-node has a uniform memory access cost, meaning that accessing memory near a node is faster than accessing the memory of other NUMA-nodes. This hit ratio is sometimes referred to as the *NUMA-factor* [10]. Depending on the system configuration and processor architecture, at least one or more NUMA-nodes exists within one physical socket.

Two of today's largest x86 processor manufacturers, AMD and Intel, are delivering cache-coherent NUMA (ccNUMA) interconnection with their current generation CPUs. ccNUMA systems use non-shared cache memory to maintain coherency across multiple NUMA-domains. AMD's technology is called HyperTransport, while Intel's technology is called QuickPath Interconnect (QPI). Previously, only high-end server processors incorporated NUMA interconnections, but today, even mainstream systems are incorporating NUMA interconnections.

NUMA's relevance is also present in today's supercomputers. Supercomputers such as Jaguar and Hopper relies extensively on NUMA to deliver increased parallelism. A recent trend among supercomputers is an increasing number of NUMA-domains within a node. Some leading experts [39] believe that this number will only continue to grow as we head towards Exascale computing. The vision of a NUMA-based Exascale supercomputer and the increasing number of NUMA systems makes NUMA a highly relevant topic among performance programmers.

From a programming point of view, NUMA impose new challenges for the programmer. If the full computational power of current and future NUMA-based parallel computers are to be exploited, a more careful distribution of data and threads is required [6]. An issue that some of the most popular parallel programming languages such as Cilk, OpenMP and UPC fail to take into account. Different techniques for making applications NUMA-aware exist [47], in section 6.5, we will examine some of them.

3.3 Graphics Processing Unit

Recent years demand for faster and greater visual experience in games, has spurred the development of devices known as graphics processing units (GPUs). Two of the largest manufacturers of GPUs today are AMD and Nvidia. In terms of architectural design, both companies employ a parallel architecture based on many simple vector cores, designed to be particularly fast at performing floating-point operations (FLOPs). A current exemplar is Nvidia's latest generation of GPU, Kepler, which

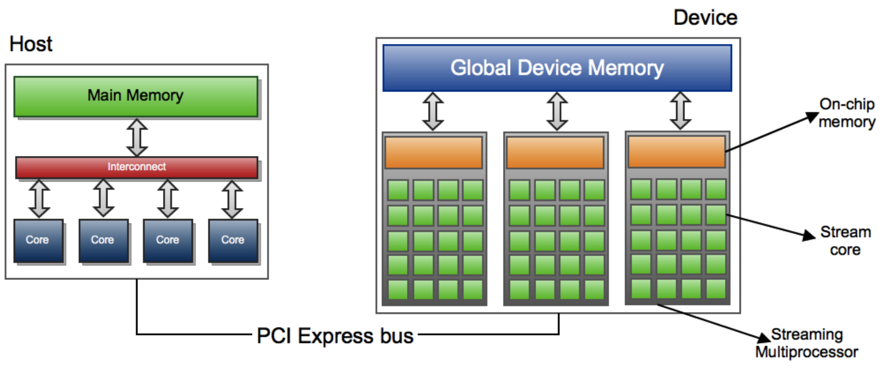


Figure 3.2: A heterogeneous computing model: the GPU device is connected to the host CPU through the PCI Express bus. Data to be computed on the device must be explicitly transferred from the host to the device. Once the computation is finished, data is transferred back to the host from the device.

has more than 1500 stream cores. Not surprisingly, this high number of parallel execution cores has given birth to the term *massively parallel architecture*.

The approach of having many simple cores instead of a few powerful cores is sometimes referred to as a *many-core* approach. The many-core design is a stark contrast to traditional CPU design, where the goal is to maximize the performance of single threaded applications. CPU technologies such as out-of-order execution, complex instruction sets, and hyperthreading, are all reminisces of this.

CPUs and GPUs also differ in terms of how memory is handled. GPUs are designed to prioritize memory bandwidth over latency since latency can be hidden by computation. CPUs however, are designed around large cache coherent memories to increase (single threaded) application performance. Last but not least, GPUs are not *general-purpose* computational units. That is, they must be installed in a system with a general-purpose CPU that can act as a host. In practice this entails that GPUs can not directly access the system's main memory and need to rely on a host CPU to do this. As a consequence, data from the host (CPU) must be explicitly transferred to the device (GPU) memory. This special way of handling memory is often referred to as a *heterogeneous* memory model and must be manually managed by the programmer.

3.3.1 Modern GPU Architecture

Figure 3.3 reveals the architecture behind Nvidia's current GPU architecture, Kepler [13]. The architecture is divided into small blocks of Streaming Multiprocessors (SM), called SMX. SMX is short for Next Generation Streaming Multiprocessors. Each SMX contains multiple stream cores.

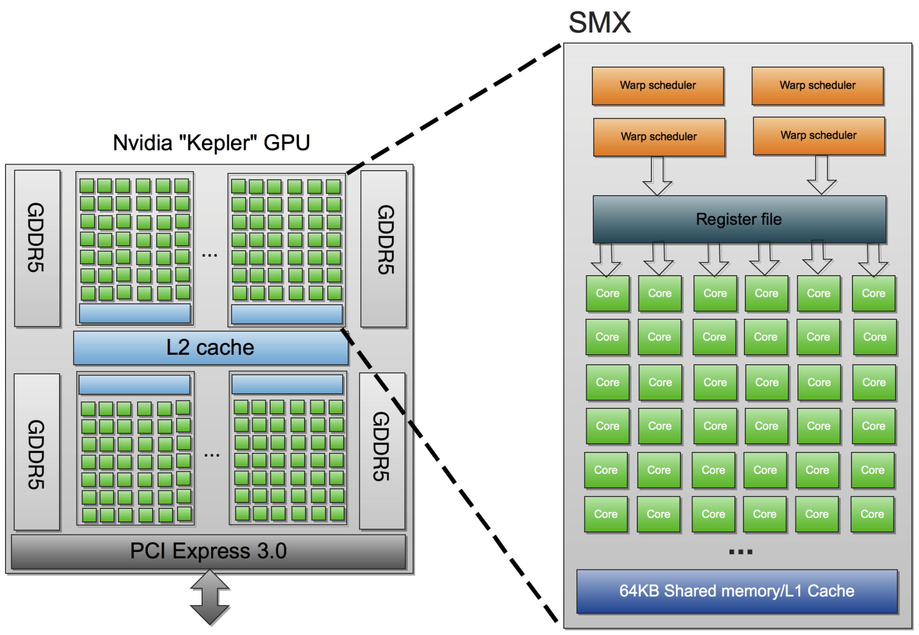


Figure 3.3: A simplified block diagram showing Nvidia's latest GPU architecture called Kepler. There are eight streaming multiprocessors (SMX) on a Kepler die. Each SMX consists of four warp schedulers and 192 stream cores. All cores have access to 64KB L1 Cache/Shared memory.

Stream cores are simple arithmetic logic units that execute the actual computations. Depending on the device configuration, one SMX typically contain many stream cores, and a device contain several blocks of SMXs. The GeForce GTX 680 for example, has 8 SMXs with 192 cores each, bringing the total tally of cores to 1536.

Communication among stream cores inside a SMX is performed using low latency memory called shared memory/L1 cache, while communication across SMXs are done using the slower *global device memory*. Each Geforce GTX 680 device comes with 2 GB of GDDR5 memory.

Kepler's shared memory/L1 cache size is 64 KB and is user configurable. The user can choose to partition this memory as 48 KB of shared memory and 16 KB of L1 cache or vice versa. Certain types of applications experience a performance boost when shared memory is used, that is, a larger shared memory partitioning scheme is beneficial. Other applications may not able to take advantage of shared memory at all. In situations where this is the case, a larger L1 cache might be more beneficial [11]. The idea of having a configurable on-chip memory is to let the user determine what might be best suited for different applications.

In addition to the shared memory/L1 cache, Kepler comes with a 512KB L2 cache that is shared across the device. The idea of the L2 cache is to provide additional low-latency storage to reduce the pressure on the shared memory/L1 cache. Contrary to the shared memory/L1 cache, the L2 cache is not user configurable and is fully managed by the device.

Data from the system memory to the device memory is transferred using the PCI Express bus. Kepler supports the latest PCI Express 3.0 standard which supports data transfer at a speed of 16 GB/s. The on-device communication bandwidth is on the other hand 192.26 GB/s. If we compare the on-device communication bandwidth with the communication bandwidth, we realize that the communication bandwidth is slow and therefore constitutes a bottleneck. Hence, communication between the host and the device should be minimized whenever possible. One way of reducing the impact of the slow communication link between the host and the device, is to increase the computation load on the device in order to hide memory latency [1].

3.4 Nvidia CUDA Framework

Since 2003, GPUs floating-point performance has been outperforming CPUs [1]. High floating point performance is regarded as a vital property in many fields of computer science, especially with respect to numerical applications. Not surprisingly, the idea of using GPUs for general-purpose processing has caught the attention of researchers and other developers in the scientific community. However, one major stumbling block is the difficulty of programming on GPUs. GPUs were originally designed to run games, not to perform general-purpose computations. Users who wanted to use GPUs for general purpose processing had to use graphics APIs such as Microsoft Direct3D or OpenGL to access the device. Very few could master the skills of using graphics API to perform general-purpose processing, but those who were able to could see good performance yields. Their results started to excite other researchers, and eventually lead to the start of a new programming paradigm called *GPGPU* (General Purpose Graphics Processing Unit) [14].

Shortly after the start of the GPGPU era, Nvidia launched their own GPGPU programming API called Compute Unified Device Architecture (CUDA). The main motivation behind the release of CUDA was to make it easier to write parallel applications for GPU structures. Writing CUDA applications requires a CUDA enabled device from Nvidia. These are devices with a special dedicated silicon area that can interpret requests from CUDA applications.

Due to their long history, programming languages such as C/C++ and Fortran have a predominant position among scientific applications. Rather than creating an entire new programming language, Nvidia invented new programming constructs and extensions for C/C++ and Fortran.

A CUDA application consists of both code that runs on the host and code that runs on the device. That is, CUDA applications interchange between running on the host and the device. Code that runs on the device have their own special constructs. CUDA is responsible for recognizing and mapping these directly to the thread or memory hierarchy of the device. Since each stream core is massively threaded, one or more threads are mapped to a stream core.

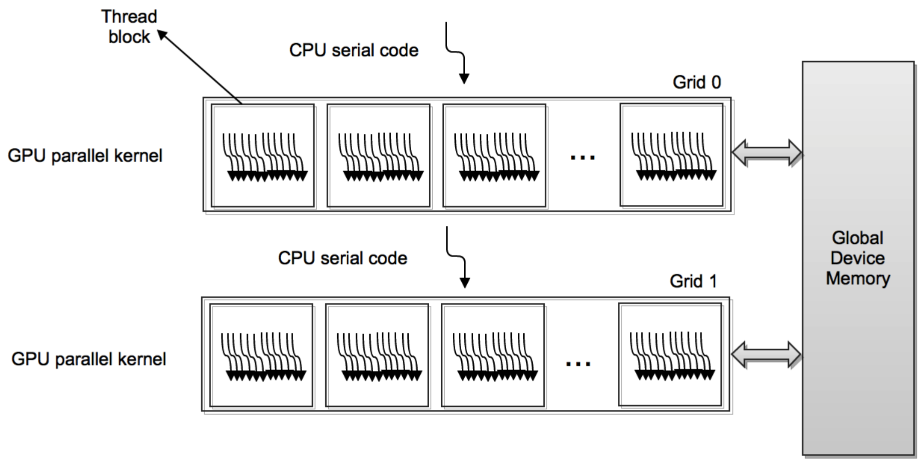


Figure 3.4: The anatomy of a typical CUDA application. The CPU part of the code is executed as normal, but as soon as a kernel is invoked, the execution is performed on the device. Kernel launches are associated with generation of a large number of threads. Threads generated by the kernel are collected in a grid.

A typical workflow (depicted in Figure 3.4) for writing and executing a CUDA application is to write special functions called *kernels*. Prior to a kernel launch (the execution of a kernel function), the kernel's associated data must be transferred to the device. Kernel functions have the CUDA-specific keyword __global__ in front of the function declaration. Once a kernel has been launched, it is executed on the device. During the execution, the device will generate a large number of threads (typically thousands to millions). This collection of threads is called a *grid*. In reality, the kernel is executed as a grid of parallel threads. Threads in the grid are organized in a two-dimensional array called *blocks*. Each block is associated with a two-dimensional coordinate system. The coordinates are determined by the CUDA specific keywords *blockIdx.x* and *blockIdx.y*. Furthermore, each block is organized as a three-dimensional array of threads. The total number of threads that can fit in a thread block is device dependent, but for Kepler, a thread block can hold up to 2048 threads. A three-dimensional coordinate is also associated with a thread block by the three reserved CUDA-keywords: *threadIdx.x*, *threadIdx.y* and *threadIdx.z*. The dimension of the grid and the thread block is passed as execution parameters to the kernel. The programmer determines the execution parameters manually.

Since a massive number of threads is associated with GPU programming, one might ask how threads are scheduled and how large the overhead associated with this task is. In order to reduce the overhead and increase efficiency, most of the tasks related to thread scheduling are implemented on the hardware. Once a thread block is assigned to a SMX, it is sliced into units of 32 threads called *warps*. SMXs executes only a subset of scheduled warps. Although, warps are not part of the official CUDA specifications, knowledge of warps is important. Warp scheduling is used for tolerating long-latency operations such as global memory access and branching. Therefore, knowledge of warp scheduling is especially important when it comes to performance optimization. Often warps are being touted as one of the main reasons why GPUs do not need large cache memories and branch prediction mechanisms.

3.5 Open Computing Language

Open Computing Language (OpenCL) is a jointly developed GPGPU programming model initiated by major industry partners such as AMD, Apple, ARM, IBM, Intel and Nvidia. The project itself is managed by the Khronos Group, the same group that manages the OpenGL API. The main purpose of OpenCL is to create a standardized programming model that targets a broad range of parallel architectures such as multi-core CPUs and many-core GPUs. In many ways OpenCL is quite similar to CUDA. Instead of being a completely new programming language, OpenCL offers new constructs and extension to popular programming languages such as C/C++ and Fortran. Another important feature of OpenCL is portability. Applications written in OpenCL should be able to run on all devices that support OpenCL without any modification.

Although the first version of OpenCL was released in 2009, the programming model is still considered to be immature. According to some, programming in OpenCL is regarded at a lower level than CUDA and less terse [1]. From a performance point of view, several comparison studies [17] has shown that CUDA outperforms OpenCL, but other studies [16] has shown the opposite. The general consensus is that CUDA is faster than OpenCL on Nvidia GPUs. This is not surprising as CUDA is tightly coupled with GPUs from Nvidia and therefore might be able to take better advantage of certain device optimizations.

3.6 Open Multi-Processing

Open Multi-Processing (OpenMP) [5] is a directives-based multi-platform API for writing parallel applications with shared-memory architectures. Like native operating system threads such as POSIX threads (Pthread), OpenMP takes aim at using threads to achieve higher application performance. Although Pthreads and OpenMP share many similarities, their approach on shared-memory programming is somewhat different. The OpenMP programming model offers different tools for automatically

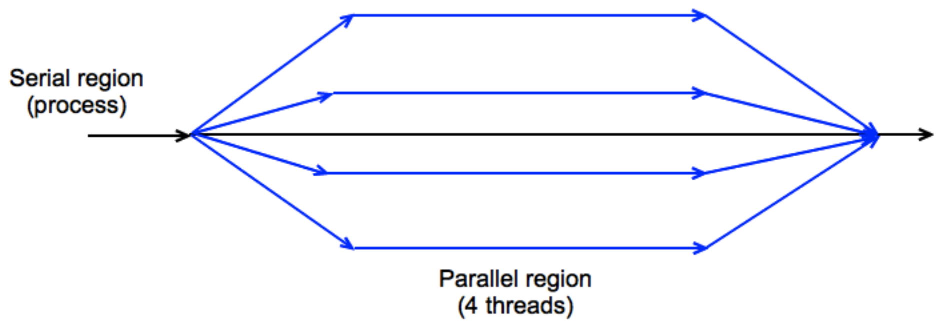


Figure 3.5: The OpenMP fork-join model showing a process forking and joining four threads.

controlling a thread's behaviour, as opposed to Pthreads, where thread behaviour must be controlled programmatically. Because of this, the Pthread programming model is associated with low-level programming. The tools that allows OpenMP programmers to manipulate thread behaviour is called *directives*. These directives are special preprocessor instructions that the compiler understands. Directives are better known as pragmas to C and C++ programmers. If an OpenMP application is compiled using a compiler that lacks OpenMP support, the compiler will simply ignore the pragmas and the application will run serially.

The most commonly used pragma in OpenMP is *parallel for*, which is placed before a for loop, signifying each iteration is independent and can be run in parallel. However, one limitation is that only the outermost loop can be parallelized [50].

This kind of pragma directed parallelization is a reminiscent of a fork-join model of computation. Annotated sections of code are executed by multiple threads and code outside of pragma statements is executed in serial. When execution proceeds through a serial section, only the master thread executes the statement, and the other threads remain asleep. In a parallel section, these threads are reawakened and work is partitioned among the active threads. Figure 3.5 shows the fork-join model in action.

One important motivation for using OpenMP is that it can be easily applied to current serial applications with little effort. Another equally important motivation is that OpenMP can take advantage of today's current and future generation of processors with continuously increasing number of processor cores. However, recent studies has shown that OpenMP does not scale well on a very large number of processor cores, that is, on systems with tens to hundreds of cores [47]. It also does not perform well on code with irregular control flow and memory access patterns [8]. While this may be true for many applications, the problem does not arise in stencil computations, which is the type of computation we are concerned about. In stencil computations, computations consist usually of a series of regularly structured loops with uniform strides when accessing memory.

3.7 An Automated C to CUDA C Translator and Optimizer

Thanks to architectural advancements, GPU performance has been steadily growing every second year [1]. Unfortunately, the same advancements has not been observed with respect to GPUs and their ease of programmability. Although, there has been some effort to make GPU programming easier, the overall consensus is that GPUs are still hard to program. The challenging task of programming a GPU has put the technology out of many researchers reach. The Mint [51] programming model is an effort to make GPU programming easier and thus make the computational power that GPUs offer more accessible.

3.7.1 The Mint Programming Model

Developed at the University of California San Diego, Mint is a directives-based programming model that makes use of programmer annotation to reduce the complexity of programming a GPU. Since both models make use of pragmas, persons familiar with OpenMP's directive style of programming will easily recognize the resemblance between OpenMP and Mint. While OpenMP uses directives to tell the runtime system to execute a structured block of code in parallel, Mint uses pragmas primarily to offload computation to a GPU.

Mint is a source-to-source translator built on the ROSE compiler framework that automatically translates annotated serial C code to CUDA C. If we look closer at Mint's translation workflow (shown in Figure 3.6), we see that Mint constitutes of three main building blocks: a *pragma handler*, a *baseline translator*, and an *optimizer*.

The task of the pragma handler is to parse Mint specific directives and clauses. Currently, Mint comes with five different directives: *parallel*, *for*, *barrier*, *single* and *copy*. Only the parallel, copy, and the for directive will be covered in this thesis. The parallel directive is used to annotate a parallel region, that is, the part of the code we want to run on the GPU. Prior to the parallel region, the copy directive is used to transfer data associated with the parallel region to the device. Likewise, the copy pragma is also used to transfer data back from the device to the host.

Usually, a parallel region is succeeded by the for directive, which

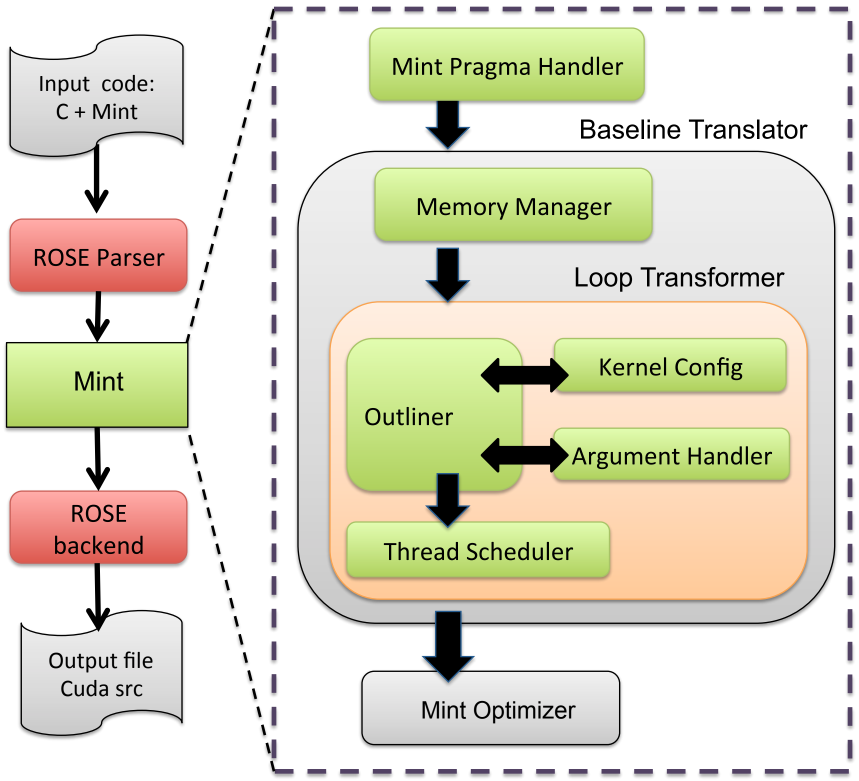


Figure 3.6: A diagram showing the core building blocks of the Mint translator. Pragmas are first handled by the Pragma Handler. Once the pragmas has been verified to be correct, queries from the Baseline Translator is done to translate a region of code to the device. The last step constitutes of optimizing translated code. Figure used with permission from [50]

marks the for loop we want to run on the device. Nested for loops are also supported, although no special optimizations are carried out for nested for loops. For loops are parallelized simply by a strict logical thread to point mapping, that is one logical thread on the device corresponds to one point in the iteration space of the loop. In addition to the for loop directive, Mint also comes with three special clauses: *nest*, *tile* and *chunksize*. Clauses are primarily used to fine-tune a loop.

The nest clause is used to reveal the depth of parallelism of nested for loops. Valid arguments are a constant integer or the *all* keyword. When no nest clause is specified, only the outermost for loop is parallelized. The tile clause is used to divide a for loop into tiles. Valid arguments are three constants separated by a comma. Each tile argument specifies the number of data points that is computed by a thread block. The chunksize clause controls the workload size within a thread block. Valid input parameters are three constants separated by a comma. Chunksize can also be combined with the tile clause in order to determine the number of threads needed to execute a tile.

Once the different pragmas has been parsed, the baseline translator can carry out the task of translating serial C-code to CUDA C. Upon successful translation, the translated code is passed to an optimizer for optimizations.

Mint's optimizer can mainly carry out two types of optimizations: architecture specific optimizations and domain-specific optimizations. The former ensures that a handful set of optimizations is performed for the device that the code will be run on. Only two Nvidia GPU architectures are supported: the GeForce 200-series and Fermi-based GPUs. The latest Kepler architecture is not supported, but due to its high resemblance to its predecessor, Fermi, it is believed that Mint might work efficiently on Kepler.

The domain-specific optimizer targets specifically stencil kernels that appear in structured grid problems. A special stencil analyzer searches for adjacent array references. These references are sorted based on their distance to the center point. This information is then used to perform optimizations with respect to the device's shared memory and registers. Frequently used array references are placed into registers, while neighbouring points are put into the shared memory.

Regardless of whether optimizations has been carried out or not, the final output from Mint is a CUDA source file (.cu) which must be manually compiled using Nvidia's C-compiler, *nvcc*.

A similar programming model to Mint, is the directive-based OpenACC programming model jointly developed by CAPS, Cray, PGI, and Nvidia. The primary influence of the OpenACC model is PGI's accelerator model. The similarities with Mint is striking. For example, the user can, with the aid of pragmas, annotate regions of code to be transferred and executed on the GPU. However, OpenACC is not able to perform domain-specific optimizations for stencil computations.

3.7.2 Performance

Compared to aggressively hand-optimized GPU code, auto-generating code will always involve a performance penalty. The optimizer will not always perform the correct optimizations, or it may be unaware of certain optimizations that is only mastered by an experienced GPU programmer. However, studies [20, 51] prior to this thesis has shown that Mint is able to achieve somewhere between 70 to 83 percent of the performance of numerous hand-optimized stencil kernels. Mint's encouraging performance makes it tempting to use Mint under more generic circumstances, for instance Matrix multiplication. This is however not recommended, as studies [50] shows that Mint's performance under these circumstances is quite limited when compared to more auto-tuned libraries that are distributed with the Nvidia CUDA SDK. Although the performance is not that good, Mint can be used to auto-generate device code that later can be hand-optimized.

3.8 Summary

In this chapter we have looked closer at some of the latest trends in computer architecture. Due to energy-consumption and heat-dissipation challenges, CPU manufacturers could not scale their current single CPU architecture. As a result, CPU manufacturers have turned their attention to multi-core architecture. A multi-core architecture is an architecture with multiple conventional processors on a single chip.

One challenge with today's current multi-core architecture is tightly coupled to memory. The increase of cores is done at the expense of poorer memory bandwidth per core, because when the number of cores increases, contention for memory among the different cores also increases. In order to increase memory bandwidth per core, processor manufacturers have started the incorporation of a technology called NUMA. With the help of NUMA, cores and memory are organized hierarchically so that each core get access to its memory. This has great implication for software developers who must take this in consideration when writing applications.

The change of processor architecture has had a deep impact on serial applications, because serial applications can not exploit the presence of multiple cores. If the performance potential of multiple core is to be exploited, software developers must convert their applications into parallel applications. To facilitate this process, several parallel programming models exists. We have discussed OpenMP, a programming model for shared-memory systems. In the OpenMP programming model, programmer annotations, called pragmas, are used to tell the compiler which part of the code we want to run on multiple cores.

Another emerging parallel computing trajectory are GPUs. These devices consists of many simple cores, rather than few, but more powerful cores. The architecture of GPUs is sometimes referred to as a many-core architecture. The availability of many cores, makes GPUs suitable for high throughput computing. This property, along with high floating-point operations per second performance, has made GPUs popular among users in the scientific computing community.

Several programming models exists for programming GPUs. The two most common models are CUDA and OpenCL. The former is a model tailored specifically for GPUs manufactured by Nvidia, while the latter is an open and a more generic model targeting multiple parallel architectures, ranging from GPUs to multi-core CPUs.

Despite the latest advances in making GPU programming easier, using GPUs for general-purpose computing is still considered to be somewhat a hassle when compared to multi-core CPUs. Inspired by the directive based programming model of OpenMP, Mint was developed to automatically translate serial C-code to parallel CUDA C-code. Being still in its infancy, Mint targets specifically computational stencils on a structured grid. Studies has shown that Mint can achieve up to 80 percent of hand-coded CUDA code for these types of kernels.

Chapter 4

NUMERICAL METHODS FOR THE EIKONAL EQUATION

4.1 Numerical Methods

Numerical methods for simulating propagating fronts are often divided in two categories, namely as front tracking methods or iterative methods. Iterative methods try to compute the solution from a distance perspective. Those methods create distance estimates, which are improved in an iterative process. Tracking methods instead simulate the physical wave as it expands over the grid. Since the wave only passes a node in the grid one time, the need for revisiting nodes is smaller for tracking methods than for iterative methods. Tracking methods are often referred to as one-pass methods. In this section we discuss one tracking method and one iterative method in detail.

There are a few methods that are partly in both categories. Those are iterative methods that update nodes in a partially ordered manner. In section 4.4 we mention a few of these methods.

4.2 Fast Marching Method

The Fast Marching Method (FMM) is a stable numerical scheme for approximating solutions to the Eikonal equation. The method was originally proposed by James A. Sethian in 1996 [40]. Coincidentally, a similar method was proposed by Tsitsiklis [49] in 1995. Since its release, FMM has been extended several times. The algorithm was first extended to anisotropic front propagation by Valdimirsky in 2001 [52] and later by Sethian and Valdimirsky in 2003 [46].

4.2.1 Algorithm Description

The main idea of the algorithm is to mimic the wave front as it expands throughout the grid. For this reason, FMM is classified as a tracking method. To illustrate how the algorithm works, Bornstein et al. [7] use a "prairie fire" as an example. The propagating front is thought of as fire that spreads across the field, which is our domain. The front evolves with a slowness field F towards directions where the fire has not yet spread. At time step $t = 0$, the fire starts at the position Γ. The nodes on the discretized domain are classified as either *Trial*, *Unknown* or *Known*. Trial nodes are the nodes that are burning, the "un-burnt" nodes are Unknown, while nodes that are already burnt are classified as Known. Just as the fire will not return to already burnt nodes, the FMM algorithm does not need to compute solution values of Known nodes, that is, nodes that are already passed by the front.

Consider a wave front Γ in a two dimensional grid that we want to follow, as described earlier in section 2.6. The starting position of the front in the grid is known, and specified in the boundary value formulation. From the given initialised nodes defining the starting position of the front, the algorithm first computes distance estimates to neighbouring grid points, and marks those nodes as Trial nodes. All Trial nodes are stored in a *min-heap* data structure. A min-heap is a binary heap, but where the values are sorted in an ascending order. This allows the smallest value to be popped off the heap first. At any time the heap consists of an approximation of the wave front, or all nodes that are currently burning.

After the update, the algorithm starts marching, by passing one node at the time. The causality principle of the Eikonal problem assures that no point can be affected by grid points with larger values of T. Therefore, the Trial point with the shortest distance (minimal value) is considered correct and fixated by being classified as Known and removed from the heap. When the Trial node with minimal value has been chosen and transformed into a Known value, its adjacent neighbouring points are adjusted (recomputed). If a neighbour is Unknown, it is changed to a Trial node and added to the heap. If a neighbour already is on the heap, its value is recomputed[1]. This readjustment makes revisiting a Known node obsolete, since the value of a Known node will never be changed in later computations. Pseudo code for the FMM algorithm is given in section 4.2.2.

The following discretization of the Eikonal equation is used for updating the adjacent points:

[1]To keep the min-heap structure, an updated Trial node will also affect the position of the node in the heap.

$$\begin{pmatrix} \max(D_{ij}^{-x}T, -D_{ij}^{+x}T, 0)^2 \\ +\max(D_{ij}^{-x}T, -D_{ij}^{+x}T, 0)^2 \end{pmatrix}^{\frac{1}{2}} = \frac{1}{F_{ij}}. \tag{4.1}$$

In equation (4.1), $D_{ij}^{\pm x}$ is a first order upwind finite difference operator of the derivative from below (+) or above (−) with respect to x at point (i, j) in the grid. Only values that are classified as Known are used when updating a node. The smaller solution of the resulting second order system is directly discarded. This is because it corresponds to the distance in the wrong direction. Implementation details on how to solve this scheme is presented in Listing 4.2. A more thorough discussion behind the choice of an upwind scheme can be found in [43]. Stencils of different shapes can also be used. A more in-depth analysis of three stencil shapes for the Eikonal equation in earth modelling is presented in [48].

4.2.2 Implementation details

Implementing FMM is regarded as a complicated process, for this reason, a full implementation is discarded. Instead, we are showing the pseudo code. The initialization step is shown in Algorithm 1.

Algorithm 1 Pseudo code for FMM's initialization step

```
1. Initialize the source points with $T(x) = g(x)$ and assign them with the Known attribute.
2. Initialize the rest of the grid points with $T = \infty$ and assign them with the Unknown attribute.
3. Compute solution estimates on all Unknown nodes that share edges with one or more of the initialized nodes, mark the nodes as Trial nodes, and add them to the heap.
```

After the initialization step, a possible way of implementing the iteration steps is shown in Algorithm 2.

Algorithm 2 Pseudo code for FMM's iteration steps

```
1. If the min-heap is empty, terminate the algorithm.
2. Else: take out the Trial node with minimal value from the min-heap,
   $x$, and mark it as Known.
3. For all nodes, $xn$, sharing an edge with $x$ do:
    • If $xn$ is Unknown, compute a new solution estimate to $T(xn)$,
      mark $xn$ as a Trial node, and add $xn$ to the heap
    • If $xn$ is a Trial node, compute a new solution estimate to $T(xn)$
      and assure the heap is correctly sorted
4. Return to step 1
```

4.2.3 Similarities with Dijkstra's algorithm

Scientists familiar with Dijkstra's algorithm might find some similarities between the Fast Marching Method and Dijkstra's algorithm [44]. The latter is regarded as a *greedy* algorithm. Greedy algorithms are algorithms that "tries to do what appears to be the best solution at each stage" [55].

Just like Dijkstra's algorithm, the FMM algorithm starts by measuring the shortest travel time to adjacent grid points. This results in an optimal ordering of the grid points. No advancement is made until such ordering is in place. The benefit of this type of ordering makes revisiting points with a known value unnecessary.

4.2.4 Performance

Performance always plays a key role when dealing with algorithms. In [45] it is said that updating a simple quadratic equation with a standard iterative method with a total of N points corresponds to a complexity of $O(N)$. The complexity of FMM is on the other hand known to be $O(N \log N)$. The $\log N$ factor comes from the need to update the min-heap every time a node gets a new solution estimate.

The downside of FMM is that the heap must be updated in a serial manner every time each grid point is replaced by a new solution value. The use of a serial heap does not allow for a massively parallel solution. Thus, the algorithm is considered to be a serial one [24].

4.3 Fast Sweeping Method

The Fast Sweeping Method (FSM) is an iterative numerical scheme for approximating solutions for the Eikonal equation. The method was originally proposed by Hongkai Zhao in 2004 [56] but has its roots in a paper [15] published by Per-Erik Danielsson.

Danielsson observed that the characteristic directions of the Eikonal equation falls into one of the four quadrants of a Cartesian grid. Moreover, Danielsson realized that covering the entire grid was enough with only a sequence of four ordered *sweeps* or *scans*. This observation was adopted and later incorporated into FSM by Zhao. Solutions of the Eikonal equation can be thought of as the minimal distance from an object (Γ_0 in equation (2.8)). Often, the minimal distance to an object is a straight line. The FSM use this observation by updating nodes in computing distances in one direction at the time, by updating nodes in a specific order.

4.3.1 Algorithm description

The key idea behind FSM is to update nodes with alternating Gauss-Seidel iterations. On a 2D uniform grid, the Gudonov upwind scheme is given by equation (4.2),

$$\left(\frac{(T_{i,j}-T_{mx})^+}{dx} \right)^2 + \left(\frac{(T_{i,j}-T_{my})^+}{dy} \right)^2 = \frac{1}{F_{i,j}^2} \tag{4.2}$$

where $T_{i,j}$ is the discrete approximation to T in equation (2.8) at $x = (i,j)$, $T_{mx} = \min(T_{i-1,j}, T_{i+1,j})$, $T_{my} = \min(T_{i,j-1}, T_{i,j+1})$ and $(x)^+ = \max(x, 0)$. This discretization is simply a reformulation of the FMM update, equation (4.1).

The algorithm sweeps over the nodes until convergence, which is when no nodes get smaller values. A full sweep of the algorithm, includes four sub-sweeps. In each sub-sweep, each point in the grid is updated in a specific order, right-up, right-down, left-up and left-down. By iterating through the grid in those directions, distances in the same directions are computed in each of the four iterations. Figure 4.1 show the solution after each of the sub-sweeps when computing the distance to a point in the middle of the domain. In this simple example, only one full sweep is needed for the solution to be correct. However, for more complicated problems several full sweeps might be needed. This is the case when objects are present, and the shortest distance is along curved paths. By arguments of causality of an expanding front, the method converges in a finite number of sweeps. This gives rise to a linear complexity, $O(N)$, where N is the total number of nodes.

Not surprisingly, parallel versions of the method has also been proposed [57], utilizing the data independence property of the algorithm.

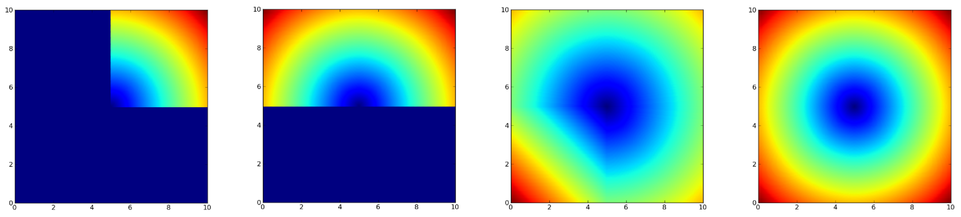


Figure 4.1: From left to right: in a sweeping method, the domain is iterated in specific directions. The figure series shows how the domain is gradually revealed after being swept until convergence (rightmost figure), for the problem of computing the distance field to a point in the middle of the domain.

However, the Gauss-Seidel update makes the algorithm dependent on a single memory location, making the algorithm inefficient or at worst inaccessible for some of the most efficient parallel architectures [24]. Others argue that the advantage of regular memory access is beneficial with respect to the caching mechanisms found in most modern processor architectures today. By altering the algorithm slightly, FSM is easily adjusted for parallel architectures. This work is presented in the following chapter.

4.3.2 Implementation details

Let us look at the actual code for the serial version of FSM. We start by generating a grid with the size of $M \times N$, where the distances between neighbouring nodes is dx and dy in the x and y directions respectively. The grid is then populated by ∞ everywhere. The velocity F is set to 1 on the entire grid in this example. The boundary condition is set to be zero at the top, bottom, left, and right, edges of the grid. With such velocity function and initiation, the solution to the problem is the minimal Euclidean distance to the domain border. In addition, a method is created to check if the neighboring grid points are located within the domain:

```
def InRange(i, j):
    return (0<=i<M and 0<=j<N)
```

Listing 4.1: Python code to check if a grid point is located within the domain.

```
def UpdateGodunovScheme(i, j):
    t_x = INFINITY
    t_y = INFINITY

    if InRange(i+1, j):
        t_x = T[i+1][j]
    if InRange(i-1, j):
        t_x = min(t_x,T[i-1][j])
    if InRange(i, j+1):
        t_y = T[i][j+1]
    if InRange(i, j-1):
        t_y = min(t_y,T[i][j-1])

    if (t_x + dx/F(i,j) < t_y):
        t_new = t_x + dx/F(i,j)
    elif (t_y + dy/F(i,j) < t_x):
        t_new = t_y + dy/F(i,j)
    else:
        t_new = (t_x*dy**2 + t_y * dx**2) / (dx**2 + dy**2) \\
        + dx*dy*sqrt((dx**2+dy**2) \\
        /F(i,j)**2-((t_x-t_y)**2))/(dx**2+dy**2)

    T[i,j] = min(T[i,j], t_new)
```

Listing 4.2: Python code to compute the Gudonov upwind difference scheme.

With the algorithm for the Gudonov upwind scheme in place, we are ready to outline the actual sweeping algorithm. This is shown in Listing 4.3.

```
# Number of sweeps
for sweeps in range(1):
        for i in range(0, M):
            #Right-up scan
            for j in range(0, N):
                UpdateGodunovScheme(i, j)
            #Right-down scan
            for j in range(N-1, -1, -1):
                UpdateGodunovScheme(i, j)

        for i in range(M-1, -1, -1):
            #Left-up scan
            for j in range(0, N):
                UpdateGodunovScheme(i, j)
            #Left-down scan
            for j in range(N-1, -1, -1):
                UpdateGodunovScheme(i, j)
```

Listing 4.3: Python code to perform ordered sweeps.

The final result can be plotted using a surface plot, as shown in Figure 4.2.

4.4 Partially Ordered Iterative Methods

On simple problems, iterative methods are faster than front tracking methods. However, as the domain geometry or the velocity function complexity gets more complicated, the performance of the iterative

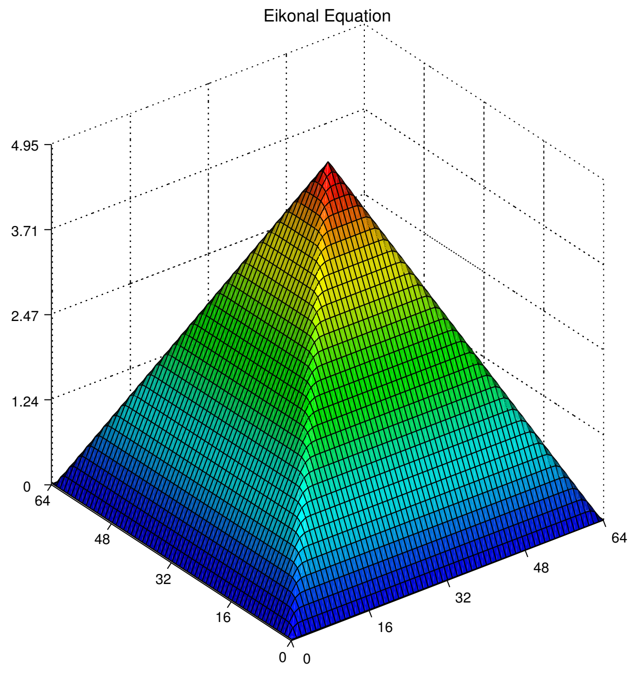


Figure 4.2: The Eikonal equation solved with FSM on a uniform grid with 65×65 nodes. All domain borders were initialized with the value 0, and the velocity is constant over the entire domain.

methods worsens. Instead, tracking methods are considered generally more efficient. Recently, several new methods have been suggested, that mix concepts of iterative and tracking methods. The Fast Iterative Method (FIM) tries to expand the wave everywhere from the initial shape Γ_0. As a result, nodes may be passed several times and the method can be implemented on parallel computers and GPUs [25].

By further enforcing a partial ordering of the updates, the Two Queue method [3] and the Semi-Ordered Fast Iterative Method (SOFI) [18], try to mimic the wave front with heuristic arguments [3, 18]. These two methods are even more stable with respect to geometry and velocity variations. The Two Queue method is applicable only on isotropic front propagations, whereas the SOFI method is able to solve more general anisotropic problems.

Finally, Chacon et al. [9] suggests a two scale approach. First, the domain is divided into sub-domains, of which a coarser grid is created. Using a front tracking method on the sub-domain grid, an order to update the sub-domains is decided. Thereafter, each of the sub-domains is updated using an iterative method.

4.5 Summary

Non-linear static Hamilton-Jacobi equations are often used to describe the arrival time of a propagating front. The numerical algorithms needed to compute solutions for these equations are usually divided into two categories: front tracking methods and sweeping methods. In this chapter we have looked briefly at two numerical algorithms, one from each category.

The FMM is a front tracking method that mimics a front expanding by updating node values in a strictly increasing order. The algorithm is strictly sequential, and considered rather complex when it comes to implementation. For this reason, an implementation was discarded.

The FSM is a sweeping method that computes the distance perspective by iterating over specific directions. Compared to tracking methods, the main advantages of sweeping methods is that they are easier to implement. Sweeping methods are also known to be faster than tracking methods on simple problems. The FSM has an optimal complexity, but the performance is highly dependable on the complexity of the velocity function, and the geometry of the domain. In cases with complex geometries or strongly bending characteristics, many iterations are needed before convergence.

As an exercise to understand the fundamentals of the FSM, the algorithm was implementation in Python. The implementation was used to solve the Eikonal equation. We observed that the iteration order was predetermined, thus, laying foundations for a possible parallelization.

Chapter 5

A PARALLEL FRONT PROPAGATION METHOD

5.1 Geological Fold Simulation

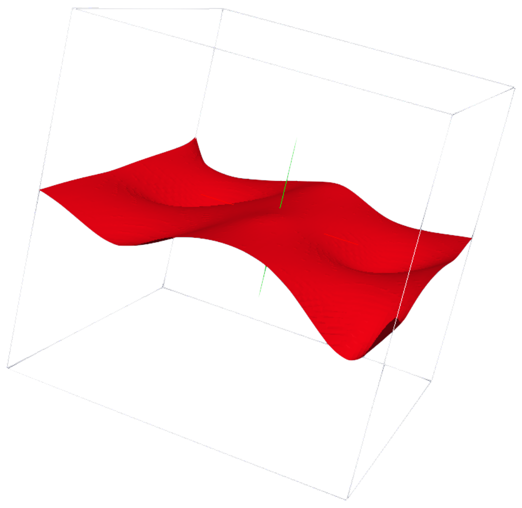


Figure 5.1: The initial geological layer, Γ_0, from which folded volumes of rock was simulated in Figure 5.2.

In this chapter we present a massively parallel front propagation algorithm. A large portion of this chapter has been published in [20] (see Appendix A), and earlier versions of the work in [19].

Kalkulo and Statoil collaborate in developing a new paradigm for highly interactive modelling of complicated geological scenarios and processes. The methodology now developed, is used to describe present-day geology as the realization of a series of geological events and processes along a geological timeline [36]. Many processes rely on the similar surfaces and their corresponding metric properties such as distances, gradients, curvature etc. [21]. Using the mathematical framework for modelling folds developed by Hjelle et al. [21], we observe that the distance maps can be described by the viscosity solution to the static Hamilton-Jacobi equation as shown in equation (5.1).

$$\begin{gathered} F\|\nabla T(\mathbf{x})\| + \psi\,(\mathbf{a}\cdot\nabla T(\mathbf{x})) = 1, \\ \text{given} \;\; T = t_0 \text{ on } \Gamma_0 \end{gathered} \tag{5.1}$$

In equation (5.1), Γ_0 is the initial horizon and **a** the axial direction of the fold. F is a propagation speed that is independent of direction of travel, and ψ is the advection speed in the direction of **a**. The same equation can be used to describe the first arrival of a wave in a media in motion [26].

This equation has earlier been solved for the application of simulating folds in 2D using SOFI [18]. The SOFI method is a mix between a front tracking and iterative method. In this chapter we present an iterative method for simulating folds in 3D.

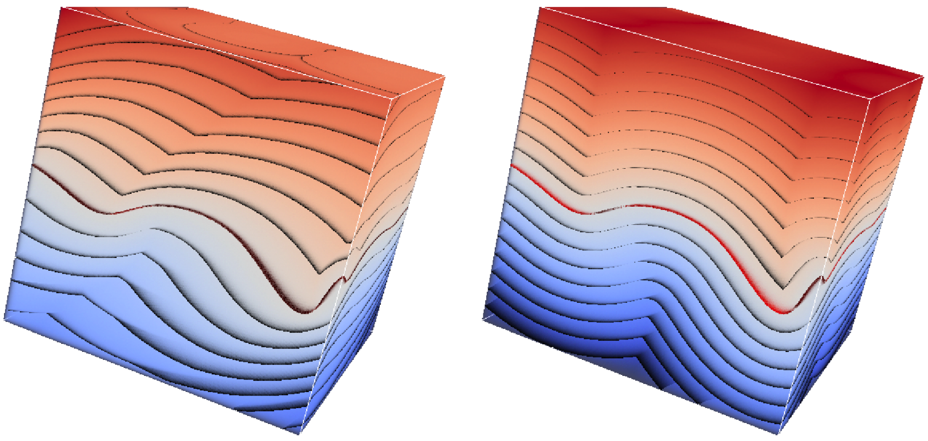

(a) Anisotropic propagation (b) Isotropic propagation

Figure 5.2: (a) Shows a folded 3D volume with $F = 1$, $\psi\mathbf{a} = \frac{1}{2}(-1,1,1)$, simulated from Γ_0. (b) Shows the same initial 3D volume, but when $F = 1$ and $\psi = 0$, resulting in an isotropic front propagation and an Euclidean distance field.

Figure 5.1 shows an example of an initial layer, Γ_0, from which a folded volume of rock-layers can be simulated. Different parameter values, results in different types of geological folded volumes. For example, if $\psi \neq 0$, the front propagation is of the *anisotropic* type. Anisotropic means that the velocity is dependent on the direction. Figure 5.2 (a) illustrates a folded volume simulated using an anisotropic front propagation solver. In the special case where $\psi = 0$, equation (5.1) is reduced to the *isotropic* Eikonal equation as presented in section 2.7. When discussing front propagation, the term isotropic means that the velocity is independent of the direction. When $F = 1$ and $\psi = 0$, the viscosity solution to the Eikonal equation is given as the minimal Euclidean distance from Γ_0.

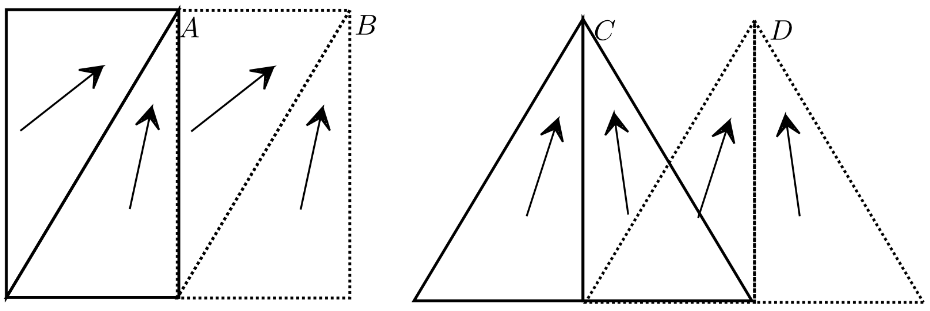
Characteristic curves, or ray-paths, play an important role in the concept of front propagation. Imagine that particles are being transported with the moving wave-front. The characteristic curves are the trajectories of the particles. Another interpretation of the characteristic curves, are as the fastest path to travel to Γ_0.

Folds are often classified by analyzing the lines that connect points with identical angles on the upper and lower boundaries of different layers. For a folded structure, dip isogons are most often straight lines. When folds are simulated with equation (5.1), the dip isogons coincides with the characteristic curves [21] and therefore, the characteristic curves are linear. For isotropic problems with linear characteristics, sweeping algorithms converge quickly. We regard this property as a motivation to investigate related algorithms for the simulation of geologically folded structures.

Front propagation is also used in other applications such as reservoir simulation [4], seismic processing [37, 38] and medical imaging [23, 30], making a fast algorithm highly versatile. For instance, a faster solver would allow more interactive applications. In connection with geological modelling, an interactive application enables users to test more geological scenarios, potentially leading to a better understanding of the inner earth. With respect to medical imaging, a quick solver could potentially lead to faster medical diagnoses. To compute arrival times of reflected seismic waves, a new wave is simulated from the reflecting surface. In all mentioned applications, computational speed is of high importance.

5.2 A Comparison of Different Algorithms

In chapter 4, we discussed two different categories of front propagation algorithms. The first one, FMM, was a strictly serial algorithm, while the second one, FSM, had potential for parallelization. Zhao [57] presents a parallel version of FSM by performing sweeps in different directions simultaneously. Each sweep is performed on different CPUs. Once the first iteration has completed, the minimum value of the different solutions is computed for each grid point. The result of this computation is then used as a basis for the next iteration. These steps are continued until convergence. However, as this study by Li et al. [30] shows, the parallelism of traditional sweeping algorithms is rather limited. Possibly due to the Single Program Multiple Data (SPMD) approach taken by the parallel FSM implementation. Furthermore, as the result from Li et al. shows, good scaling for parallel FSM is first observed when moving on large-scale systems using Message Passing Interface (MPI). In contrary, FIM whose target is Single Instruction Multiple Data (SIMD) architectures, performs better on GPUs.



(a) Traditional Stencil (b) PMM Stencil

Figure 5.3: (a) Traditional stencil shape used in FSM. Notice that the stencils for updating node B depend on the value of node A. (b) Alternative stencil used in PMM. The stencils for updating node C is independent of the value of node D, and therefore the nodes can be computed simultaneously.

Another approach to parallelize sweeping algorithms can be achieved by alternative formulation of the stencil and the iteration order. Weber et. al. [54] has explored the idea of alternative stencil formulation, resulting in a new parallel algorithm called the Parallel Marching Method (PMM). Recall from section 4.3.1 that a bottleneck with the parallel FSM algorithm is that the algorithm becomes dependent on a single memory location. Also, as noted by Weber et. al., another disadvantage with the parallel FSM algorithm is its limited parallelism. Because the number of computation in each step is fluctuating, the real advantage of the parallelization is acquired only on sufficiently long diagonals. These two shortcomings are addressed by rotating the direction of each sweeps 45 ° as shown in Figure 5.3. Changing the stencil leads to a different iteration of the domain, this is shown in Figure 5.4.

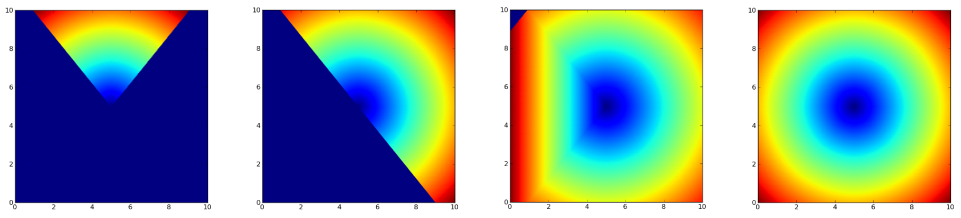


Figure 5.4: The figure series shows how the domain is swept using the PMM stencil, as opposed to FSM's iteration shown in Figure 4.1.

By rotating the direction of all sweeps, nodes on the grid can be updated concurrently. Thus, increasing the level of parallelism offered by the algorithm, as well as allowing coherent memory access. Coherent memory access is an important condition to achieve good scaling on SIMD architectures like GPUs.

Algorithm Name	Independent of Geometry	Parallel Implementation	Anisotropy Support	3D Support
FMM	No	No	No	Yes
FSM	No	No	Yes	Yes
Parallel FSM	No	Yes	Yes	Yes
PMM	No	Yes	Yes	No
FIM	No	Yes	Yes	Yes
3D PMM	No	Yes	Yes	Yes

Table 5.1: A table comparing the different algorithms with each other.

One shortcoming of the PMM algorithm is that it was originally created for computing geodesic distances on surfaces. Therefore, it is only applicable in 2D. Table 5.1 summarises some properties of different front propagation algorithms. To our knowledge, only the PMM and FIM have been implemented on GPUs. Furthermore, only the FIM is applicable in 3D, and to be implemented on a GPU detailed knowledge of CUDA is needed. In the next section, we present a new 3D front propagation algorithm that is well suited for parallel architectures, and easily ported to GPUs.

5.3 3D Parallel Marching Method

Based on PMM, we present our algorithm called 3D PMM. Since our algorithm is based on the idea of the alternative stencil formulation and iteration order, similar to that of the PMM method [54], we have decided to call it 3D PMM. The algorithm has a massively parallel structure, as nodes on an entire surface (planar cut of the 3D volume) can be updated in parallel.

Suppose we have a 3D domain with the nodal values $T_{i,j,k}$ where $(1,1,1) \leq (i,j,k) \leq (n_x,n_y,n_z)$, and with a spacing of (dx,dy,dz). For simplicity, we assume that the values at the nodes closest to Γ_0 are given. Similar to our FSM implementation from section 4.3.1, we set the values of all other nodes to ∞. Because we solve for the minimal distance, that is, the first time of arrival, a smaller value for T is considered as a better approximation. As in most algorithms, monotonic convergence is an essential property for convergence towards the viscosity solution, meaning, if the approximation value is too small, it will not be increased. Hence, we must assure that this never occurs. One way to address this issue is by computing the characteristic curve of the approximation, and assert that the characteristic curve is embedded in the convex hull of the nodes used in the computation. If it is not, the new approximation is discarded. In the appendix of [20] we present details of such a methodology and the conditional upwind discretizations of equation (5.1).

However, a caveat with this discretization is that it introduces a lot of branching, which possibly might lead to reduced computation speed.

The algorithm iterates through the grid in axial directions. Based on the nodal values along the iteration direction, new distance values are computed. For instance, in x-direction, the 3D volume is first iterated in the increasing order of the i index (increasing x value), and then in the decreasing order of the same index. The same process is repeated in the y- and z-directions. We refer to such a full iteration as a *sweep*. Each sweep consists of 6 sub-sweeps in the 3D box. Pseudo-code for the sub-sweeps in the x-direction is shown in Algorithm 3.

Algorithm 3 Algorithm for sub-sweeps in x-direction

```
for $i = 2, \dots, n_x$ do
  for all $j = 1, \dots, n_y$ do
    for all $k = 1, \dots, n_z$ do
      Update $T_{i,j,k}$ using values $T_{i-1,j\pm a,k\pm b}, a \in \{0,1\}, b \in \{0,1\}$
    end for
  end for
end for
for $i = n_x - 1, \dots, 1$ do
  for all $j = 1, \dots, n_y$ do
    for all $k = 1, \dots, n_z$ do
      Update $T_{i,j,k}$ using values $T_{i+1,j\pm a,k\pm b}, a \in \{0,1\}, b \in \{0,1\}$
    end for
  end for
end for
```

The value $T_{i,j,k}$ is computed using nine nodes located in the previously updated plane. Figure 5.5 illustrates the nodes used when the generalized distance is computed in an upwards direction. Each sub-sweep is in the direction of the top of the pyramid.

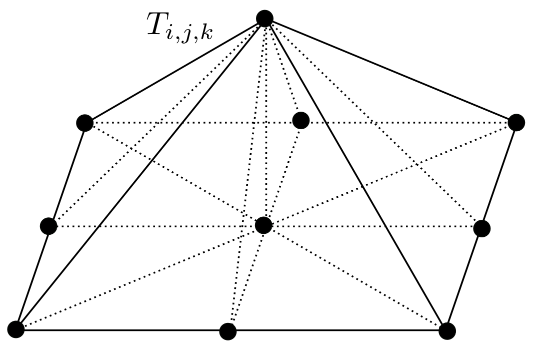


Figure 5.5: The shape of the update stencils. Each sub-sweep is in the direction of the top of the pyramid.

When designing parallel algorithms, it is important that the work (computations) that needs to be done are independent of each other. An algorithm is known to be *embarrassingly parallel* if the work can be divided among processes/threads [34]. Since there are no internal dependencies between nodes on the same plane, it is possible to compute all nodes in the plane simultaneously. This corresponds to computing the two inner loops in each sub-sweep as shown in Algorithm 3. Hence, the 3D PMM algorithm is considered to be an embarrassingly parallel algorithm.

5.4 Multi-core Implementation

Thanks to the embarrassingly parallel structure, the algorithm is easily parallelized for multi-core architectures. We have relied on the OpenMP API to port our algorithm to such architectures. In addition to compiler support, a prerequisite to get access to OpenMP's prototypes and macros is that the OpenMP header file, *omp.h*, must be included. In order to fork of multiple threads, a region must be defined as a parallel region. Usually, this is done to share data among threads to be forked. There are multiple ways of defining a region as parallel, but an often used method is by using the *#pragma omp parallel directive* as shown in Listing 5.1.

```
void Sweep(_DOUBLE_*** T, int nbrSweeps)
{
    #pragma omp parallel
    {
        int di, dj, dk;
        int i, j, k;

        // Nodal values for stencil computations
        _DOUBLE_ tnew, st, xt, yt;
        _DOUBLE_ txy, xnt, ynt, txm, tym, txnyn;

        ...
        // Code for all 6 sub-sweeps
}
}
```

Listing 5.1: Source code showing how a parallel region is defined using OpenMP.

Once a parallel region has been defined, special directives can be used to tell OpenMP to fork new threads. This is typically done by placing the *#pragma omp for* directive right above the for loops to be parallelized. Listing 5.2 illustrates this.

```
    for(i = 2; i < _nx+ 1; ++i)
    {
        #pragma omp for schedule(static, 1)
        for(j = 1; j < _ny+1; ++j) {
            #pragma ivdep
            for(k = 1; k < _nz+1; ++k) {
                ...

}
}
}
```

Listing 5.2: Source code excerpt showing how OpenMP is used to parallelize the two inner for loops of a sub-sweep.

In addition to the basic *parallel for* directive, a special type of schedule clause, *static*, has been used. The clause is also followed by a *chunksize* number, which in our case is 1. By choosing a static schedule type, we tell the system to assign chunks of *chunksize* iteration to each thread in a round-robin fashion. For instance, if we have 12 iterations and three threads, using the schedule(static, 1) clause would lead to the following iteration assignment:

- Thread 0: 0, 3, 6, 9
- Thread 1: 1, 4, 7, 10
- Thread 2: 2, 5, 8, 11

Usually, the static scheduling can be omitted, but we observed a small increase (5-8 percent) in performance by using the clause.

As mentioned earlier, our algorithm is embarrassingly parallel, meaning each computation is independent of each other. Sometimes compilers do not detect this. To assist the compiler with more effective vectorization, the *#pragma ivdep* directive has been used. This directive is independent of OpenMP and instructs the compiler to ignore detected vector dependencies. Most compilers such as *GCC* (GNU Compiler Collection), *ICC* (Intel C/C++ Compiler) and *PGI* (The Portland Group, Inc), all support this pragma.

The process of placing OpenMP directives over the loop inside the Sweep function is repeated for all 6 sub-sweeps. An in-depth analysis of the multi-threaded implementation is presented and discussed in section 6.4.

5.5 GPU Implementation

As mentioned earlier in section 3.4, writing GPU-applications by hand can be cumbersome and requires painstakingly attention to low-level programming details. To avoid manual GPU programming, the Mint programming model was used to annotate regions of code that we wanted to offload to the GPU. Only one function, Sweep, was annotated.

Listing 5.3 shows the Mint pragma used to transfer the required data from the host to the device.

The ghost cell pattern is a common programming pattern used in shared memory systems to parallelize the computation of structured grid problems. The idea with the pattern is to simply divide the grid into smaller chunks among available threads or processors on the system. Each thread/processor is then responsible for computing its chunk. However, one challenge this pattern poses is that updating points at the periphery (defined as the nodes except the center point), requires values from neighbouring chunks. By creating extra regions around the peripheral nodes, referred to as ghost regions, neighbouring values can be accessed faster. As a consequence, more time is spent on computation and less on communication between the different chunks. Mint's stencil analyzer uses this pattern to perform better shared memory optimizations [50]. To aid the stencil analyzer with the optimizations, we have padded the grid size in different axial directions with 2. The padding is in fact special ghost regions that we have created to reduce communication and increase computation. This explains the philosophy behind (_nx+2), (_ny+2) and (_nz+2).

```
void Sweep(_DOUBLE_*** T, int nbrSweeps)
{
    #pragma mint copy(T, toDevice, (_nx+2), (_ny+2), (_nz+2))
    //*** Definitions of constants
    //*** Code for all sub-sweeps
    ...
}
```

Listing 5.3: Source code showing the programmer annotations required by Mint to transfer data from the host to the device.

After the data transfer specific pragmas, annotations similar to that of OpenMP is used to declare a parallel region. This is done by writing *#pragma mint parallel* as shown in Listing 5.4. A parallel region is typically succeeded by annotations for parallelizing for loops. This is done by writing *#pragma mint for pragma* followed by a clause. We have used the **nest(all)** and **tile(16,16,1)** clause. The **nest(all)** clause indicates that all loops are independent an can be run in parallel on the device. The **tile** clause is used to break our 3D grid into 3D tiles of $16 \times 16 \times 1$ as shown in Figure 5.6. A thread block is then assigned to compute the created tile. A more in-depth analysis of the effect of the different tile size and chunk size is discussed in section 6.7.

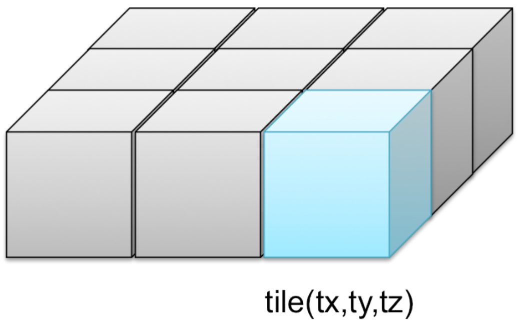


Figure 5.6: Tile decomposition: the tiling clause breaks our 3D grid into smaller tiles. A thread block is then assigned to compute the decomposed tile.

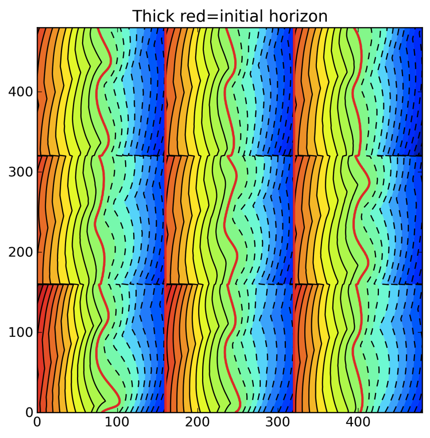
```
void Sweep(_DOUBLE_*** T, int nbrSweeps)
{
    ...
    #pragma mint parallel
    {
        ...
    for(i = 2; i < _nx+ 1; ++i)
    {
        #pragma mint for nest(all) tile(16,16,1)
        for(j = 1; j < _ny+1; ++j) {
                for(k = 1; k < _nz+1; ++k) {
        ...
}
```

Listing 5.4: Source code segment showing Mint specific programmer annotations used to parallelize a sub-sweep in the Sweep function.

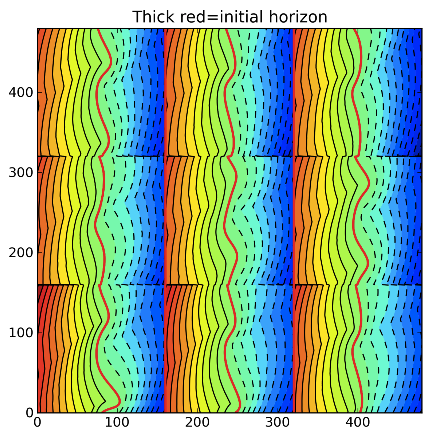
```
void Sweep(_DOUBLE_*** T, int nbrSweeps)
{
    ...

    } // End of all sweeps
 } // End of Mint region
#pragma mint copy(T, fromDevice, (_nx+2), (_ny+2), (_nz+2))
}
```

Listing 5.5: Source code showing the programmer annotations required by Mint to transfer data from the device to the host.

In order to transfer the data from the device to the host, a similar pragma to the one used to transfer the data to the device is used. Listing 5.5 shows the Mint pragma used to transfer data from the device to the host. Usually, this pragma is the last pragma we need to write in a Mint annotated program.



(a)



(b)

Figure 5.7: Figure (a) shows 9 cuts of a solution simulated using OpenMP, while (b) shows the same planar cuts when simulated on a GPU.

When dealing with computational simulations, the correctness of the results plays an important role. The accuracy of our code is verified by using a Python script that reads the output from the C-code and measures the error in three norms, the L^1, L^2 and L^∞-norms. The numerical difference between CPU and GPU solutions is negligible. In addition, the verification script also generates plots that let us determine the impact of the error in a visual manner. These plots are a collection of 3 cuts of the domain in each of the axial directions. Figure 5.7 shows two such plots simulated on different platforms. Compared to the CPU implementation, the accuracy difference on the GPU is minimal.

5.6 Summary

In the first part of this chapter we discussed a method to simulate layers of rocks as the position of a propagating front at different times. Fast simulation of these layers plays a vital role in the exploration software developed for Statoil by Kalkulo. Since fast simulation depends on fast solvers, developing fast algorithms is instrumental for more rapid simulations. Faster solvers enable users to test more geological scenarios, leading to a better understanding of the inner earth. Even though it may appear that such a front propagation is tailored for geological modelling, front propagation is widely used in other fields of science as well. Applications such as reservoir simulation, seismic processing and medical imaging can all benefit from a faster solver.

In the second part of this chapter, we presented a novel algorithm for anisotropic front propagation in three-dimensions. The algorithm has a simple structure and a high degree of parallelism. Due to its simple structure, the algorithm is easily adopted to two of the leading parallel architectures available today, multi-core CPUs and GPUs. The OpenMP API was used to create a multi-threaded implementation for multi-core architectures with little effort. Thanks to the Mint programming model, the algorithm was easily offloaded to a GPU. Developing the GPU implementation was almost as easy as using OpenMP, which proves to be quite time saving.

Chapter 6

RESULTS AND DISCUSSION

6.1 Performance Measurements

Cray XE6 ”Hopper”	
CPU model	AMD Opteron 6172
CPU frequency	2.1 GHz
FLOPs/core	8.4 GFLOP/s
Private L1 per core	64 KB
Private L2 per core	512 KB
NUMA Domains	4
Cores per NUMA Domain	6
Shared L3 per NUMA domain	6 MB
Memory	32/64GB DDR3
Memory frequency	1333 MHz
Memory Channels per NUMA Domain	2

Table 6.1: System overview for one of the nodes on Hopper.

The goal of the experiments was to simulate a folded volume. Figure 5.1 shows the surface the sweeps were performed on. All experiments were conducted using eight sweeps. It is thought that eight sweeps is enough for the solution to converge sufficiently. In these experiments, $\psi a = (-0.35, 0.4, 0.7)$ and $F = 1.1$, while the domain length was set to be 10 in x, y and z directions.

All experiments were run on a three-dimensional grid with a uniform number of grid points. As a baseline, a grid size of 160^3 was chosen. The second grid has 320^3 nodes, while the last size was set to be 400^3. One might ask why the last grid size was chosen to be 400^3 and e.g. not 640^3. The idea with the largest grid size was to find a problem size that was big enough to not fit in the memory of the GPU or the cache of the CPU. This was primarily done to avoid super-linear speedup, but also to be able to measure a reasonable data transfer size. After a careful set of experiments, 400^3 was chosen as it balances execution time while still fulfilling our requirements with respect to performance measurements.

Moreover, all experiments have been conducted using *strong scaling*. In a strong scaling study, the problem size is untouched, while the number of cores is increased after a successful run. This scaling scheme is usually preferable when one is interested in uncovering how the execution time varies with the number of cores. The main reasoning for choosing this scheme was in other words to map the scalability of our application, as the number of cores increases.

Unlike CPUs, GPUs have a greater performance gap between the different precision modes. Therefore, all experiments were conducted using both single and double precision. The main motivation for conducting experiments with both set of precision modes is to unearth any potential performance issues related to precision. For our test device, the peak double precision performance is $\frac{1}{8}$ of the peak single precision performance. However, this difference is only $\frac{1}{2}$ on the more expensive Nvidia Tesla cards.

The time to transfer data to and from the GPU is included in the reported GPU execution times. Moreover, only the computational times for the actual sweeps are measured. A high-resolution timer is placed right before and after the Sweep function is called.

6.2 System Information

GeForce GTX 590	
CUDA cores per GPU	512
Stream Processor frequency	1260 MHz
Single Prec. FLOPs/GPU	1288 GFLOP/s
Double Prec. FLOPs/GPU	161 GFLOP/s
L1/Shared Memory per SM	64 KB
L2 per GPU	768 KB
Memory interface	GDDR5
Memory clock	1728 MHz
Memory	1536 MB per GPU
Memory bandwidth per GPU	163.9 GB/s
Bus Support	PCI Express 2.0 x16

Table 6.2: System overview for the GPU testbed.

For result evaluation, two different machines were used. All OpenMP results were run on one of the nodes on the NERSC Cray XE6 ”Hopper” supercomputer. Each compute node contains two separate chips, each sporting a twelve core AMD ”Magny-Cours” multiprocessor. Within each

AMD "Magny-Cours" chip, there are two hex-core multiprocessors. These hex-cores are connected via two cache coherent HyperTransport 3 (cHT3) ports, one at 16x and another at 8x. Each hex-core also has an additional 16x cHT3 port that can be used to connect to an adjoining chip. There is also a non-cache coherent port, but this is not utilized in Hopper for connecting multiple CMPs together. Full system specifications for Hopper is presented in Table 6.1.

Several actions was taken to ensure that the results were as accurate as possible. To avoid being interrupted by other system processes, the experiments on Hopper were run in a non-interactively fashion. Special PBS scripts was therefore written and later submitted to the Torque batch system. In order to maximize application performance, all jobs were submitted to the regular queue and not the debug queue. A sample PBS script is shown in Listing 6.1.

```
#!/bin/bash
#PBS -q regular
#PBS -l mppwidth=24
#PBS -l walltime=01:30:00
#PBS -N 3dpmm_cube
#PBS -e 3dpmm_cube.$PBS_JOBID.err
#PBS -o 3dpmm_cube.$PBS_JOBID.out
#PBS -V

cd $PBS_O_WORKDIR/pgi

for app in ./3dpmm-160d; do
 echo "*** APP is $app ***"
  for i in 0;  do
    export OMP_NUM_THREADS=1
    export MP_BIND=yes
    export MP_BLIST=5
    $app >> n160d_pgi_cube_1
   done
 done
```

Listing 6.1: A sample PBS script submitted to Torque queue system on Hopper.

GPU experiments were conducted on a system with two Nvidia GeForce GTX 590 cards. Although these cards are dual-GPU cards, meaning each physical card holds two GPUs, only one device was utilized. The cards were connected to the host using the PCI Express 2.0 bus. Full GPU specifications is shown in Table 6.2. The core host specifications for the GPU testbed consists of a quad-core Intel Xeon E5620 "Westmere-EP" CPU running at 2.4 GHz.

6.3 Compiler and Package Information

In our set of experiments with OpenMP, we used *gcc* version 4.3.4, while experiments on the GPU were conducted using *nvcc* 4.0. For our study of different thread to core mappings, we used the PGI compiler version 11.9.0. We determined that the performance of the libGOMP affinity specifier and numactl was quite poor on Hopper. Using an Intel compiler was also not an alternative because Intel compilers do not support thread affinity specification on AMD CPUs. For compiler optimizations, we used the *-O3* flag for all set of compilers. Finally, Mint version 1.0.1 was used, built on the Rose compiler framework version 0.9.5a.

In addition to the compilers, several profiling tools were also used. For CPU profiling, valgrind v3.7.0 with the cachegrind option was utilized, while Nvidia's default CUDA profiling tool was used for GPU profiling.

6.4 CPU Performance Results

N	t_1	t_2	t_4	t_8	t_{12}	t_{16}	t_{24}
160^3	194.17	100.66	52.62	27.20	19.12	14.07	10.22
320^3	1795.86	822.48	423.87	219.39	145.80	112.57	81.40
400^3	3142.23	1628.12	853.50	430.59	286.83	223.55	177.86

Table 6.3: Computational times for three grids with a total of N nodes using single precision. t_i is the CPU time for i cores.

Table 6.3 displays the computational times for experiments using OpenMP with single precision. Independent of the grid size, the speedup is near-linear (1.9x) up to 12 cores. However, the speedup drops to 1.3x when the number of cores is increased beyond 12 cores. We believe this performance decrease is not necessarily connected to the algorithm itself, but rather to the implementation that fails to take the NUMA architecture on Hopper in account. Issues related to NUMA is discussed in section 6.5.

A visual representation of the speedup factor can be seen in Figure 6.1. From the figure, we can see a near-linear speedup up to 12 cores. After that, the scaling flattens out. As mentioned earlier, we believe that data locality plays a big role when we scale past 12 cores.

OpenMP results using double precision are displayed in Table 6.4. As we can see, the performance difference between single and double precision on the CPU is quite small (10-12 percent). More interestingly, the speedup factor trend is the same as for single precision. A near-linear scaling up to 12 cores, but as soon as data is transferred across different sockets, the performance gets a hit and the scaling drops to 1.2-1.3x.

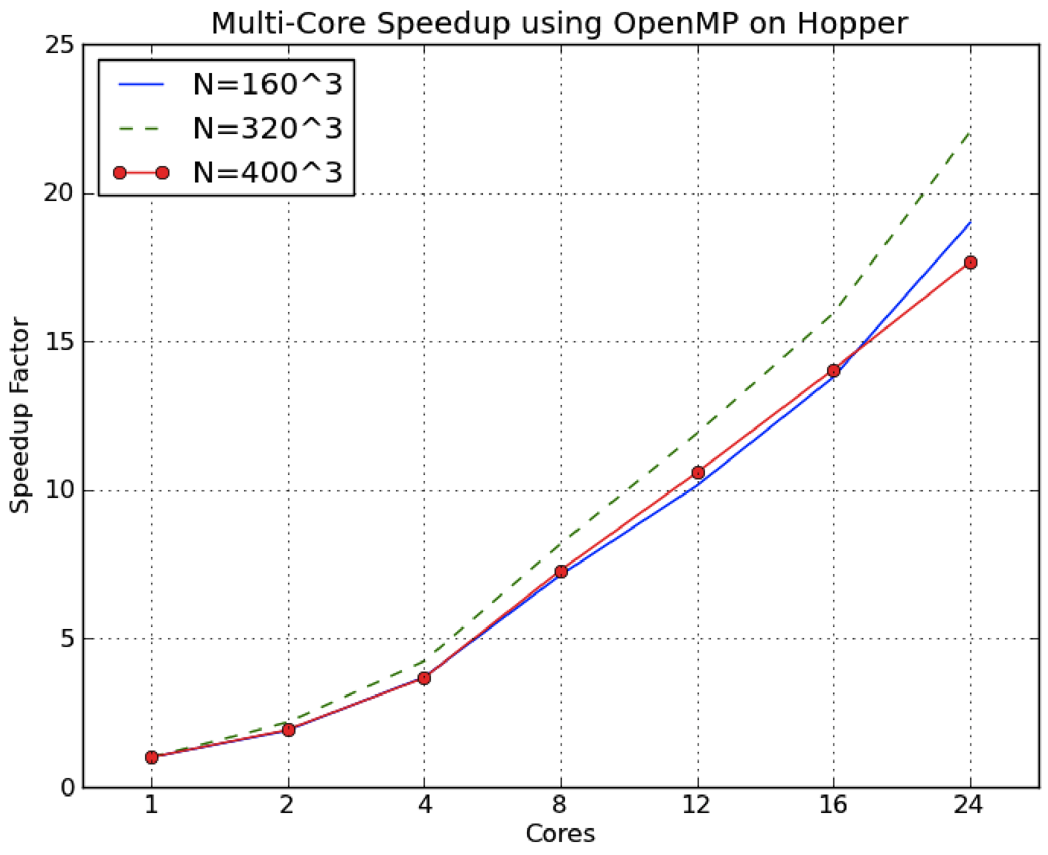


Figure 6.1: The speedup factor for the different grid sizes. Results from Table 6.3 is used as basis to create the plot.

N	t_1	t_2	t_4	t_8	t_{12}	t_{16}	t_{24}
160^3	217.54	112.85	58.93	30.44	21.64	15.73	11.40
320^3	2016.50	911.12	472.84	245.08	170.76	125.08	90.07
400^3	3886.31	1799.22	928.89	481.99	339.72	246.80	240.05

Table 6.4: Computational times for three grids with a total of N nodes using double precision. t_i is the CPU time for i cores.

6.5 NUMA optimizations

Recall that Hopper consists of two AMD Magny-Cours CPUs with 12 core each connected in a multi-socket configuration. Figure 6.2 shows the different possible NUMA topologies available on Hopper: diagonal (6.4 GB/s), vertical (19.2 GB/s) and horizontal (12 GB/s). A fourth configuration is possible by spreading the core across each domain e.g. core 5, core 16, core 7 and core 18. This configurations is sometimes referred to as a *cube* or *spread*.

Choosing the optimal topology is strongly linked to configurations that maximize the bi-directional memory bandwidth. The worst configuration

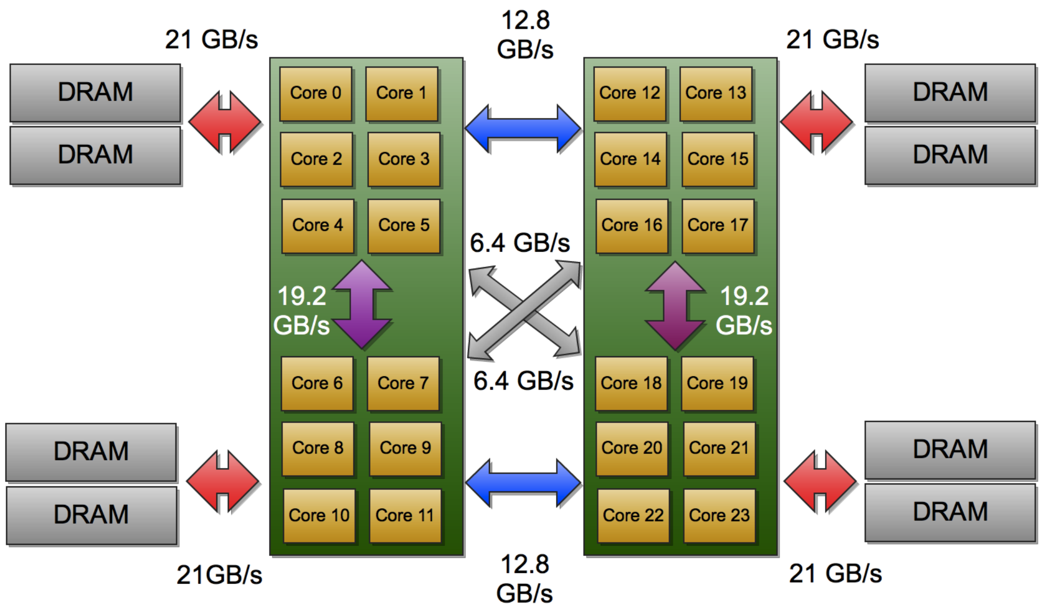


Figure 6.2: The different NUMA topologies available on Hopper. The difference between the horizontal and the vertical links is accounted for by the extra 8x cHT3 port between hex-cores co-located on the same package.

is therefore the diagonal configuration where the memory bandwidth is only 6.4 GB/s. The best configuration is regarded to be a vertical configuration where the memory bandwidth is 19.2 GB/s. If we are not careful, we might experience that a diagonal topology is chosen and as a consequence, our application is limited by poor memory bandwidth.

The OpenMP results from Table 6.3 and Table 6.4 showed that the near-linear scaling stopped at around 12 cores. After that, we suspected that the performance was hurt due to the implications of having non-uniform memory access. In such configuration, an non-optimal thread to core mapping will hurt the performance. Additionally, a volatile thread configuration, might also cause unexpected effects on the performance. Threads migrations across NUMA-domains are extremely costly, and threads that access proximal data should be clustered within the same NUMA-node. Performing NUMA optimizations is in other words about preserving data locality. There are several ways of improving the data locality: through implementation and through environmental configurations.

The OpenMP programming model gives indirect access to improve data locality. Usually, this is done by performing a parallel data initialization, better known as the *first-touch policy* [47]. Performing such initialization on Hopper, ensures that each thread accesses memory pages on the same die as the computation part of the code is performed. In our implementation, this is done in the ImplicitInitialiser function. An excerpt from this function is shown in Listing 6.2.

```
void ImplicitInitialiser(_DOUBLE_*** T, ...)
{
    #pragma omp parallel
    {
        #pragma omp for schedule(static,1)
        for(i=eb;i<nx-1+eb;i++) {
            for(j=eb;j<ny-1+eb;j++) {
                for(k=eb;k<nz-1+eb;k++) {
```

Listing 6.2: A code excerpt demonstrating the concept of first-touch policy in the ImplicitInitialiser function.

Parallel data initialization is only one way of reducing NUMA-penalties. Another alternative is manually specifying the thread to core mapping, better known as the *thread affinity*. Specifying the thread affinity is considered challenging as it requires the programmer to acquire low-level details about the system that the code will be run on. Since thread-affinity is tightly coupled to machine configuration and the compiler used, it also reduces the code portability.

We have relied on the compiler to bind threads and specify the thread affinity. For the PGI compiler, specifying the environmental variable MP_BIND=yes, enables thread binding, while thread affinity can be specified through the MP_BLIST environmental variable.

In addition to extending our code with parallel initialization, experiments have also been carried out to map the application performance under the different NUMA topologies. Conducting experiments with different topologies often leads to generation of many performance results. For simplicity reasons, only the results from the smallest grid size using single precision is presented in Table 6.5.

Top.	N	t_1	t_2	t_4	t_8	t_{12}	t_{16}	t_{24}
Cube	160^3	189.59	97.71	50.22	25.62	17.72	13.01	8.95
Vert.	160^3	190.02	97.83	50.36	25.72	17.81	13.76	9.66
Horiz.	160^3	189.90	97.67	50.30	25.68	17.82	13.08	9.10
Diag.	160^3	378.61	205.88	111.13	57.25	40.99	33.79	105.74

Table 6.5: Computational times for the different topologies.

Looking at the results from Table 6.5, it is quite clear that using a diagonal topology has a dramatic effect on the performance, especially when using 24 cores. Due to the highly ordered affinity, the communication is restricted to the diagonal link where the memory bandwidth is at its lowest. Comparing the performance from the other topologies reveal that the application might not be limited by memory, as there are very small difference between horizontal, vertical and cube. Figure 6.3 mirrors this observation as well.

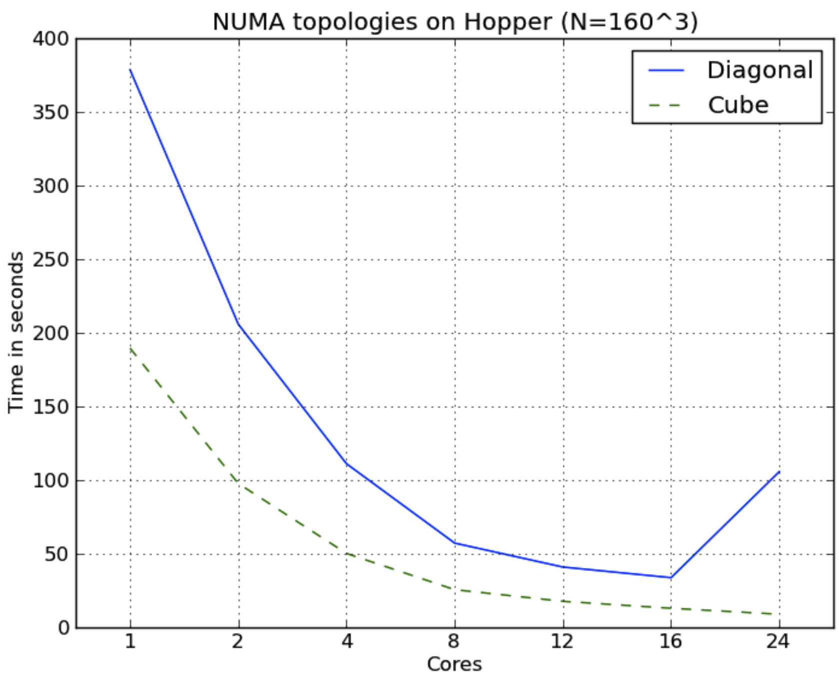


Figure 6.3: Performance scaling using diagonal and cube topology.

Although the different optimizations has increased the performance, the overall speedup has not had increased as much as we had hoped. This leads us to believe that the poor scaling has its root in the large stride of the memory accesses. On a 160^3 grid, this amounts to accessing distant locations in memory. With this grid size, a stride length is defined to be 160 x 160 x 4 bytes, meaning much of the benefit of having a series of hierarchical caches is lost.

6.6 GPU Performance Results

N	Serial	$OpenMP_{24}$	Mint	$Speedup_1$	$Speedup_{24}$
160^3	194.17	10.22	2.43	79.9x	4.2x
320^3	1795.86	81.40	14.84	121.0x	5.4x
400^3	3142.23	177.86	25.28	140.1x	7.0x

Table 6.6: Comparing the CPU execution times with GPU execution times using single precision.

Table 6.6 shows a modified version of Table 6.3, but where GPU results are included. For the baseline grid size, the difference in speedup between a single core and the GPU is approximately 80x. When comparing with 24 cores, the speedup is 4.2x. The difference in speedup continues to increase with the size of the domain. Recall from section 3.3 that latency (data transfer) can be hidden by increasing the computation on GPUs. When the grid size increases, the computational complexity also increases, and hence the cost connected with transferring data from the host to the device is amortized.

N	Serial	$OpenMP_{24}$	Mint	$Speedup_1$	$Speedup_{24}$
160^3	217.54	11.40	4.77	89.5x	2.4x
320^3	2016.50	90.07	31.09	64.8x	2.9x
400^3	3886.31	240.05	57.42	67.6x	4.1x

Table 6.7: Comparing the CPU execution times with GPU execution times using double precision.

The double precision results for the GPU is presented in Table 6.7. If we compare these results with the results from Table 6.6, it is clearly visible that the difference between double and single precision on the GPU is indeed poor. While the difference between the different precision modes are around 10-12 percent on the CPU, the difference is approximately 40-43 percent on the GPU. The poor double precision performance also explains the sudden drop in speedup factor for the GPU results (140x vs 67.6x). It is quite possible that the double precision results could have been better if the experiments were run on a Nvidia Tesla card, where the double precision performance is more close to the single precision performance.

6.7 Mint Performance Analysis

Analyzing the performance of Mint is possible, even though we do not have a hand-coded version. For the analysis, we have used the Nvidia Occupancy Calculator. A greater occupancy results usually in better memory saturation and thus, higher performance. This is not universally true as this study [53] shows. In some cases where optimization with respect to shared memory is used, a lower occupancy might reduce the shared memory traffic, leading to higher performance.

Parallel Thread Execution (PTX) is a virtual machine and instruction set architecture [12] provided by Nvidia with the CUDA software development environment. When a CUDA application is compiled, the code is in reality translated to PTX. According to [12], a "PTX-to-GPU translator and driver enable NVIDIA GPUs to be used as programmable parallel computers". The PTX functionality is built into the default CUDA compiler, nvcc, and as a consequence, nvcc can generate PTX instructions upon compilation. By using the nvcc compiler setting, –*ptxas-options=-v*, we were able to read the reported number of registers used. For our implementation this number was reported to be 61. When we inserted this number into the occupancy calculator, along with our

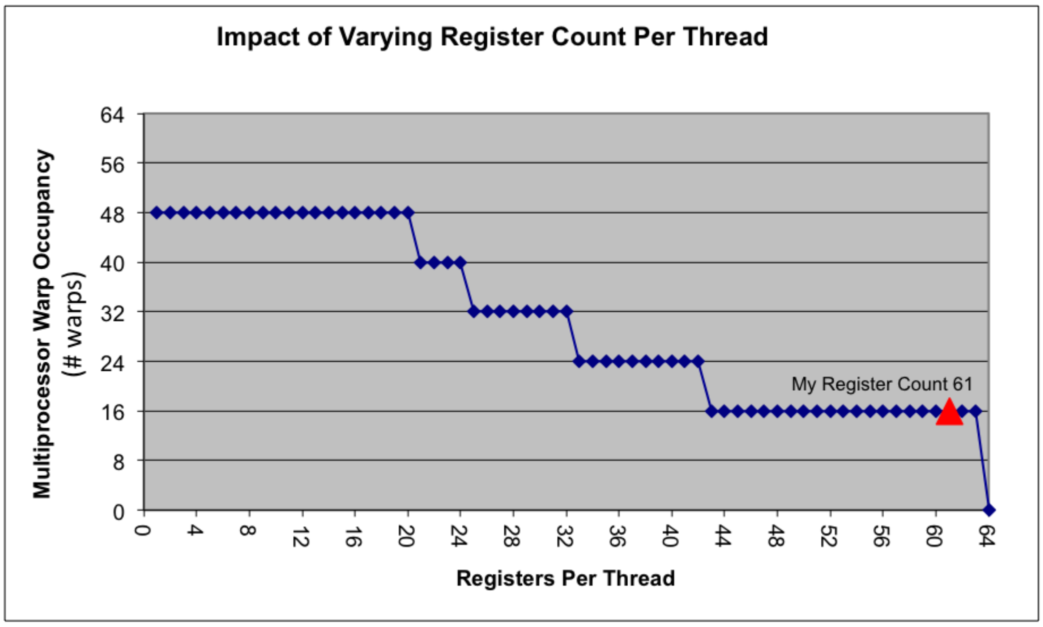


Figure 6.4: Mint's register usage is high and is holding back the performance. Ideally the red triangle should align on the blue horizontal line on the left hand side.

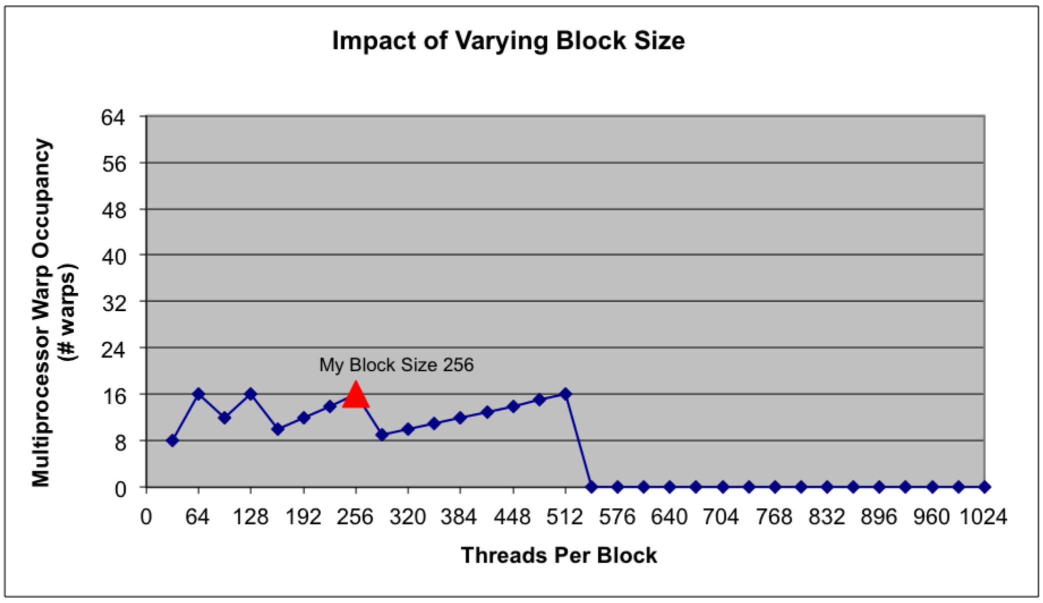


Figure 6.5: In our Mint configuration we are using a thread block size of 256. One limitation in Mint is that the block size can not be larger than 512. This is a limiting factor for newer GPUs.

device specifications, we discovered that the register usage was rather high. Figure 6.4 illustrates Mint's register usage. The high register usage is limiting the performance of the Streaming Multiprocessors. Our device has 48 warps per Streaming Multiprocessor, but as Figure 6.5 shows, due to the high register usage, only 16 is currently in use.

It is possible to limit the register usage by using the *maxregcount* compiler flag. However, when this option was used, the Mint translated code would not run. Mint also comes with an optimization flag, *useSameIndex*, that can reduce the register usage. Turning this option on, caused the application to not run.

Block size	Tile size	N	Mint
64	16x4x1	240^3	5.96
128	16x8x1	240^3	6.19
256	16x16x1	240^3	6.20
512	16x32x1	240^3	7.06

Table 6.8: Computational times for the different tile size using single precision and aligned memory access (x-direction).

Block size	Tile size	N	Mint
64	1x16x4	240^3	16.59
128	1x16x8	240^3	29.48
256	1x16x16	240^3	58.65
512	1x16x32	240^3	118.29

Table 6.9: Execution times for different tile values using single precision. The multiplier is shifted one place to the right (y-direction).

Block size	Tile size	N	Mint
64	4x1x16	240^3	48.26
128	8x1x16	240^3	53.75
256	16x1x16	240^3	60.07
512	32x1x16	240^3	77.57

Table 6.10: Execution times for different tile values using single precision. The multiplier is shifted to the outmost right position (z-direction), resulting in unaligned memory access.

In section 3.7.1 we discussed the different clauses that Mint is bundled with. According to [50], a tile size with a multiple of 16 in x-direction is recommended, as this will ensure an aligned memory access. By carefully shifting direction of the multiplier to the right, that is, moving it from x-direction to y-direction, then from y-direction to z-direction, we can see how the performance varies with respect to both block size, tile size and memory alignment. The results for the unaligned memory accesses are presented in Table 6.10 and Table 6.9. As we can see from the tables, the performance worsens as the multiplier is shifted from y-direction to

z-direction. As expected, best performance is observed when the memory access is completely aligned (Table 6.8). The performance gap between x-direction and z-direction is quite noticeable. As a side note, we can observe that largest performance variation occurs when the tile size increases in z-direction.

Chunksize is another clause that can be modified by the user. The idea of the chunk size is to choose the number of threads needed to execute a tile. In [50], no real recommendation is given, instead the programmer is encouraged to ”experiment with different configurations”. We experimented with different values for the chunksize clause. However, no real performance improvement could be observed. Instead, a small decrease in performance could be tracked for the larger grids. This might be due to the fact that there is a small overhead associated with chunking as reported in [50].

Currently, there is a high number of branching in the update step. As discussed in section 3.4, branching is expensive on a GPU and leads to reduced computational speed. Experiments were carried out with reformulated code to reduce branching. To our surprise, reducing the number of branching increases the execution times for our code. Further investigation is needed to find the most efficient balance. For a more detailed comparison, full result comparison from the version of the code with reduced branching is disclosed in Appendix B.

6.8 Discussion

A performance improvement of 2.5x to 7x when using a GPU may appear to be modest, especially nowadays when certain studies claiming GPUs to deliver speedups between 10x to 1000x. In the work by Lee et al. [29], 14 of the most used computing kernels is benchmarked using a CPU and a GPU, with the purpose of uncovering performance differences between CPUs and GPUs. According to Lee et al., in cases where substantial GPU speedups is reported, it is possibly because the GPU performance is being compared with a serial CPU implementation or running on a single CPU core. However, when the GPU performance is compared with a parallel CPU implementation, the difference in performance is vastly less. Lee et al. concludes that real performance difference between CPUs and GPUs are in reality much closer, claiming that GPUs are on average 2.5x faster in their study.

An astute reader might argue that Lee and the other fellow authors of the article are all employed by Intel and that their work was indeed sponsored by Intel. Erroneous performance reporting in the high-performance computing community is nothing new. Over the years, there have been several studies claiming that erroneous reporting is hurting the credibility of the high-performance community [2, 29]. More recently, a new article was published supporting the work by Lee et al. In [35], the ten most dubious performance reporting techniques are shown. Both in connection with this thesis and [20], a large effort has gone into in making fair performance comparison. For example, both double and single precision performance has been tested, a large enough grid sizes has been chosen so it does not fit directly in the GPU memory etc. As far as we can tell, our performance measurements complies with the ten points enlisted in [35].

Core Architecture	**AMD Magny-Cours**	**Nvidia Fermi**	**Nvidia Kepler**
SP GFLOP/s	100.8	1228	3090
Maximum Load Power (W)	115 W	365 W	195 W
Minimum System Power (W)	750 W	700 W	550 W
GFLOPs/Watt (CPU/GPU-only)	0.87	3.36	15.84
GFLOPs/Watt (System)	0.87	1.75	5.61
Price (USD)	$1039	$449	$499

Table 6.11: A table showing the single precision FLOPS per watt for the CPU and the GPU used in our evaluation platforms. To show future trend, specifications for Nvidia's latest architecture, Kepler is also included.

When comparing the performance of GPUs with CPUs, another interesting area of comparison is FLOPS per watt, referred to as performance per watt. Krueger et al. [27], has compared the energy efficiency of three different parallel architectures: multi-core CPUs, GPUs and a custom many-core FPGA architecture called "Green Wave". The study was performed in connection with seismic modelling, where PDEs using stencils with an order of eight or twelve is computed. Although the Green Wave was the better performer, Krueger et al. observed that the Fermi-based Tesla C2050 GPU had a higher performance per watt ratio than the Nehalem-based Intel Xeon Xeon E5530 CPU. This observation is applicable to our evaluation platform as well. Table 6.11 shows the single precision GFLOPS/W rates for the CPU and the GPU used in our experiments. The GPU from our evaluation platform can deliver approximately 3.36 GFLOPs per watt, while the CPU can only deliver 0.87 GFLOPs per watt. However, because GPUs are not general-purpose, they need a CPU to operate, meaning that power consumption of the host should be included in the final calculations. Even if we include the hosts' power requirements in our calculations, GPUs still deliver more performance per watt. Looking ahead, Nvidia's latest GPU architecture, Kepler, can deliver approximately 15.84 GFLOPs per watt, maintaining GPUs lead in this area.

Another interesting area of comparison is price. At the moment of writing, each CPU used in Hopper costs around $1000 USD, while the GeForce GTX 590 card we have used costs around $450 USD. If we take the performance in account, our GPU results deliver 2.5x to 7x the performance, while costing approximately 4.5x more than the 24-core CPU node.

6.9 Summary

In this chapter we have presented the results of the 3D PMM algorithm running on two multi-core CPUs using OpenMP, and on a GPU using Mint. The results shows that the algorithm scales well on both architectures. The near-linear scaling on the multi-core CPU is hampered by the underlying NUMA architecture of our CPU testbed. Several NUMA-specific optimizations such as incorporation of a first-touch policy, thread binding and specifying a thread affinity was performed to reduce possible NUMA-effects. The optimizations showed a mild performance increase. Further investigation is needed to better understand the possible NUMA-effect on the performance.

Depending on grid size and precision, the overall GPU performance is approximately 2.5x-7x faster than the multi-threaded CPU version. When compared to the CPU results, the speedup factor of the GPU implementation looks quite modest. For fair performance comparison, we have chosen to compare our GPU results with the maximum number of available CPU cores. This explains the modest speedup factor. In addition to comparing pure performance, other factors come at play. For instance, if we look at the bigger picture, the CPU testbed costs several times more than the GPU testbed, has a higher power usage, as well as a lower performance per watt rating. If these factors are taken in consideration, the GPU performance is impressive.

Moreover, for consumer grade GPUs there is a stark difference between single and double precision performance. If high precision is crucial, server grade GPUs should be considered as they come with a higher double precision performance.

There is a high number of branching in the update step. Branching is expensive on a GPU, and often reduces the computational speed. However, experiments with reformulated code to reduce branching appears to increase computational times for our code. Further investigation is needed to find the most efficient balance.

Due to the lack of a hand-coded CUDA version, it is difficult to determine the performance of Mint. Hence, any future work should focus on implementing a hand-coded CUDA version of the code. Nevertheless, profiling shows that there are still room for improvement. For example, Mint has an extensive register usage and does not choose the optimal device occupancy. Mint is not optimized for our Fermi-based GPU, and lacks support for both our dual-GPU and multi-GPU setup. In the future, we would like to extend Mint with support for these features.

Chapter 7

FUTURE WORK AND CONCLUSION

7.1 Future Work

There are two approaches that can be explored from this point. One approach is to improve the algorithm, the other is to optimize the current implementation with respect to multi-core CPUs, GPUs and other upcoming parallel architectures. Below, both approaches are discussed.

7.2 Algorithmic Extensions

Although the complexity of the algorithm is $O(N)$, 3D PMM requires more sweeps than the traditional FSM to converge [56]. This makes the algorithm unsuitable for applications with strongly curved characteristics such as seismic data processing. If seismic data processing applications are to be targeted, the 3D PMM algorithm needs many sweeps to create a correct solution. Instead, methods with stricter ordering of the updates are more efficient.

Several ordered algorithms exists, but to our knowledge, the existing methods have either not been extended to apply for problems in three-dimensions, or they have not been optimized for parallel architectures. One method that addresses both of these problems to some extent is FIM. Unfortunately, the ordering of this method is considered to be too weak for complicated problems [25].

A more recent approach is the two-scale method [9], which aims to create a hybrid method by combining the best features of FMM and FSM. Contrary to FIM, the ordering of the two-scale method is more sophisticated, because the domain is divided into smaller sub-domains. Each sub-domain is then swept in a specific order, making the method more suitable for applications on domains with bending characteristics. The two-scale method works only on problems in two-dimensions, but the underlying idea might be extended to three-dimensions. In three dimensions each subdomain can be updated using the 3D PMM method. On a GPU each sub-domain can be updated in parallel. Furthermore, by changing the sorting method for the sub-domains to methods used in [18], several sub-domains can be updated in parallel using several SMXs. Such a method will have parallel features on both a small and large scale, but will also be more challenging to implement.

Finally, another interesting extension to the algorithm would be to implement task-parallelism by taking a SPMD approach. The idea of a SPMD approach is to divide the problem among threads/processes, before the final answer is synchronized and merged together. SPMD is a commonly used technique in distributed memory systems. One such approach has already been suggested [57] and consists of sweeping the domain in different directions simultaneously, before each individual part is merged together. If successful, this approach would make it possible to use MPI, opening the algorithm for computation on large-scale systems. Moreover, a SPMD extension would also make it possible to take advantage of a multiple GPU setup. The workload can then be divided among multiple GPUs using OpenMP or Pthread.

7.3 GPU Optimizations

One current shortcoming is the lack of a hand-coded CUDA-version of the algorithm. Due to the lack of a hand-coded version, it is difficult to conduct a complete performance analysis of the Mint translated code. Hence, all conclusions with respect to performance are currently based on comparison with a parallel CPU implementation.

Performance wise, the translated code appears to be performing well when compared to the CPU version of the code. Even though the generated code might be performing well, there are room for further optimizations. Profiling from section 6.7 shows that the GPU performance could be better with a better register usage. Device occupancy is another area that needs further attention. The current implementation does not saturate the memory bandwidth and the occupancy is not as high as it could be.

Moreover, Mint is neither fully optimized for Fermi or Kepler-based GPUs, but comes with several optimization flags that specifically targets Fermi based GPUs. Additionally, it would be interesting to compare the performance of the algorithm using OpenACC. By analyzing the PTX code, a performance comparison between Mint and OpenACC could reveal possible performance bottlenecks.

7.4 Targeting Future Platforms

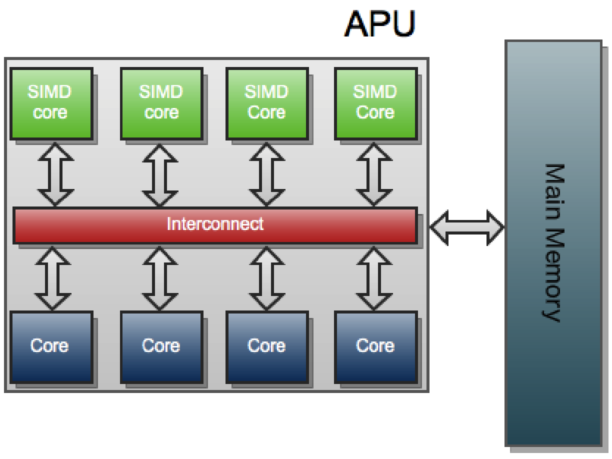


Figure 7.1: AMD's APU platform integrates multiple CPU cores and many SIMD cores (streaming multiprocessors) on the same chip.

GPUs appear to be the only alternative architecture to multi-core CPUs. However, this is about to change as new parallel architectures are under development. A particular interesting architecture is the Intel Many Integrated Core (MIC). The Intel MIC chip consists of more than 50 simple Pentium cores with 512-bit wide registers and private L1 and L2 cache, and Intel's main aim with the new architecture is to compete with GPUs. In contrary to GPUs that can run several thousands to millions of threads, each MIC core can run up to four threads. Cards based on MIC will slot into a PCIe slot and communicate with a host CPU through the PCIe bus. As a consequence, programmers must manually transfer data from the host to the device and vice versa, similarly to GPU programming. Intel has positioned OpenMP, OpenCL, OpenMPI and Cilk as the main programming models for MIC.

More importantly, MIC will employ a NUMA architecture. From a programming point of view, NUMA impose new challenges for the programmer. If the full computational power of MIC is to be exploited, a more careful distribution of data and threads is required. This is an issue that MIC's programming models fail to take into account. Hence, optimizing the 3D PMM algorithm for the MIC will require changes to the current algorithm and implementation, especially with respect to memory access. Customizing, 3D PMM for MIC will require a careful revision of the algorithm's memory access pattern in order to avoid any potential NUMA-penalties. Alternatively, work can be carried out to extend Mint with support for MIC.

AMD's Accelerated Processor Unit (APU) takes aim at integrating a GPU and a CPU on a single chip. The main advantage of such integration is that it will eliminate the need of transferring data from the host to the device. AMD is positioning OpenCL as the main programming language for programming APUs. Any future work related to APUs needs extensions with respect to OpenCL, either by manually hand-coding a version or by extending Mint with support for OpenCL.

7.5 Conclusion

We have presented a novel and an effective front propagation algorithm that works in three dimensions. The main findings of this thesis have been published as a refereed paper in Elsevier's Procedia Computer Science (acceptance rate: 30 percent). In addition to work in three dimensions, the proposed algorithm targets parallel architectures. Performance results show near-linear scaling on multi-core CPUs using OpenMP.

By using an automated C-to-CUDA source code translator called Mint, the method was also offloaded to a GPU. The performance on GPUs looks promising. Depending on the size of the grid, the GPU implementation is somewhere between 2.5 to 7 times faster than the fastest multi-threaded CPU version using 24 cores. For smaller grid sizes, the performance is considered good enough to be used interactively.

The algorithm needs to be extended for seismic processing applications, but plans for such extension is already in place. Finally, we plan to further enhance our GPU performance by reducing register spillage whilst increasing device occupancy.

Appendix A

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2012) 1–9

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

A new parallel 3D front propagation algorithm for fast simulation of geological folds

Tor Gillberg[a,c,*], Mohammed Sourouri[a,b], Xing Cai[a,b],

[a]*Computational Geoscience, CBC, Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway*
[b]*Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, 0316 Oslo, Norway*
[c]*Kalkulo AS, P.O. Box 134, 1325 Lysaker, Norway*

Abstract

We present a novel method for 3D anisotropic front propagation and apply it to the simulation of geological folding. The new iterative algorithm has a simple structure and abundant parallelism, and is easily adapted to multithreaded architectures using OpenMP. Moreover, we have used the automated C-to-CUDA source code translator, Mint, to achieve greatly enhanced computing speed on GPUs. Both OpenMP and CUDA implementations have been tested and benchmarked on several examples of 3D geological folding.

Keywords: Static Hamilton-Jacobi equations, front propagation, automated C-to-CUDA code translation

1. Introduction

The arrival time of a propagating front is often described by non-linear static Hamilton-Jacobi equations. Advanced numerical algorithms are needed to efficiently compute solutions to those equations. It is therefore a challenge to implement fast solvers, especially for large 3D simulations. Solution algorithms are often divided into two groups, Front Tracking methods and Sweeping methods. Front Tracking methods [1, 2, 3] update node values in a strictly increasing order, and thus mimic a front expanding from the initial object Γ_0. These algorithms are sequential by construction, since the front passes only one node at a time. Front tracking methods for anisotropic propagation are known as Ordered Upwind Methods. These methods are complicated, and some must be simplified for implementation. Moreover, they often assume prior knowledge of the degree of anisotropy in the problem, see for instance [4, 5, 6]. Sweeping methods [7] compute the solution from a distance perspective by iterating over directions. This makes them faster than Front Tracking methods on simple problems [8]. In the cases with complex geometries or velocities that force the characteristics to be curved, the Sweeping methods are slow because many iterations are needed before convergence [9]. Since the iteration order is predetermined, Sweeping methods [10] are readily parallelized. However, the parallelism of the traditional Sweeping algorithms is limited, and the parallel speedup is therefore modest [11]. By an alternative formulation of the stencil and iteration order, abundant parallelism can be obtained as shown with

[*]Corresponding author, `torgi@simula.no`.

the Parallel Marching Method (PMM) [12]. Early iterative algorithms trace the front in specific patterns such as expanding boxes [13], or by updating all nodes until convergence [14]. These can be made parallel as described in [15]. There are also a few algorithms that use concepts from both Front Tracking and Sweeping methods [16, 17, 18, 9].

To our knowledge, only two methods for front propagation have been successfully implemented on graphics processing units (GPUs), namely the Fast Iterative Method [17] and the PMM [12]. Of these two methods only the Fast Iterative Method is applicable in 3D, since the PMM was created for computing geodesic distances on surfaces. In this article, we present a new 3D algorithm with abundant parallelism, making the algorithm suitable for both multicore CPU and GPU architectures. Since we use the idea of an alternative stencil formulation from the PMM, we refer to our new algorithm as the 3D PMM. The 3D PMM has a highly parallel structure as nodes on an entire surface (planar cut of the 3D volume) can be updated in parallel. Moreover, thanks to the automated C-to-CUDA translator Mint [19], we can maintain an annotated serial C code for our 3D PMM method, without having to do low-level CUDA programming by hand. In comparison, the more general Fast Iterative method has a more intricate algorithmic structure, making CUDA programming much more involved.

1.1. Simulation of Folds and other Applications

Over the past several years, Statoil and Kalkulo have developed a novel paradigm for highly interactive modelling of complicated geological scenarios and processes. This methodology describes present-day geology as the realization of a series of geological events and processes along a geological timeline [20, 21]. Many processes rely on relevant surfaces and their corresponding metric property fields or maps like distances, gradients etc. [22]. These distance maps are described by the viscosity solution to the static Hamilton-Jacobi equation:

$$F\|\nabla T(\mathbf{x})\| + \psi(\mathbf{a}\cdot\nabla T(\mathbf{x})) = 1, \tag{1}$$

$$\text{given } \; T = t_0 \text{ on } \Gamma_0. \tag{2}$$

Here, Γ_0 is an initial horizon, and **a** marks the axial direction of the fold. Other folded layers are implicitly given as iso-surfaces of T, that is, the position of a front propagating from Γ_0 at different times. This equation can replicate all traditional folding classes, as defined by [23], by altering the size and sign of F, ψ and direction of **a**. For details and derivation of this system we refer to [22]. The same equation also describes the first arrival of a wave in a media in motion [24]. Figure 1(a) shows an example of an initial surface Γ_0. From this surface Figures 1(b) and (c) show two simulations of folds, created with different parameter choices. When $\psi \neq 0$, the front propagation is of the anisotropic type. In the special case where $\psi = 0$, (1) reduces to the isotropic eikonal equation, which is solved in many applications [1] and is isotropic in the sense that the velocity is independent of direction. When $F = 1$, the viscosity solution to the eikonal equation is the minimal Euclidean distance from Γ_0. The concept of characteristic curves or ray-paths is important in front propagations [22, 25]. These are curves along which a particle on the front is transported, and can also be interpreted as curves defining the shortest distance (fastest path) to Γ_0.

When modelling folds, the dip isogons, and thus the characteristic curves [22], are in general linear. For isotropic problems with linear characteristics Sweeping algorithms converge quickly [9], which motivates us to investigate related algorithms for the simulation of geologically folded structures. A powerful laptop is sufficient for an interactive geological modelling application in 2D, but the computational requirement is vastly higher in 3D. Other geological applications where the simulation of a propagating front is central include reservoir simulations [26] and simulation of seismic traveltimes [25, 27]. In seismic applications, front propagation solvers are used to simulate entire first arrival traveltime fields. By repeating the simulation from reflecting surfaces, multiple reflected traveltimes can be computed with front propagation solvers [25, 28]. Front propagation is also heavily used in medical imaging [11, 29, 30]. In these applications, the computational speed remains often a challenge. A faster solver would allow for more interactive applications, potentially leading to faster medical diagnoses and faster seismic processing. An interactive geological modelling application allows users to test many geological scenarios faster, leading to a better understanding of the inner earth. One method to achieve faster solvers is by making use of the powerful computational resources available in GPUs.

In this paper, we present a novel 3D front propagation algorithm, as well as a numerical stencil for solving (1). We also show how to parallelize the algorithm for both multicore CPU and GPU architectures. The parallelized codes are tested on several examples of geological folding, for which the parallelized codes scale well.

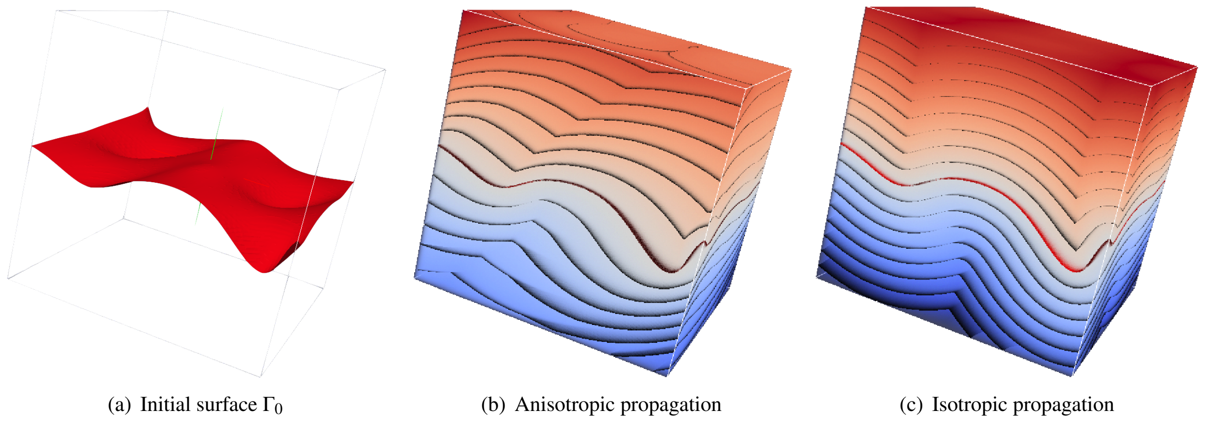

(a) Initial surface Γ_0 (b) Anisotropic propagation (c) Isotropic propagation

Figure 1: (a) An initially given surface Γ_0. (b) A folded 3D volume with $F = 1, \psi\mathbf{a} = \frac{1}{2}(-1, 1, 1)$, simulated from Γ_0. (c) A folded 3D volume from the same initial surface, but this time with $F = 1, \psi = 0$, resulting in isotropic front propagation and an Euclidean distance field.

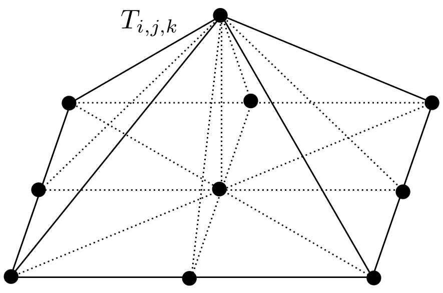
2. The 3D Parallel Marching Method

Consider a 3D box grid with nodal values $T_{i,j,k}$ where $(1, 1, 1) \leq (i, j, k) \leq (n_x, n_y, n_z)$, and with a spacing of (dx, dy, dz). In this paper we assume that values at the nodes closest to Γ_0 are given, and all the other nodes are initially set to an infinite value. (Efficient methods for initializing such values are outside the scope of this paper.) In every iteration, a smaller T value is a better approximation, since we solve for the minimal distance (the first time of arrival). It is of great importance that a new approximation is not too small, since such values are never corrected. A methodology for assuring such a discretization of (1) is presented in Appendix A. The PMM iterates through the grid in axial directions, and computes new distance values based on nodal values along the iteration direction. In the x-direction, the 3D volume is first iterated in the increasing order of the i index, and then in the decreasing order of the i index. The same sub-sweeps are also repeated in the y- and z-directions. We refer to such a full iteration as a *sweep*, which consists of 6 *sub-sweeps* of the 3D domain. Pseudocode for the sub-sweeps for the x-direction is given below.

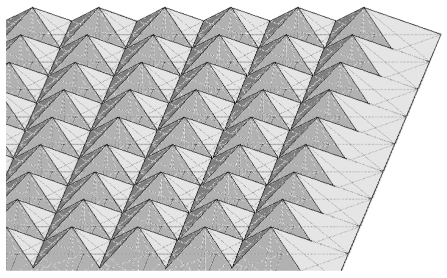
```
for $i = 2, \ldots, n_x$ do
  for all $j = 1, \ldots, n_y$ do
    for all $k = 1, \ldots, n_z$ do
      Update $T_{i,j,k}$ using values $T_{i-1,j\pm a,k\pm b}, a \in \{0, 1\}, b \in \{0, 1\}$
    end for
  end for
end for
for $i = n_x - 1, \ldots, 1$ do
  for all $j = 1, \ldots, n_y$ do
    for all $k = 1, \ldots, n_z$ do
      Update $T_{i,j,k}$ using values $T_{i+1,j\pm a,k\pm b}, a \in \{0, 1\}, b \in \{0, 1\}$
    end for
  end for
end for
```

We remark that $T_{i,j,k}$ is computed using nine nodes in the previously updated plane. The form of the update stencils are illustrated in Figure 2(a), where the sub-sweep is in the direction of the pyramid top. Every approximation's update step includes a significant amount of computations, as shown in Appendix A.

Since there are no internal dependencies between nodes on the same update plane, all nodes in the plane can be computed simultaneously. Figure 2(b) shows a plane of stencil shapes that can be solved in parallel. In the pseudocode this corresponds to computing the two inner loops in each sub-sweep entirely in parallel. Because of the simplicity of this parallelism, the algorithm is easily parallelized using OpenMP. For the OpenMP parallelization, the parallel



(a) One stencil shape



(b) Independent updates

Figure 2: (a) Nodes used when computing the generalized distance in the upward direction. (b) Illustration of the fine-grained parallel feature of the algorithm. All nodes in one plane can be computed simultaneously since they have no internal dependencies, making the algorithm suitable for parallel architectures.

region which encapsulates all the sweeps, is declared using **#pragma omp parallel**. Inside each of the six sub-sweeps, the two innermost nested loops are parallelized by adding **#pragma omp for**.

As shown in the next section, such a straightforward OpenMP parallelization achieves good speedup on multicore CPUs. Still, the OpenMP implementation is not sufficient for the application to be interactive for large grid sizes. This can be remedied by porting the algorithm to a GPU. To avoid manual GPU programming, we have made use of the automated C-to-CUDA source code translator Mint, freely available at https://sites.google.com/site/mintmodel/. Mint takes as input annotated serial C code and generates (optimized) CUDA code. The needed Mint pragmas are very similar to the OpenMP pragmas, except for two additional pragmas: **#pragma mint copy(T,toDevice,nx,ny,nz)** and **#pragma mint copy(T,fromDevice,nx,ny,nz)**, for transferring data between the host CPU and the device GPU. In Listings 1 and 2, we show two CUDA code segments that are automatically generated by Mint.

```
for (int SweepNbr = 0; SweepNbr < nbrSweeps; SweepNbr++) {

  // x-direction: sweep from bottom to top
  for (i = 1; i < 400; ++i) {
    int num2blockDim_6_1527 = (400) % 16 == 0?(400) / 16 : (400) / 16 + 1;
    int num1blockDim_6_1527 = (400) % 16 == 0?(400) / 16 : (400) / 16 + 1;
    dim3 blockDim_6_1527(16,16,1);
    dim3 gridDim_6_1527(num1blockDim_6_1527,num2blockDim_6_1527);

    mint_6_1527<<<gridDim_6_1527,blockDim_6_1527>>>(DXYP,DXZP,DYZP,dev_1_T,i,tnew,st,xt,yt,txy,xnt,
        ynt,txm,tym,txnyn,F,ax,ay,az,dzz,dxx,dyy);
    cudaThreadSynchronize();
  }

  // x-direction: sweep from top to bottom
  // ...

  // y-direction: sweep from bottom to top
  // ...
  // y-direction: sweep from top to bottom
  // ...

  // z-direction: sweep from bottom to top
  // ...
  // z-direction: sweep from top to bottom
  // ...
}
```

Listing 1: The main computational body of the `Sweep` function after automated Mint translation from C to CUDA; The number of nodes in each spatial direction is 400.

```
__global__ void mint_6_1527(double DXYP,double DXZP,double DYZP,cudaPitchedPtr dev_1_T,int i,double
    tnew,double st,double xt,double yt,double txy,double xnt,double ynt,double txm,double tym,double
    txnyn,double F,double ax,double ay,double az,double dzz,double dxx,double dyy)
{
  double *T = (double *)dev_1_T.ptr;
  int _width = dev_1_T.pitch / sizeof(double );
  int _slice = dev_1_T.ysize * _width;
  int _p_j;
  int _p_k;
{
    int _upperb_y = 400;
    int _upperb_x = 400;
    int _idx = threadIdx.x + 1;
    int _gidx = _idx + blockDim.x * blockIdx.x;
    int _idy = threadIdx.y + 1;
    int _gidy = _idy + blockDim.y * 1 * blockIdx.y;
{
      if (_gidy >= 1 && _gidy <= 400) {
            if (_gidx >= 1 && _gidx <= 400) {
        // the same computations as in the original C code
            }
      }
}
}
}
```

Listing 2: The CUDA kernel function `mint_6_1527` that is automatically generated by Mint; for the purpose of sweeping one *yz*-plane.

3. Results

In this section we present numerical results from an example of simulating a folded volume. From the same initially given surface as in Figure 1, we ran 8 sweeps on the three uniform grids with a total of $160^3, 320^3$ and 400^3 nodes. After 8 sweeps the solution has converged sufficiently. In these computations $\psi a = (-0.34, 0.4, 0.7)$, $F = 1.1$, and the domain has length 10 in x, y and z directions. We have measured the computational time for the OpenMP code using one node on the NERSC Cray XE6 "Hopper" supercomputer. Each node is equipped with two twelve-core AMD 'Magny-Cours' 2.1 GHz processors. The Mint-translated CUDA code for the same problem was executed using a Nvidia GeForce GTX 590 card. Table 1 shows elapsed times for the three grids on 1, 2, 4, 8, 16 and 24 CPU cores, as well as for the GPU. The time to transfer data to and from the GPU are included in the reported times. Both the CPU and GPU executables were compiled with the `-O3` flag, using `nvcc 4.0, V0.2.1221` and `gcc v4.3.4` respectively.

In Appendix A we present conditions that reduce the number of unnecessary computations in the update step. If all conditions are used, the number of branches increases. The update scheme already has many branches, as is indicated of profiling of the code. The profiling also indicate that the registers are under high pressure. Therefore, we have tried to formulate the update step to reduce unnecessary branching and register use. Several experiments was run with different update conditions, showing that the computational time is reduced the most when all conditions in Appendix A are used. This result holds for both the multicore and GPU versions of our code. Both the CPU and GPU codes was tested with both single and double precision. The difference between the single and double precision solutions is very small. Therefore, when modelling folds the gain in accuracy might not be worth the associated cost in computational time. Further investigation is needed before making any conclusion in this matter. In the geological modelling software, the derivative of the computed solution is used in post processes. This puts extra demand for high accuracy of the computed distance field. Thus, future research will be focused on extending the discretization to higher order schemes.

The code scales well on the multicore CPU, with near-linear speedup (1.9x) measured when conducting a strong scaling study up to 16 cores. Beyond 16 cores, the speedup drops to 1.3x. This drop in speedup is possibly due to the underlying NUMA (Non Uniform Memory Architecture) architecture on Hopper. With a more careful distribution of threads and data, we might be able to reduce the challenges NUMA imposes on the performance.

For the largest grid of 400^3 nodes, the GPU needs **25.28** seconds to perform 8 sweeps using single precision. As comparison, 24 CPU cores need **177.86** seconds to perform the 8 sweeps. When double precision is used on the GPU, the time usage is **57.42** seconds, more than **4** times faster than using 24 CPU cores (**240.05** seconds). It can also be seen in Table 1 that the speed advantage of GPU computations increases with the grid size.

At the moment of writing, a GeForce GTX 590 GPU costs around $500 USD, while one AMD 'Magny-Cours' 2.1 GHz costs more than $1000 USD. Depending on the grid size and precision, using a GPU will deliver 2-7x the performance, while costing[1] four times less than a 24-core CPU node. This makes the results even more impressive.

[1]Other system parts such as memory, motherboard etc., are not taken into account.

Table 1: Computational times for three grids with a total of N nodes using single (top table) and double precision. t_i is the CPU time for i cores. The speedup factors S_1 and S_{24} are calculated using the running time from 1 core and 24 cores (the highest number of CPU cores available). The speedup factor increases as N increases, possibly due increased computational complexity that amortises the data transfer cost from the CPU to the GPU. The data in Tabular (a) are single precision results while data in Tabular (b) are double precision results.

N	t_1	t_2	t_4	t_8	t_{16}	t_{24}	t_{GPU}	S_1	S_{24}
160^3	194.17	100.66	52.62	27.20	14.07	10.22	2.43	79.9x	4.2x
320^3	1795.86	822.48	423.87	219.39	112.57	81.40	14.84	121.0x	5.4x
400^3	3543.14	1628.12	853.50	430.59	223.55	177.86	25.28	140.1x	7.0x

(a) Computational times (t) and GPU speedup (S) for single precision

N	t_1	t_2	t_4	t_8	t_{16}	t_{24}	t_{GPU}	S_1	S_{24}
160^3	217.54	112.85	58.93	30.44	15.73	11.40	2.43	89.5x	2.4x
320^3	2016.50	911.12	472.84	245.08	125.08	90.07	31.09	64.8x	2.9x
400^3	3886.31	1799.22	928.89	481.99	246.80	240.05	57.42	67.6x	4.1x

(b) Computational times (t) and GPU speedup (S) for double precision

4. Discussions

Mint has been shown to deliver good performance on 3D finite difference codes [19]. Our algorithm is a 3D finite difference solver, but a non-traditional one. The grid is iterated in specific orders and a non-traditional stencil is used. Mint has delivered a surprisingly good GPU performance for our 3D PMM. A detailed comparison of the Mint-translated code with a hand-coded CUDA version would be an interesting investigation. We have experimented with the formulation of the update step, to search for an efficient formulation. Those optimization investigations indicate that the high number of branches introduced from conditioning the update computations, reduces the computational speed. Nevertheless, profiling has indicated some further optimization possibilities, for both CPU and GPU implementations. For instance, the current register use is very extensive. A better use of the registers might improve both implementations.

An interesting algorithmic extension is to try ideas from [10], in which approaches for parallelization of the otherwise sequential Fast Sweeping Method are presented. One of the suggested ideas there is to sweep the domain in different directions at the same time on copies of the data structure, and then synchronize the results. Sub-sweeps in different directions can for instance be computed on several GPUs simultaneously. The approach of performing full sweeps of subdomains from the same paper may increase the convergence rate of the algorithm. Weber et al. [12] present a variant of the 2D PMM method to make it run faster on a GPU. Similar extensions in 3D are possible, and might assure good reuse of transferred data.

Although an $O(N)$ method [12], the PMM method often needs more Sweeps than the Fast Sweeping method to converge. Therefore, the algorithm is not suitable for applications with strongly curved characteristics, such as seismic data processing. For such applications the updates need to be ordered somehow. The Fast Iterative Method is one approach to this, but the ordering is too weak for complicated problems [17]. A related method exists for sequential 2D code on isotropic examples [9], in which subdomains are swept in a specified order. This idea can be extended to 3D and parallelized for GPUs by sweeping a list of subdomains simultaneously, using one streaming multiprocessor each, on which the streaming vector processors make use of the parallelism of the 3D PMM method. Furthermore, the subdomains can be ordered using an approach similar to that of [16] to ensure a stronger ordering, and convergence also on anisotropic problems. Such an algorithm will be efficient also on problems with bending characteristics.

5. Conclusion

Simulating a propagating front is a computationally challenging problem, especially in 3D applications. Simulations are needed in several applications, where the solution is needed within a few seconds for the software to be used in an interactive manner. In 3D, sequential algorithms are only applicable on small grid sizes. We have presented a simple 3D Parallel Marching Method and applied it to the simulation of geological folding. The algorithm can be easily implemented on parallel architectures. Numerical experiments using OpenMP show near-linear scaling on multicore CPUs. Using the automated C-to-CUDA code translator Mint, we obtained a CUDA implementation without manual GPU programming. The GPU implementation runs approximately 2.4-7 times faster than the fastest multi-threaded version on 24 CPU cores, giving hope to compute large 3D grids interactively in the future.

6. Acknowledgments

The presented work was funded by Statoil through its Academia Program, and the Research Council of Norway under grant 202101/I40. The work has been conducted at Kalkulo AS, a subsidiary of Simula Research Laboratory.

Appendix A. Conditional Upwind Approximations

As in most front propagation methods, it is of great importance that approximations are computed from upwind values. Upwind values are values that are passed by the front. Monotonic convergence is a fundamental property for convergence toward the viscosity solution for most algorithms [17, 1, 14]. A too small approximation will not be increased, since that would contradict the monotonicity assumption. Therefore, one must assure that the computed value uses solution values that are upwind from the updated node. We assert this by computing the characteristic curve of the approximation, and make sure that it is embedded in the convex hull of the nodes used in the computations. If it is not, the new approximation is rejected. A similar approach for isotropic problems are presented in [31], where the entrance point is used to find the rays in a seismic processing setting. From [22] we have the characteristics $x(s)$ to (1)

$$\frac{\partial x}{\partial s} = F\nabla T + \psi \mathbf{a}|\nabla T|. \tag{A.1}$$

Consider the stencil shape as given in Figure 2(a), where a new solution is sought for the pyramid-top node value $T_{i,j,k}$, and the nine nodes in the lower plane all have the third coordinate as $k-1$. The nodes on the lower plane are divided in eight groups of three nodes that form trirectangular tetrahedras with $T_{i,j,k}$ as apex. Similarly, 16 groups of two nodes forming right-angled triangles are created (dotted lines on the lower plane) as well as nine groups of one node each. From each of these groups solution estimates are created. If $T_{i,j,k}$ is bigger than the smallest acceptable of these approximations, we update the value at (i, j, k) with the new estimate. With estimates of the gradient of T and equation (A.1), the entrance point in the lower plane is given as

$$x_e = -dz\frac{F\frac{\partial T}{\partial x} + \psi \mathbf{a}_x|\nabla T|}{F\frac{\partial T}{\partial z} + \psi \mathbf{a}_z|\nabla T|}, \quad \text{and} \quad y_e = -dz\frac{F\frac{\partial T}{\partial y} + \psi \mathbf{a}_y|\nabla T|}{F\frac{\partial T}{\partial z} + \psi \mathbf{a}_z|\nabla T|}. \tag{A.2}$$

Assume the nodes $T_{i,j,k-1}, T_{i+1,j,k-1}$ and $T_{i+1,j+1,k-1}$ are three nodes in a trirectangular shape. With these nodes we estimate the partial derivatives in x and y direction with

$$\frac{\partial T}{\partial x} = \frac{T_{i+1,j,k-1} - T_{i,j,k-1}}{dx}, \quad \text{and} \quad \frac{\partial T}{\partial y} = \frac{T_{i+1,j+1,k-1} - T_{i+1,j,k-1}}{dy}. \tag{A.3}$$

Using these two estimates, we directly discretize (1) and get $\frac{\partial T}{\partial z}$ as the solution of a second degree polynomial. From (A.2) we get the entrance coordinates x_e, y_e, and the solution estimate $T^{new}_{i,j,k} = T_{i,j,k-1} + dz\frac{\partial T}{\partial z}$ is accepted if

$$0 < \min x_e, y_e, \quad y_e dx < x_e dy, \quad \text{and} \quad x_e < dxx. \tag{A.4}$$

That is to assure the entrance point is within the convex hull of the nodes used in the stencil. The remaining stencils using three values, are identical up to a rotation and reflection.

For the two node group using $T_{i,j,k-1}$ and $T_{i+1,j,k-1}$ we estimate $\frac{\partial T}{\partial x}$ with $\frac{T_{i+1,j,k-1} - T_{i,j,k-1}}{dx}$. That the characteristic curve to $T_{i,j,k}$ cut the line segment between $(i, j, k-1)$ and $(i+1, j, k)$ is to say that $y_e = 0$ of (A.2), that is

$$\frac{\partial T}{\partial y} = \frac{-\psi a_y|\nabla T|}{F}. \tag{A.5}$$

Together with (1) we get $\frac{\partial T}{\partial z}$ as the solution to a second degree polynomial. If the entrance point x_e from (A.2) satisfies $0 < x_e < dx$, we accept the new approximation. The remaining 15 stencils using two values are identical up to a rotation and reflection.

When only the value at $T_{i,j,k-1}$ is used, the characteristic curve must go from $(i, j, k-1)$ through (i, j, k). That is $x_e = 0, y_e = 0$ in (A.2), resulting in

$$F\frac{\partial T}{\partial x} + \psi a_x|\nabla T| = 0, \text{ and } F\frac{\partial T}{\partial y} + \psi a_y|\nabla T| = 0. \tag{A.6}$$

The traveltime solution for this system is easiest found using the group velocity v_G, that is the velocity in the direction of motion [7, 3]. In our case we have the the group velocity in the z-direction as $v_G = F\frac{\frac{\partial T}{\partial z}}{|\nabla T|} + a_z$, and the arrival time to $T_{i,j,k} = T_{i,j,k-1} + \frac{dz}{v_G}$. For a general point with value $T_{l,m,n}$ at the distance $\mathbf{x} = (x, y, z)$ from index (i, j, k) the corresponding solution is

$$T_{i,j,k} = T_{l,m,n} + \frac{\mathbf{x}\cdot\mathbf{x}}{\mathbf{a}\cdot\mathbf{x} + F\sqrt{(1-|\mathbf{a}|^2)\frac{|\mathbf{x}|^2}{F^2} + \frac{(\mathbf{a}\cdot\mathbf{x})^2}{F^2}}}. \tag{A.7}$$

In 2D this result is the same as the analytical solution of a point source as shown in [24].

Reducing the number of redundant computations

In isotropic front propagations, only strictly smaller nodes should be used for creating solution estimates with upwind stencils [1]. For a general anisotropic problem the same principle does not hold. However, at least one of the used nodes must be smaller than the old estimate for there to be a possibility of an acceptable solution estimate [16]. In the above stencil formulation this correspond to $T^{old}_{i,j,k} > \min_{(a,b)\in\{(0,0),(1,0),(1,1)\}} T_{i+a,j+b,k-1}$ in the three node case, $T^{old}_{i,j,k} > \min_{a\in\{0,1\}} T_{i+a,j,k-1}$ for the two node case, and $T^{old}_{i,j,k} > T_{i,j,k-1}$ for the one node case.

Moreover, we can derive the following condition for the characteristics entrance point x_e and y_e to be grater than 0

$$n_x = \frac{\frac{\partial T}{\partial x}}{|\nabla T|} < \frac{a_x}{F}, \text{ and } n_y = \frac{t_y}{|\nabla T|} < \frac{a_y}{F}. \tag{A.8}$$

If $a_x < 0$ then we must have $\frac{\partial T}{\partial x} < 0$, otherwise the solution will not be accepted, and hence we need not to create the estimate. The corresponding argument holds for the y_e entrance point.

Remark; Signed Distance

When simulating folds, one must distinguish between the inside and outside of the structure. In order for the fold to be consistent, the axial direction **a** is negative on the inside, and positive on the outside. The same holds for the solution T. Accordingly, the 3D PMM adjusts the initialisation step slightly. The initialised data is set to $-\infty$ on the inside, and $+\infty$ on the outside. The sign of $T_{i,j,k}$ during the update step is saved locally, and the update is performed with absolute values of the nodes. If a new solution is found, it is saved with the same sign as the previous approximation was.

References

[1] J. A. Sethian, Level Set Methods and Fast Marching Methods, Cambridge University Press, 1999.

[2] F. Qin, Y. Luo, K. B. Olsen, W. Cai, G. T. Schuster, Finite-difference solution of the eikonal equation along expanding wavefronts, Geophysics 57 (3) (1992) 478–487.

[3] Y. Wang, T. Nemeth, R. Langan, An expanding-wavefront method for solving the eikonal equations in general anisotropic media, Geophysics 71 (5) (2006) T129.

[4] E. Cristiani, A fast marching method for Hamilton-Jacobi equations modeling monotone front propagations, Journal of Scientific Computing 39 (2) (2009) 189–205.

[5] J. Sethian, A. Vladimirsky, Ordered upwind methods for static Hamilton-Jacobi equations: theory and algorithms, SIAM Journal on Numerical Analysis 41 (1) (2004) 325–363.

[6] K. Alton, Dijkstra-like ordered upwind methods for solving static Hamilton-Jacobi equations, Ph.D. thesis (2010).

[7] J. Qian, Y.-T. Zhang, H.-K. Zhao, A fast sweeping method for static convex hamilton-jacobi equations, J. Sci. Comput. 31 (1-2) (2007) 237–271.

[8] P. A. Gremaud, C. M. Kuster, Computational study of fast methods for the eikonal equation, SIAM Journal on Scientific Computing 27 (6) (2006) 1803–1816.

[9] A. Chacon, A. Vladimirsky, Fast two-scale methods for eikonal equations, SIAM Journal on Scientific Computing 34 (2) (2012) A547–A578.

[10] H.K.Zhao, Parallel implementations of the fast sweeping method, Journal of Computational Mathematics 25 (4) (2007) 421 – 429.

[11] S. Li, K. Mueller, M. Jackowski, D. Dione, L. Staib, Physical-space refraction-corrected transmission ultrasound computed tomography made computationally practical, in: Medical Image Computing and Computer-Assisted Intervention - MICCAI 2008, Vol. 5242 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008, pp. 280–288.

[12] O. Weber, Y. S. Devir, A. M. Bronstein, M. M. Bronstein, R. Kimmel, Parallel algorithms for approximation of distance maps on parametric surfaces, ACM Transactions on Graphics 27 (4) (2008) 1–16.

[13] J. Vidale, Finite-difference calculation of travel times, Bulletin of the Seismological Society of America 78 (6) (1988) 2062–2076.

[14] E. Rouy, A. Tourin, A viscosity solutions approach to shape-from-shading, SIAM J. Numer. Anal. 29 (3) (1992) 867 – 884.

[15] P. Podvin, I. Lecomte, Finite difference computation of traveltimes in very contrasted velocity models: a massively parallel approach and its associated tools, Geophysical Journal International 105 (1991) 271–284.

[16] T. Gillberg, A Semi-Ordered Fast Iterative Method (SOFI) for Monotone Front Propagation in Simulations of Geological Folding, in: MODSIM2011, 19th International Congress on Modelling and Simulation, 2011, pp. 631–647.

[17] W.-K. Jeong, R. T. Whitaker, A fast iterative method for eikonal equations, SIAM Journal on Scientific Computing 30 (5) (2008) 2512–2534.

[18] S. Bak, J. McLaughlin, D. Renzi, Some Improvements for the Fast Sweeping Method, SIAM Journal on Scientific Computing 32 (2010) 2853–2874.

[19] D. Unat, X. Cai, S. Baden, Mint: Realizing CUDA performance in 3D stencil methods with annotated C, in: Proceedings of the 25th International Conference on Supercomputing (ICS'11), ACM Press, 2011, pp. 214–224.

[20] S. A. Petersen, Ø. Hjelle, Earth recursion, an important component in shared earth model builders, EAGE 70th Conference & Exhibition, Extended Abstracts.

[21] A. Tveito, A. M. Bruaset, O. Lysne (Eds.), Simula Research Laboratory – by thinking constantly about it, Springer, 2010, Ch. Turning Rocks into Knowledge.

[22] Ø. Hjelle, S. A. Petersen, A Hamilton-Jacobi framework for modeling folds in structural geology, Mathematical Geosciences 43 (7) (2011) 741–761.

[23] J. G. Ramsay, Folding and fracturing of rocks, McGraw-Hill, New York and London, 1967.

[24] E. Kornhauser, Ray Theory for Moving Fluids, The Journal of the Acoustical Society of America 25 (5) (1953) 945–949.

[25] N. Rawlinson, M. Sambridge, Multiple reflection and transmission phases in complex layered media using a multistage fast marching method, Geophysics 69 (5) (2004) 1338–1350.

[26] I. Berre, K. H. Karlsen, K.-A. Lie, J. R. Natvig, Fast computation of arrival times in heterogeneous media, Computational Geosciences 9 (4) (2005) 179–201.

[27] A. M. Popovici, J. A. Sethian, 3-D imaging using higher order fast marching traveltimes, Geophysics 67 (604, Issue 2).

[28] J.-W. Huang, G. Bellefleur, Joint transmission and reflection traveltime tomography using the fast sweeping method and the adjoint-state technique, Geophysical Journal International 188 (2) (2012) 570–582.

[29] W.-K. Jeong, P. T. Fletcher, R. Tao, R. Whitaker, Interactive visualization of volumetric white matter connectivity in DT-MRI using a parallel-hardware Hamilton-Jacobi solver, IEEE Transactions on Visualization and Computer Graphics 13 (2007) 1480–1487.

[30] Q. Lin, Enhancement, extraction, and visualization of 3d volume data [elektronisk resurs], Ph.D. thesis, Linköping University, Institute of Technology (2003).

[31] J. Zhang, Y. Huang, L.-P. Song, Q.-H. Liu, Fast and accurate 3-D ray tracing using bilinear traveltime interpolation and the wave front group marching, Geophysical Journal International 184 (3) (2011) 1327–1340.

Appendix B

N	Precision	Parameter	Mint
160^3	Single	Unrolled	3.071949
320^3	Single	Unrolled	17.105794
400^3	Single	Unrolled	28.852771
160^3	Single	Optimal	2.432136
320^3	Single	Optimal	14.84756
400^3	Single	Optimal	25.284259
160^3	Double	Unrolled	6.021
320^3	Double	Unrolled	35.899724
400^3	Double	Unrolled	65.26106
160^3	Double	Optimal	4.776959
320^3	Double	Optimal	31.097744
400^3	Double	Optimal	57.420805

Table B.1: Computational times for three grids with a total of N nodes. In the Unrolled parameter the number of branches are reduced.

The same experiments where also carried out using OpenMP. The results are shown in Table B.2. Only the results for the two first grid sizes and single precision are shown. The result difference for the largest grid were inseparable. For visual clarity, the results were discarded.

N	Parameter	t_1	t_2	t_4	t_8	t_{16}	t_{24}
160^3	Unrolled	217.647678	112.566775	58.702850	30.286851	15.660032	11.326857
320^3	Unrolled	1792.445170	908.953014	471.882546	243.368088	124.792670	88.859847
160^3	Optimal	194.174903	100.662550	52.623391	27.200562	14.073829	10.226937
320^3	Optimal	1795.863088	822.488735	423.871437	219.399592	112.579793	81.400111

Table B.2: A comparison of computational times for three grids with a total of N nodes with reduced branching.

The difference between the version with reduced branching and the regular version of the code follows the same trend as for the results from the GPU. This is surprising as CPUs often come with mechanisms as branch prediction to improve branching performance.

References

[1] David B. Kirk abd Wen-mei W. Hwu. *Programming Massively Parallel Processors, A Hands-on Approach*. Morgan Kaufmann, 2011.

[2] David H. Bailey. Highly parallel perspective: Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review*, 4(8):54–55, 1991.

[3] Stanley Bak, Joyce McLaughlin, and Daniel Renzi. Some improvements for the fast sweeping method. *SIAM J. Sci. Comput.*, 32(5):2853–2874, September 2010.

[4] Inga Berre, Kenneth Hvistendal Karlsen, Knut-Andreas Lie, and Jostein R. Natvig. Fast computation of arrival times in heterogeneous media. *Computational Geosciences*, 9(4):179–201, November 2005.

[5] The OpenMP Architecture Review Board. Openmp.org. Web Page. http://www.openmp.org/.

[6] Timothy Brecht. On the importance of parallel application placement in numa multiprocessors. In *Proceedings for the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, USA, 1993.

[7] Alexander M. Bronstein, Michael M. Bronstein, and Ron Kimmel. Weighted distance maps computation on parametric three-dimensional manifolds. *J. Comput. Phys.*, 225(1):771–784, July 2007.

[8] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. Forestgomp: An efficient openmp environment for numa architectures. *International Journal of Parallel Programming*, 38:418–439, 2010. 10.1007/s10766-010-0136-3.

[9] Adam Chacon and Alexander Vladimirsky. Fast two-scale methods for eikonal equations. *SIAM Journal on Scientific Computing*, 34(2):A547–A578, 2012.

[10] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *Micro, IEEE*, 30(2):16 –29, march-april 2010.

[11] Nvidia Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009. http://goo.gl/ooikw.

[12] Nvidia Corporation. Ptx: Parallel thread execution isa version 2.3. Web, March 2011.

[13] Nvidia Corporation. *Whitepaper: NVIDIA GeForce GTX 680*, 2012. http://goo.gl/hEHtZ.

[14] W.J. Dally, F. Labonte, A. Das, P. Hanrahan, Jung-Ho Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T.J. Knight, and U.J. Kapasi. Merrimac: Supercomputing with streams. In *Supercomputing, 2003 ACM/IEEE Conference*, page 35, nov. 2003.

[15] P.-E. Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980.

[16] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 2011.

[17] Jianbin Fang, A.L. Varbanescu, and H. Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216 –225, sept. 2011.

[18] Tor Gillberg. A semi-ordered fast iterative method (sofi) for monotone front propagation in simulations of geological folding. In F. Chan, D. Marinova, and B. Anderssen, editors, *MODSIM2011, 19th International Congress on Modelling and Simulation*, pages 631–647. Modelling and Simulation Society of Australia and New Zealand, Modelling and Simulation Society of Australia and New Zealand Inc. (MSSANZ), 2011.

[19] Tor Gillberg, Øyvind Hjelle, and Are Magnus Bruaset. A parallel 3d front propagation algorithm for simulation of geological folding on gpus. In *EAGE 74th Conference & Exhibition, Extended Abstracts*. EAGE, EarthDoc, 2012.

[20] Tor Gillberg, Mohammed Sourouri, and Xing Cai. A new parallel 3d front propagation algorithm for fast simulation of geological folds. In *Proceedings of the International Conference on Computational Science, ICCS 2012*, Procedia Computer Science. Elsevier, 2012.

[21] Øyvind Hjelle and Steen Agerlin Petersen. A hamilton-jacobi framework for modeling folds in structural geology. *Mathematical Geosciences*, 43(7):741–761, 2011.

[22] Shu-Ren Hysing and Stefan Turek. The eikonal equation: Numerical efficiency vs. algorithmic complexity on quadrilateral grids. In *Proceedings of Algoritmy*, pages 22–31, Podbanské, March 2005.

[23] Won-Ki Jeong, P.T. Fletcher, Ran Tao, and R.T. Whitaker. Interactive visualization of volumetric white matter connectivity in dt-mri using a parallel-hardware hamilton-jacobi solver. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1480 –1487, nov.-dec. 2007.

[24] Won ki Jeong. *Interactive Three-Dimensional Image Analysis and Visualization Using Graphics Hardware*. PhD thesis, University of Utah, USA, December 2008.

[25] Won ki Jeong and Ross T. Whitaker. A fast iterative method for eikonal equations. *SIAM journal on scientific computing*, 30:2512–2534, 2008.

[26] ET Kornhauser. Ray theory for moving fluids. *The Journal of the Acoustical Society of America*, 25(5):945–949, 1953.

[27] Jens Krueger, David Donofrio, John Shalf, Marghoob Mohiyuddin, Samuel Williams, Leonid Oliker, and Franz-Josef Pfreund. Hardware/software co-design for energy-efficient seismic modeling. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 73:1–73:12, New York, NY, USA, 2011. ACM.

[28] Hans Petter Langtangen. *Computational Partial Differential Equations, Numerical Methods and Diffpack Programming*. Springer-Verlag, Berlin, 2nd edition, 2003.

[29] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38(3):451–460, jun 2010.

[30] Shengying Li, Klaus Mueller, Marcel Jackowski, Donald Dione, and Lawrence Staib. Physical-space refraction-corrected transmission ultrasound computed tomography made computationally practical. In *Proceedings of the 11th International Conference on Medical Image Computing and Computer-Assisted Intervention, Part II*, MICCAI '08, pages 280–288, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] Sophie Michelet. 12.510 introduction to seismology. Web page, February 2005.

[32] International Conference on Computational Science. Iccs 2012 empowering science through computing. Web, April 2012.

[33] Stan Openshaw and Ian Turton. *High Performance Computing and the Art of Parallel Programming*. Routledge, 1st edition, Nov 1999.

[34] Peter S. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[35] Scott Pakin. Ten ways to fool the masses when giving performance results on gpus. Web, December 2011.

[36] S. A. Petersen and Øyvind Hjelle. Earth recursion, an important component in shared earth model builders. *EAGE 70th Conference & Exhibition, Extended Abstracts*, 2008.

[37] Alexander Mihai Popovici and James A. Sethian. 3-d imaging using higher order fast marching traveltimes. *Geophysics*, 67(2):604–609, 2002.

[38] N. Rawlinson and M. Sambridge. Multiple reflection and transmission phases in complex layered media using a multistage fast marching method. *Geophysics*, 69(5):1338–1350, 2004.

[39] Mary Hall Saman Amarasinghe. Programming challenges for exascale computing, July 2011.

[40] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proc. Nat. Acad. Sci. USA*, 93:1591–1595, February 1996.

[41] J. A. Sethian. *Level Set Methods and Fast Marching Methods*, page 4. Cambridge University Press, New York, 2nd edition, 1999.

[42] J. A. Sethian. *Level Set Methods and Fast Marching Methods*, page 5. Cambridge University Press, New York, 2nd edition, 1999.

[43] J. A. Sethian. *Level Set Methods and Fast Marching Methods*, page 47. Cambridge University Press, New York, 2nd edition, 1999.

[44] J. A. Sethian. *Level Set Methods and Fast Marching Methods*, page 93. Cambridge University Press, New York, 2nd edition, 1999.

[45] J. A. Sethian. *Level Set Methods and Fast Marching Methods*, page 87. Cambridge University Press, New York, 2nd edition, 1999.

[46] James A. Sethian and Alexander Vladimirsky. Ordered upwind methods for static hamilton–jacobi equations: Theory and algorithms. *SIAM J. Numer. Anal.*, 41(1):325–363, January 2003.

[47] Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. Data and thread affinity in openmp programs. In *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?*, MAW '08, pages 377–384, New York, NY, USA, 2008. ACM.

[48] Øyvind Hjelle Tor Gillberg and Are Magnus Bruaset. Accuracy and efficiency of stencils for the eikonal equation in earth modelling. *Computational Geosciences*, 2012. Accepeted for publishing.

[49] J.N. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *Automatic Control, IEEE Transactions on*, 40(9):1528 –1538, sep 1995.

[50] Didem Unat. *Domain-Specific Translator and Optimizer for Massive On-Chip Parallelism*. PhD thesis, University of California San Diego, March 2012.

[51] Didem Unat, Xing Cai, and Scott Baden. Mint: Realizing cuda performance in 3d stencil methods with annotated c. In David K. Lowenthal, Bronis R. de Supinski, and Sally A. McKee, editors, *Proceedings of the 25th International Conference on Supercomputing (ICS'11)*, pages 214–224. ACM Press, 2011.

[52] Alexander Boris Vladimirsky. *Fast methods for static Hamilton-Jacobi Partial Differential Equations*. PhD thesis, University of California, Berkeley, 2001.

[53] Vasily Volkov. Better performance at lower occupancy,. Web, September 2010. http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf.

[54] Ofir Weber, Yohai S. Devir, Alexander M. Bronstein, Michael M. Bronstein, and Ron Kimmel. Parallel algorithms for approximation of distance maps on parametric surfaces. *ACM Trans. Graph.*, 27(4):104:1–104:16, November 2008.

[55] M. A. Weiss. *Data Structures and Algorithm Analysis in Java*, page 304. Addison Wesley, Reading, MA, 2nd edition, 1999.

[56] Hongkai Zhao. A fast sweeping method for eikonal equations. *Mathematics of Computation 74A*, pages 603–627, 2004.

[57] Hongkai Zhao. Parallel implementations of the fast sweeping method. *Journal of Computational Mathematics 25*, 4:421–429, 2007.