NTNU
Norwegian University of
Science and Technology

Seismic Shot Processing on GPU

Owe Johansen

Master of Science in Computer Science

Submission date:	June 2009
Supervisor:	Anne Cathrine Elster, IDI
Co-supervisor:	John Hybertsen, StatoilHydro

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Seismic processing applications typically process large amounts of data and are very computationally demanding. A wide variety of high performance applications have recently been modified to use GPU devices for off-loading their computational workload, often with considerable performance improvements.

In this project, we will in collaborations with StatoilHydro, explore how a seismic application used for handling shot data on large clusters, can take advantage of GPU technoligies. The project may include looking at relevant GPU libraries and compilers as well as considering porting relevant library components currently provided by the StatiliHydro application enviroment.

Performance comparison with the original codes will be included, as well as an evaluation of the developement effort required for implementing such techniques.

Assignment given: 23. January 2009
Supervisor: Anne Cathrine Elster, IDI

Abstract

Today's petroleum industry demand an ever increasing amount of computational resources. Seismic processing applications in use by these types of companies have generally been using large clusters of compute nodes, whose only computing resource has been the CPU. However, using Graphics Processing Units (GPU) for general purpose programming is these days becoming increasingly more popular in the high performance computing area. In 2007, NVIDIA corporation launched their framework for developing GPU utilizing computational algorithms, known as the Compute Unied Device Architecture (CUDA), a wide variety of research areas have adopted this framework for their algorithms. This thesis looks at the applicability of GPU techniques and CUDA for off-loading some of the computational workload in a seismic shot modeling application provided by StatoilHydro to modern GPUs.

This work builds on our recent project that looked at providing checkpoint restart for this MPI enabled shot modeling application. In this thesis, we demonstrate that the inherent data parallelism in the core finite-difference computations also makes our application well suited for GPU acceleration. By using CUDA, we show that we could do an efficient port our application, and through further refinements achieve significant performance increases.

Benchmarks done on two different systems in the NTNU IDI (Department of Computer and Information Science) HPC-lab, are included. One system is a Intel Core2 Quad Q9550 @2.83GHz with 4GB of RAM and an NVIDIA GeForce GTX280 and NVIDIA Tesla C1060 GPU. Our second testbed was an Intel Core I7 Extreme (965 @3.20GHz) with 12GB of RAM hosting an NVIDIA Tesla S1070 (4X NVIDIA Tesla C1060). On this hardware, speedups up to a factor of 8-14.79 compared to the original sequential code are achieved, confirming the potential of GPU computing in applications similar to the one used in this thesis.

Problem Description

Seismic processing applications typically process large amounts of data and are very computationally demanding. A wide variety of high performance applications have recently been modified to use GPU devices for off-loading their computational workload, often with considerable performance improvements.

In this project, we will in collaborations with StatoilHydro, explore how a seismic application used for handling shot data on large clusters, can take advantage of GPU technologies.

The project may include looking at relevant GPU libraries and compilers as well as considering porting relevant library components currently provided by the StatoilHydro application environment. Performance comparison with the original codes will be included, as well as an evaluation of the development effort required for implementing such techniques.

Acknowledgments

There have been several people involved with making the work in this thesis possible. Here, I would like to especially thank the following:

- Dr. Anne Cathrine Elster, who has been my main thesis advisor at the Norwegian University of Science and Technology (NTNU). Dr. Elster has helped getting the project approved, and has also given valuable advice during the course of the work.

- StatoilHydro, for supplying the application subject to this thesis, as well as the data sets used.

- John Hybertsen of StatoilHydro. Mr. Hybertsen helped define the problem of this thesis, and has also been a very positive influence by supporting the work.

- Jon André Haugen of StatoilHydro. Mr. Haugen is one of the chief developers of the application subject to the work in this thesis. His help with explaining the finer points of the application, especially the geophysical aspects, as well has his feedback on this work, has been invaluable.

- Dr. Joachim Mispel of StatoilHydro. Dr. Mispel, also one of the Chief developers of the library (SPL) which the application in this thesis is based on, has been very helpful by giving useful comments on this work, as well as participating in clarifying discussions.

Contents

List of Figures

Chapter 1

Introduction

Using Graphics Processing Units (GPU) for general purpose programming is becoming increasingly popular in high performance computing (HPC). Since NVIDIA corporation in 2007 launched a framework for developing GPU utilizing computational algorithms, known as the Compute Unified Device Architecture (CUDA), a wide variety of research areas have adopted this framework for their algorithms. Examples of research disciplines using CUDA include computational biophysics, image and signal processing, geophysical imaging, game physics, molecular dynamics and computational fluid dynamics (see e.g. [1], [2], [3] and [4]).

The massively parallel architecture of CUDA enabled GPU devices makes them a perfect fit for algorithms which behave in a highly data parallel manner. Reported algorithm speedups in excess of 100x compared to conventional CPU implementations, and relatively low cost, further motivates the use of such hardware for computationally intensive algorithms.

1.1 Our Targeted Seismic Application

The work presented in this thesis continues the work we previously did in [5], where StatoilHydro provided us with a seismic shot modeling application and sample synthetic data. The application is MPI based for parallel execution on the large production compute clusters at StatoilHydro. In [5], we implemented checkpoint/restart functionality, which guaranteed that all the work assigned to the application would complete regardless of potential situations in which nodes of the compute cluster failed.

The main focus of the work presented in this thesis will be different compared to the previous work in that we here will attempt to improve the efficiency of the application by off-loading some of the numerical calculations to the GPU, where as we previously focused on improving application consistency and fault tolerance. The application we worked on back then was an earlier revision of the one used in this thesis.

1.2 Project Goals

In this project, we will in collaborations with StatoilHydro explore the potential of using CUDA based GPU offloading for improving the performance of the provided seismic shot modeling application. The provided application is currently being used on a production compute cluster at StatoilHydro.

We will attempt to include a review of existing GPU enabled HPC libraries and compilers, as well as consider porting library components of parts of the StatoilHydro application environment to GPU.

We will conduct performance comparisons of the developed GPU enabled application and the original, and finally evaluate the development effort required for implementing the CUDA enabled GPU utilizing components.

1.3 Outline

The contents of this thesis is structered as follows:

- Chapter 2 introduces our application targeted for GPU computational offloading. It starts off by describing the fundamentals of marine seismic acquisition, and proceeds by describing the workings of our application, as well as how it relates to this principle.

- Chapter 3 will explore the details of GPU programming, in particular with respect to NVIDIA's CUDA framework. It will also review the basics of the Finite Difference Method, as well as mention some of the works done in the field of GPU programming on applications with various algorithmic commonalities to our target application.

- Chapter 4 starts off with profiling a run of the original application, which is subsequently used to indicate which areas to target for GPU offloading. Following the profiling results, the Chapter describes the functionality of the selected target areas, and proceeds by detailing the process of modifying them so that they take advantage of GPU driven computational offloading. 3 GPU utilizing implementations will be developed, with the last being the most optimal. The Chapter concludes by describing the implementation of multiple GPU support in the application.

- Chapter 5 contains performance measurements of the developed GPU implementations. The Chapter starts off describing test methodology and the test environment, followed by benchmarking results for all developed GPU implementations, as well as the original. In the end, the results obtained from the benchmarking are discussed.
- Chapter 6 concludes the work done in this thesis, and suggests some possible alternatives for future work.

Chapter 2

Seismic Shot Processing

In this Chapter we will introduce the application in which we intend to implement GPU computational offloading techniques. We will start off by introducing the basics of seismic acquisition, and proceed by describing the workings of our application and how it relates to this principle.

2.1 Marine Seismic Acquisition

Marine seismic acquisition (see Chapter 7 of [6]), is the process where a vessel (boat) releases pressure waves (shots) from an air gun (source) underneath the sea surface. The vessel tows a cable (streamer) with listening devices (hydrophones). When the pressure wave from a shot reaches an interface in the sub surface, some of its energy is reflected towards the surface. The hydrophones of the streamer collects these reflections, or signals, which contain information about pressure changes in the water. Data collection from the streamers is commonly performed for several shots taken with increasing or decreasing distance between source and receiver (hydrophone), producing what is known as a seismic shot gather. An illustration of marine seismic acquisition is shown in Figure 2.1

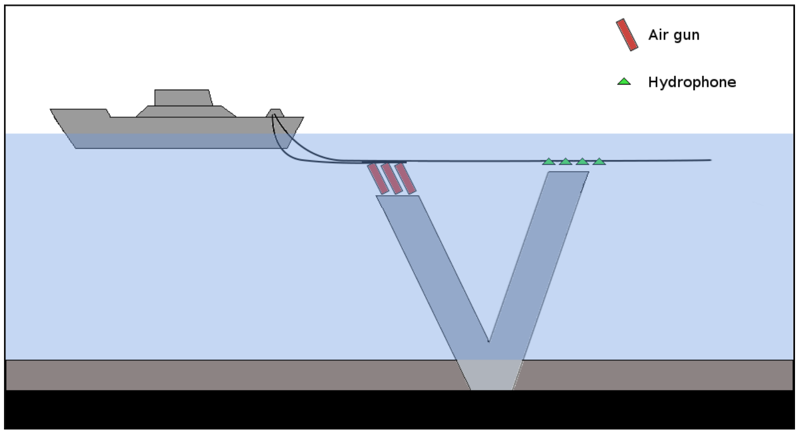

Figure 2.1: Seismic acquisition. The vessel in the Figure tows a streamer and air guns. The reflections of the pressure wave off interfaces in the sub surface are illustrated

2.2 Application Details

Our application, SplFd2dmod, which is a part of StatoilHydro's *Seismic Processing Library* (SPL), simulates seismic shot data by means of finite difference modeling. The data on which the application bases its modeling, are synthetic velocity and density models of the sub surface which describe the acoustic impedance (interfaces/contrast) in the different areas of the model. SplFd2dmod computes a sub surface pressure field by propagating a pressure wave (shot) through the modeled area characterized by the aforementioned velocity and density models. Figure 2.2 is an example output of modeling a single shot gather, and Figure 2.3 is an example of several shot gathers. Contrast in both of these Figures has been enhanced by discarding amplitudes outside the range $[-0.0001, 0.0001]$.

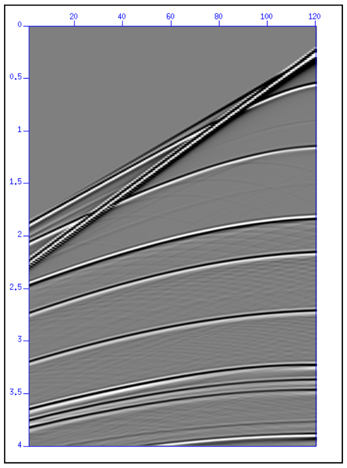

Figure 2.2: Application output after modeling a shot

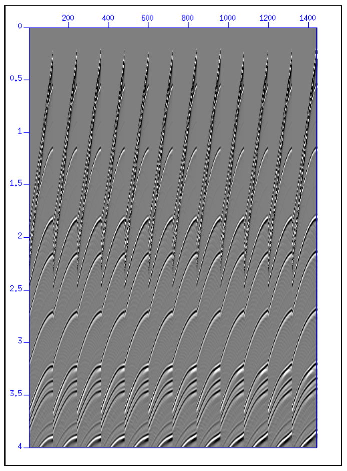

Figure 2.3: Shot gathers containing 12 modeled shots

The application has a single- and a multiprocessor (MPI) mode of execution. The multi processor mode is built in an SPMD fashion, meaning it is written as a single program working on multiple data elements. The data elements of the application are the seismic shot records. The application is parallelized by shot record, meaning each slave process on nodes of the computational cluster running it will process a single shot. The master process (rank 0) of the application works as an I/O server, with the intent of assuring atomic read/write access to files requested by the slave processes. This enforces consistency and minimizes file system pressure. FIO is the library component of SPL containing the I/O server.

The slave processes pick a seismic shot from a file containing all the available shots. The selection of shots is governed by a key-map file, keeping track of which shots are available and which are being/have been modeled. The key-map file is, as with the other input files, shared among all the processes.

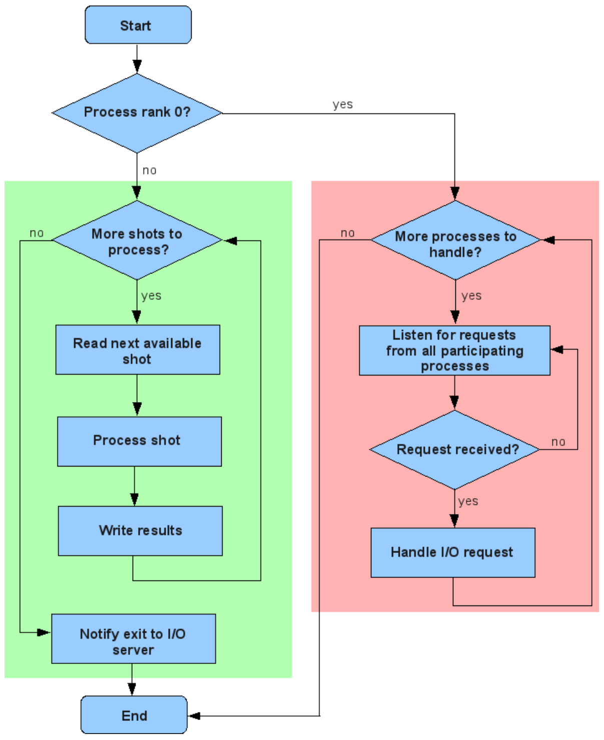

Figure 2.4: Flowchart for the application taken from [5]. The flow of the slave processes is shown in the green rectangle. The red rectangle shows the flow of the I/O server

Each slave process does a 2D acoustic finite difference modeling of a given shot, and outputs its result in a single shared file, which in the end will contain all of the modeled shots. When a slave process has finished modeling a shot, it will try to get a new one by reading the key-map file, looking for remaining shots still in need of modeling. If there are no more shots available, the process will exit.

A flowchart of the application can be seen in Figure 2.4. One of the new features of this revision of SplFd2dmod compared to the one in [5], is that finished shots are sorted according to their shot record numbers in the output file. The application in [5] did no such ordering, resulting in potentially different ordering of the final shot gathers depending on the order in which the slave processes finished their computations.

The component of the SPL library which is most important for our application is called Fd2d (see [7]). This is a high level library component for doing the 2D acoustic finite difference modeling central to SplFd2dmod.

Chapter 3

GPU Programming & FDM

In this Chapter, we will review various aspects of GPU programming, as well as introduce the Finite Difference Method (FDM), which is a key numerical method used in our application We will end this Chapter by looking at previous work done in the field of CUDA GPU computing related to FDM computations

3.1 GPU Programming

In this section we will examine what GPU programming, also referred to as General Purpose GPU (GPGPU) programming, is. In addition, we will explore some of the available programming languages related to this topic.

Before shading languages and GPU programming APIs, general purpose computations on the GPU required either low level assembly programming of the GPU or use of the programmable stages in the processing pipelines of graphics APIs such as OpenGL and Direct3D (DirectX). Using assembly level programming is a very time consuming process, and requires a great deal of expertise, making it unsuitable for most developers. Using the programmable stages of graphics pipelines is achieved by using shader languages. Although significantly more convenient than the low level approach, shading languages forces the developer to reformulate his/her problem to account for the inherent graphics processing intentions of the pipeline.

Graphics Pipeline

Shading languages such as NVIDIA's *Cg*, *The OpenGL Shading Language*(GLSL) and *The High Level Shading Language*(HLSL), are targeted at the programmable stages of graphics pipelines used in the Direct3D (DirectX) and OpenGL graphics APIs.

In graphics programming, the developer specifies geometry in a scene using vertices, lines and triangles. These definitions are in turn submitted to the graphics pipeline, which essentially translates the geometry to the screen with the intended lighting and coloring specified.

Graphics pipelines used in Direct3D and OpenGL contains several fixed function and programmable stages. Each stage in such a pipeline processes entities provided in a stream by the previous stage. The order in which these particular entities appear in the pipeline is as follows:

1. Vertices
   - Vertices are what the programmer uses to specify geometry. Each vertex can contain a position, color value, normal vector and texture coordinate.
2. Primitives
   - Primitives are e.g. points, lines and triangles. Primitives are formed from connected groups of vertices.
3. Fragments
   - Fragments are generated by rasterization, which is the process of breaking up primitives into a discrete grid. Fragments consists of a color value, position and a distance value from the scene camera.
4. Pixels
   - Pixels, or picture elements, are the final entities sent to screen. Color contributions from fragments are processed for each position in the framebuffer and combined into the final pixel value stored in that particular location.

A simplified overview of a graphics pipeline is given in [8], and can be seen in Figure 3.1.

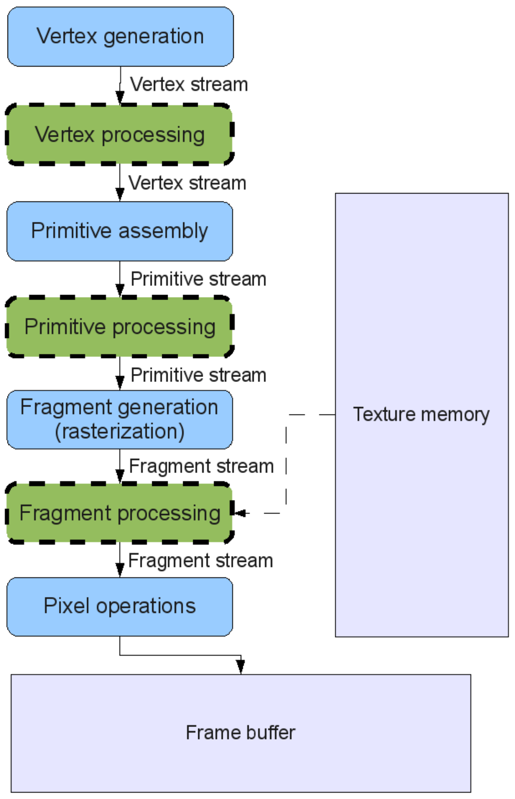

Figure 3.1: Simplified view of a graphics pipeline with indication of streams between the different stages as well as their memory accesses

The first stage, denoted *vertex generation*, collects the vertices in the geometry specified by the programmer and sends them in a stream to the next stage in the pipeline, denoted as *vertex processing*. The most significant operation performed in this stage is doing a transformation on the vertex position in order to project it from the world/scene space into screen space. The vertex processing stage is the first programmable stage in the pipeline, and thus allows the programmer to alter projection, color values, normals and texture coordinates for each incoming vertex.

After the vertex processing stage, vertices, along with information given by the programmer on their relationship, are being assembled into primitives, namely points, lines and triangles. Once primitives have been assembled, they are sent in a stream to the next stage for additional processing. This stage is referred to as the *primitive processing* stage.

Primitive assembly is then followed by the rasterization stage, where each primitive is broken down into fragments, which are discrete points in a grid corresponding to one pixel location in the final framebuffer. This stage, referred to in [8] as *fragment generation*, arranges the fragments from each primitive into a stream sent to the following *fragment processing* stage. This stage is programmable, and is in graphics intended for lighting the fragments, as well as applying textures.

The final stage in the pipeline, referred to as *pixel operations*, decides which fragments will be displayed in the final picture. Each fragment contains a depth value (Z value), which indicates how far the fragment is from the camera in the scene. When fragments overlap one pixel position, the closest of them is the one being stored in the framebuffer for final display. Blending, or combining the colors of several overlapping fragments, will occur if some visible fragments have some degree of transparency.

The programmable stages mentioned above, *vertex processing*, *primitive processing* and *fragment processing*, will be the basis for the shading languages we will review next.

Shading Languages

There are three major shading languages intended primarily for programming graphics shaders; *NVIDIA's C for Graphics*(Cg), *The OpenGL Shading Language*(GLSL) and *The High Level Shading Language*(HLSL).

HLSL, developed by Microsoft, is targeted at their Direct3D graphics API. The language was first deployed when DirectX 8 was launched. The capabilities of the language varies depending on what *shader model* specification is supported by the target GPU.

With the latest shader model 4, HLSL can be used to develop vertex-, geometry- and fragment-shaders at their respective stages in the pipeline. The first DirectX distribution supporting shader model 4 was DirectX 10.

GLSL is the OpenGL shading language. According to [9], the GLSL API was originally a set of extensions to OpenGL, but as of OpenGL 2.0, it became a built-in feature.

Programming model for GPGPU shader programming

When programming shaders for use with non-graphical computations, the usual steps are as follows:

- Draw geometry using the graphics API
- Make texture of input data grid
- Fetch previously submitted textures with input data and do computations
- Write results for each fragment in the solution grid
- Copy final results to a texture
- Retrieve results of computation by fetching texture

A 2D data grid can be represented by drawing a simple, appropriately sized square in OpenGL/Direct3D. After the square has been submitted to the graphics pipeline, first stage of interaction is in the vertex processor. At this stage, initial values for the corners of the solution grid can be set. At the rasterization stage when the area of the square is being broken into fragments, the color values of each fragments is set based on the previously specified color of the corner vertices. If specified, different values at the corners can be interpolated over the area of the square.

After the rasterization is completed, the fragment shader kernel is invoked. In this kernel, a previously stored texture can be read as input data for the computations. Kernels are invoked in parallel, and work on a single fragment, or point, in the solution grid. The complete result after the fragment shader stage is written to a texture.

If the programmer requires to do more computations on the results from a single pass through the pipeline, he simply replaces the previous input texture with the result texture from the fragment shader, and calls another draw operation to make another pass through the pipeline.

Once all computations have been completed, the resulting texture is fetched from GPU memory.

High-Level GPU programming languages

In addition to the shading languages mentioned above, there exists higher level languages which compile their source with shading languages as compile targets. These languages use more general structured programming syntax, minimizing the requirement of formulating the problems in graphics programming terms.

Some of the most established high level GPGPU languages are Brook, Sh, Microsoft's Accelerator, Peakstream and RapidMind. A quick review of these languages can be seen in [1].

3.2 CUDA

In this section, we will explore the recent advances in programming APIs for numerical computation on the GPU. Since 2006, APIs for such programming have evolved by eliminating the need for an underlying graphics API such as Direct3D and OpenGL. These APIs are now interfacing directly with the GPUs, allowing APIs expressed in more familiar terms to the general purpose developer.

The Compute Unified Device Architecture (CUDA) is a parallel processing framework developed by NVIDIA for their GPUs. CUDA has an associated programming environment which enables application programmers to utilize the power in these GPUs in order to offload compute intensive, data-parallel computations.

For the purposes of this thesis, CUDA will be the only high level GPU programming API reviewed here. There are other alternatives such as the OpenCL heterogeneous many-core API, and the AMD Stream SDK. The reasons for using CUDA in this thesis, are:

1. During this project, most of the hardware available has NVIDIA CUDA enabled GPUs. Using the AMD Stream SDK will not be an alternative given this fact.

2. The CUDA architecture is quite mature compared to OpenCL. First editions of CUDA were ready in 2006 (see [10]), and its most recently released version is 2.1 (04.08.09). OpenCL was first conceived during the first half of 2008, and the first release came in December 2008 (see [11]).

Figure 3.2 shows part of the internals of the NVIDIA GTX200 architecture. This architecture is used in e.g. the NVIDIA GeForce 280GTX and 260GTX. What the Figure more specifically shows, is what is known as a Thread Processing Cluster (TPC). There are a total of 10 such clusters on the GTX200 chip. The architecture depicted is similar to the one of the NVIDIA Tesla T10 series, which is the architecture of the Tesla C1060 card. The C1060 is a pure GPU card with no video output, and is available as a single stand alone card, or as part of the Tesla S1070 rack unit, which holds 4 of them. Additional information on the GTX200 architecture can be found in [12] and [13]

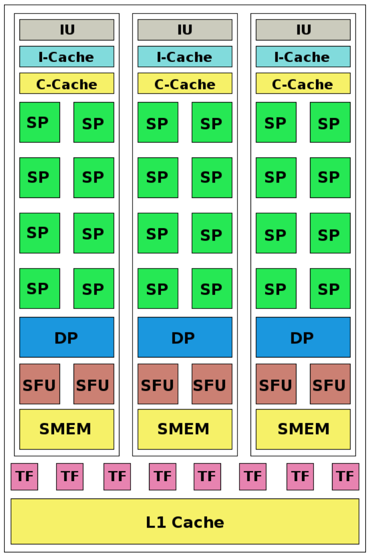

Figure 3.2: TPC of the GTX200 architecture. A GeForce GTX280 GPU card contains a total of 10 TPCs. (With permission from NVIDIA)

The TPC in Figure 3.2 contains an L1 texture cache available to all 3 Streaming Multiprocessors (SM). The *TF* units are units which perform texture filtering and addressing operations concerning texture memory. Each of the 3 SMs in the TPC consist of the following components. The Streaming processors (SP) perform single precision floating point calculations in compliance with the IEEE 754 standard. New to this architecture is the addition of the *DP* stream processing unit. This processing unit performs double precision floating point calculations, and as the Figure shows, there are only 1 of these per SM. There are in addition 2 Special Function Units (*SFU*) per SM, used for operations such as *sin* and *cosin*. The Shared Memory (*SMEM*) region of the SM is a fast, on-chip low latency memory which can be compared to an L1 cache of a traditional CPU. The *I-Cache* is the instruction cache, while the *C-cache* is a read only data cache for constant memory (see 3.4). *IU* is a unit which distributes instructions to the DP, SP, and SFUs of the SM.

The *Compute Capability* of a CUDA enabled GPU describes capacities and capabilities of classes of devices. Devices such as the GeForce GTX280 and the Tesla C1060 have compute capability 1.3, whose features can be seen in Appendix A.1.1 of [10]

The CUDA Programming Model

C for CUDA is the programming language used to program CUDA enabled devices. The language is based on the ANSI C programming language, with some CUDA specific extensions. When developing a CUDA program, the source code separates code segments which will be run on the host (CPU) and the device (GPU). Device code is written as *kernels*, which is functions that correspond to a single thread run on a *streaming multiprocessor*(SM) (depicted in Figure 3.2). When such a kernel function is invoked, the programmer specifies how many instances/threads of the kernel will be executed on the GPU in terms of number of *blocks* of *threads*. The blocks in this execution configuration are arranged in a two-dimensional *grid*, where each block is a three-dimensional array of threads. The relationship between grids, blocks and threads is explained in [10] and [14]. Figure 3.3 shows an arrangement of a grid with dimensionality 3x3, containing blocks of 3x3x1 threads, totaling 81 threads.

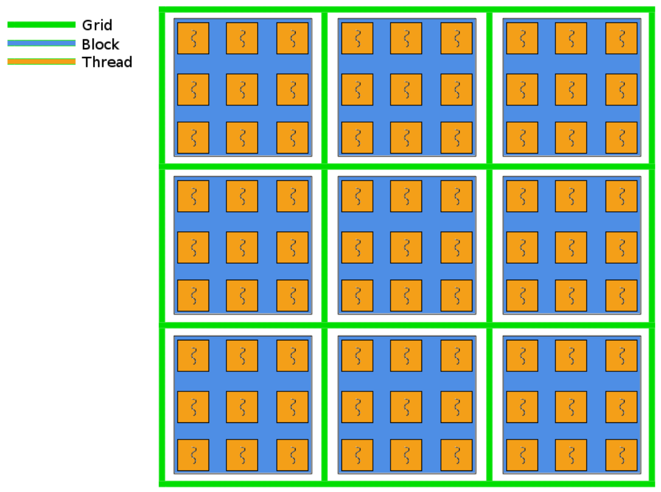

Figure 3.3: Thread organization on GPU in terms of grids and blocks

As Chapter 3 of [14] explains, the SMs on the GPU are the execution units which are assigned the blocks of threads specified upon kernel invocation. [14] explains the capacities of the SMs in the GeForce 8800 GTX card. Each SM in those cards hold a maximum capacity of 8 active thread blocks each, with a maximum total of 768 concurrent threads. Once the blocks are assigned to the SMs, they are further divided into *warps*, which are groups of 32 threads each. warps are executed in what's known as a *Single Instruction Multiple Thread*(SIMT) manner. SIMT can be thought of as a similar concept as *Single Instruction Multiple Data*(SIMD) on CPUs, where a single instruction is executed on several data elements at the same time. With SIMT, this means instructions of each of the threads in a warp are executed concurrently. The SMs schedule warps from different blocks so that whenever a warp requires a long latency read operation from memory, the SM puts that warp on hold and selects another for execution, providing maximum efficiency in the execution and hiding long latency operations such as global memory reads.

The CUDA Memory Model

A CUDA enabled device has a total of six different memory types, distributed as on-chip and off-chip memories. Figure 3.4 shows these types as based on the indications in [15] and [10]. The Figure shows a device containing an off-chip DRAM region of global memory and an SM containing a block of threads and the associated on-chip memory types. The characteristics of the memory types depicted can be seen in *Table 3.1*. The columns of the Table indicates the following:

- *Location*
  Indicates where on the device the memory resides, i.e. on-chip or off-chip (DRAM)
- *Cached*
  Indicates whether the memory is accessible through an on-chip data cache.
- *Scope*
  Indicates who can access the memory type in terms of per-grid, per-block and per-thread.
- *Access*
  Indicates whether memory type has read, write or read/write access.

Memory Type	**Location**	**Cached**	**Scope**	**Access**
Global Memory	off-chip	No	Grid	Read/Write
Constant Memory	off-chip	Yes	Grid	Read
Local Memory	off-chip	No	Thread	Read/Write
Registers	on-chip	N/A	Thread	Read/Write
Shared Memory	on-chip	N/A	Block	Read/Write
Texture Memory	off-chip	Yes	Grid	Read

Table 3.1: Memory type characteristics of a CUDA enabled device

The non-array variables declared within a kernel function are allocated by the run-time environment of CUDA in the per-thread registers. The off-chip local memory holds according to [10] the arrays declared within kernel functions, as well as other structures too big to be stored in the register of the current thread.

The shared memory type is a read/write on-chip cache, and allows low-latency sharing of data among threads within the same block. Shared memory can be used as a buffer for communication messages between threads within the same block. The way this works, is that threads write their messages to the shared memory, then synchronizes through a barrier, and finally reads back the messages from the shared memory.

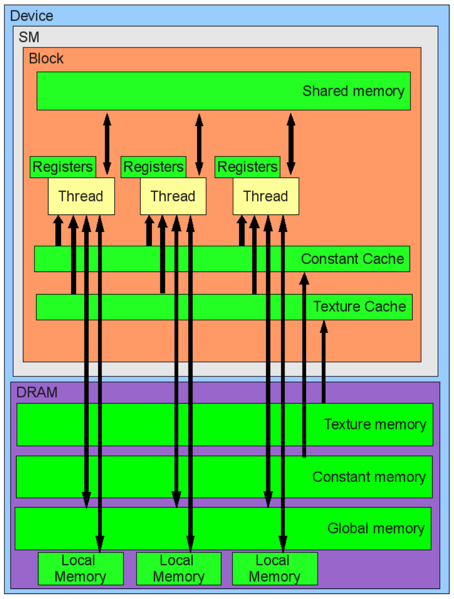

Figure 3.4: CUDA Memory Layout, Showing the different memory types

Texture memory is a read only portion of the global DRAM, and has an associated on-chip cache as depicted in Figure 3.4. [10] indicates that this type of memory is optimized for 2D spatial locality, meaning it is efficient for reading 2D structures. Constant memory is similarly to texture memory, a read-only memory type which has an on-chip cache. Memory access patterns has a great impact on performance. The high degree of control given by the CUDA memory hierarchy makes it easier to optimize this aspect of a GPU application.

3.3 The Finite Difference Method (FDM)

Solving Partial Differential Equations (PDEs) requires that we discretize the problem before implementing a solver. The terms in these equations involve partial differentials, which we can approximate with numerical differentiation by means of the Finite Difference Method (FDM) (see e.g. Chapter 8.6 of [16]).

If we want to approximate a first derivative of a function $f(x)$ with respect to x, e.g.

$$f^{'}(x) = \frac{df(x)}{dx} \tag{3.1}$$

we can for use the following approximations

$$f^{'}(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{3.2}$$

$$f^{'}(x) \approx \frac{f(x) - f(x-\Delta x)}{\Delta x} \tag{3.3}$$

$$f^{'}(x) \approx \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} \tag{3.4}$$

where Δx is the distance between successive discrete points in the function, 3.2 is the forward difference approximation, 3.3 is the backward difference approximation and 3.4 is the most accurate of the 3; the centered difference approximation. For the second derivative of a function $f(x)$ with respect to x, e.g.

$$f^{''}(x) = \frac{d^2 f(x)}{dx^2} \tag{3.5}$$

we get a centered difference approximation like this (equation 3.6):

$$f^{''}(x) \approx \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{(\Delta x)^2} \tag{3.6}$$

The accuracy of the previously mentioned approximations can be further improved by considering additional off center points, e.g. by adding $x \pm 2\Delta x, x \pm 3\Delta x ...$ to the equations and appropriately adjusting the denominator to account for the new expanded range of the approximation. An example of adding an additional point to either side of the first derivative centered approximation looks like this:

$$f^{'}(x) \approx \frac{(f(x+2\Delta x) + f(x+\Delta x)) - (f(x-2\Delta x) + f(x-\Delta x))}{4\Delta x} \tag{3.7}$$

For a two dimensional function $f(x,z)$, we can approximate its first order partial derivative with respect to x, $\frac{\partial f(x,z)}{\partial x}$ with e.g. a centered difference approximation like the one in Equation 3.7 like this:

$$\partial_x f(x,z) \approx \frac{(f(x+2\Delta x,z)+f(x+\Delta x,z))-(f(x-2\Delta x,z)+f(x-\Delta x,z))}{4\Delta x} \tag{3.8}$$

Equation 3.8 can be visualized as a 1 dimensional stencil over the 2 dimensional solution area, depicted in Figure 3.5

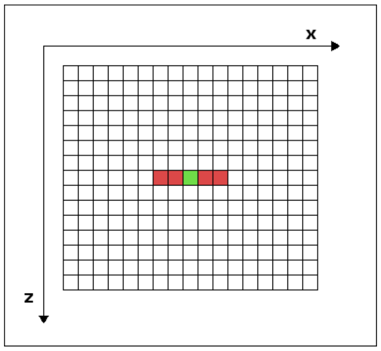

Figure 3.5: Visualization of the centered difference approximation in Equation 3.8. Point computed is in green, while evaluated points are in red

3.4 Related Work on CUDA GPU Offloading

Chapter 38 of [3] presents work done by *CGGVeritas* on implementing CUDA based GPU computational offloading for a seismic imaging application. The algorithm of the application, referred to as SRMIP, performs seismic migration. Central to this algorithm is acoustic wave propagation, which is performed by a finite-differencing algorithm in the frequency domain. Performance comparisons of the developed GPU offloading implementation in this work show results of 8-15X speedup with a NVIDIA G80 based GPU (Quadro FX 5600) over an optimized single threaded version of the original CPU based implementation.

[17] is an article covering a 3D finite differencing computation done with CUDA enabled hardware. The article covers two implementations, one for just doing computations with a 3D finite differencing stencil, and the other

for computing a finite difference approximated wave equation. The wave equation discretization is also implemented with support for multiple GPUs, where the domain of the computation is partitioned among the different GPUs, allowing for larger problem sizes. In the multi-GPU version, the decomposition of the domain distributes e.g. for the 2 GPU case, each of the GPUs gets half the domain + a layer of points overlapping the other half. The multi-GPU implementation of the article shows a throughput for e.g. a 4 GPU (Tesla S1070) computing a volume of dimensions $480 \times 480 \times 800$, of 11845.9 million points per second.

[4] is an article which describes the implementation of the Navier-Stokes equations for incompressible fluids. The discretization made of the equations involves centered second order finite difference approximations. The implementation in this article uses double precision calculations on a NVIDIA Quadro FX5800 4GB card, and shows a speedup of 8X compared to a multi-threaded Fortran code running on an 8 core dual socket Intel Xeon E5420 CPU. This kind of speedup is especially impressive considering the fact that there are only 1 double precision stream processor on each of the 30 streaming multiprocessors of the GPU.

Chapter 4

Application Analysis and Optimization

In this Chapter, we will begin by profiling our application, identifying the areas in need of GPU computational offloading. Once these areas have been identified, we will describe their workings before proceeding with implementing GPU replacements for them.

We will go through the different GPU implementations step by step, continually adding new optimizations, explaining motivations for them, as well as implementation details. We end this Chapter by describing the details of adding multiple GPU support to our application.

4.1 Profiling and Analysis

While the core computational areas of this application is known to us beforehand, profiling the application will allow us to easily identify the source code areas in need of GPU offloading. We will use the Valgrind[18] based tool CallGrind, which profiles the call paths for an application, as well as the duration for each of the called functions.

The results obtained by running the single processor version of our application through the CallGrind tool can be seen in Figure 4.1. Here it is seen that the subroutine *Fd2dModfd* is the main contributor of the computational overhead of the application. Looking further down, we see that the *Fd2dTimestep* subroutine is being called 13334 times. For each call to this subroutine, four different subroutines (*DerDifxfw*, *DerDifyfw*, *DerDifxba*,

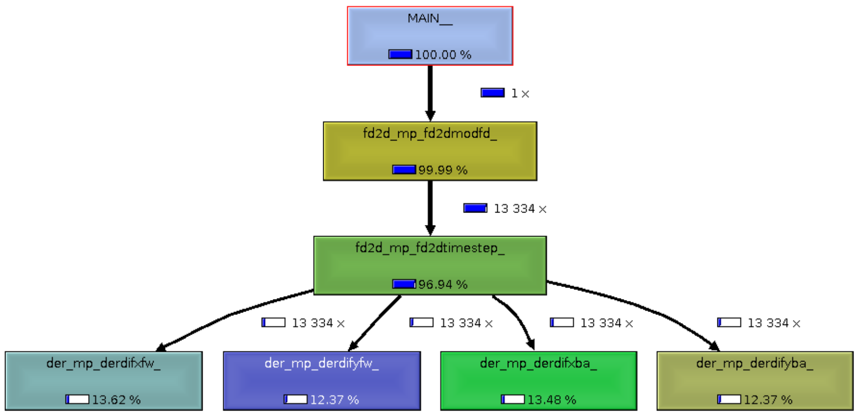

Figure 4.1: Function call path of our application obtained from the CallGrind tool

DerDifyba) are called.

The *Fd2dTimestep* routine performs a single time step of the finite difference method applied to the 2D acoustic equations of motion,

$$\rho * \partial_t v_x = \partial_x \sigma \tag{4.1}$$

$$\rho * \partial_t v_z = \partial_z \sigma \tag{4.2}$$

and the constitutive relation,

$$\partial_t \sigma = K * \partial_i v_i + \partial_t I \tag{4.3}$$

Einstein's summation convention i is used, where $i = (x, z)$, the $*$ operator denotes time convolution, and $I = I(t, x, z)$ is the source of injection type. The equations above are performed on a regular 2D grid, and solves for the stress σ components. The equations are relating the particle velocities v_x and v_z, the density ρ and the bulk modulus K. x and z in the equations denote the horizontal and vertical axis respectively, and t denotes the time. The first order time derivatives in the previous equations (4.1, 4.2, 4.3) are approximated with backward and forward finite differences,

$$\partial_t f_{n-\frac{1}{2}} = \frac{f_n - f_{n-1}}{\Delta t} \tag{4.4}$$

, and

$$\partial_t f_{n+\frac{1}{2}} = \frac{f_{n+1} - f_n}{\Delta t} \tag{4.5}$$

As we can see in the two previous equations, the $\partial_t f$ are evaluated half a grid point behind and in front of the current grid point respectively. In the same 3 PDE equations, the spatial derivatives are approximated using an 8th order centered difference approximation (see [19]), e.g. in the x direction as,

$$d_x^- \sigma(n, k - \frac{1}{2}, l) = \frac{1}{\Delta x} \sum_{m=1}^{8} \alpha_m [\sigma(n, k + (m-1), l) - \sigma(n, k - m, l)] \tag{4.6}$$

and

$$d_x^+ \sigma(n, k + \frac{1}{2}, l) = \frac{1}{\Delta x} \sum_{m=1}^{8} \alpha_m [\sigma(n, k + m, l) - \sigma(n, k - (m-1), l)] \tag{4.7}$$

As with the temporal approximations seen in Equations 4.4 and 4.5, we see that the resulting stress is evaluated half a grid point behind or in front of the current grid point respectively (seen as $k - \frac{1}{2}$ and $k + \frac{1}{2}$ respectively). This half point spacing shows that our application uses a staggered grid both in time and in space (see Appendix C in [6]). The difference approximations in Equations 4.6 and 4.7 are defined with differentiator coefficients α_m, which are pre-computed values. The values of these coefficients are computed by matching the Fourier spectrum of the approximated differentiators with the Fourier spectrum of a perfect differentiator (see [19]).

Equations 4.6 and 4.7 for the x-direction, and similar equations for the z-direction, are located in the *DerDixxxx* routines we observed in the call graph of Figure 4.1. In the original application, the order of these differentiators is determined by how many points before and after the point being computed are considered, and can be selected in the range $[1-8]$, where the 8th order gives the most accurate result. In our implementations, as shown in Equations 4.6 and 4.7, we will only consider the 8th order differentiators.

Reflections of wave energy from the edges of the modeling domain is absorbed using the PML absorbing layer detailed in [20], [21] and [22].

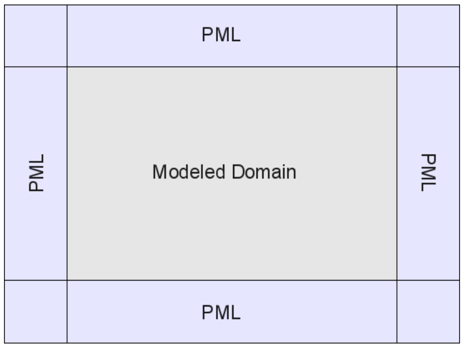

Figure 4.2: Problem domain with added PML layers

As there are 4 differentiations being performed for each time step, the PML absorption is done on the borders in the direction of the differentiation for each of these. The problem domain with the added PML boundary regions can be seen in Figure 4.2.

Figure 4.3 shows the central work flow of the modeling routine *Fd2dModfd*. All steps contained within the enclosing blue rectangle in the center of the Figure are part of the *Fd2dTimestep* routine.

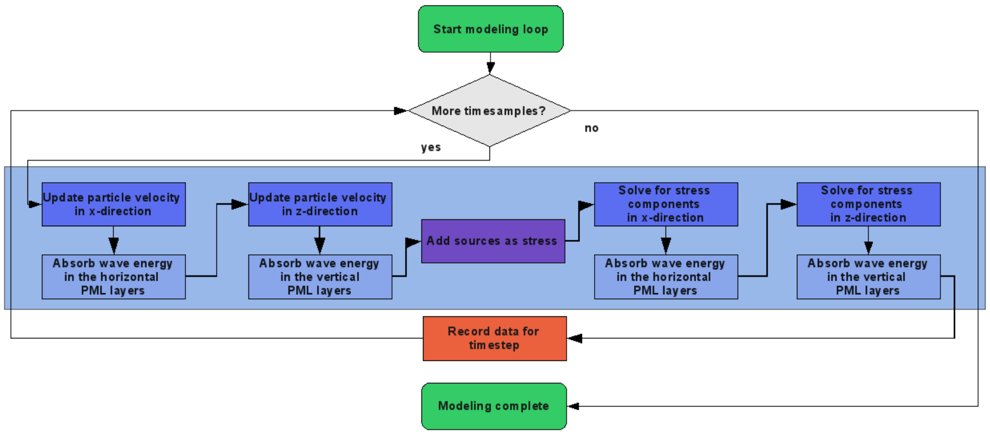
Start modeling loop
More timesamples?
no
yes
Update particle velocity in x-direction
Absorb wave energy in the horizontal PML layers
Update particle velocity in z-direction
Absorb wave energy in the vertical PML layers
Add sources as stress
Solve for stress components in x-direction
Absorb wave energy in the horizontal PML layers
Solve for stress components in z-direction
Absorb wave energy in the vertical PML layers
Record data for timestep
Modeling complete

Figure 4.3: Modeling loop of the Fd2dModfd routine

The dimensions of the modeled area are N_x in the horizontal direction and N_z in the vertical direction. The width of the PML layers seen in Figure 4.2 is denoted N_{pml}, making the dimensions of the 2 layers in the horizontal dimension $[N_{pml}]$x$[N_z]$, and the 2 layer in the vertical direction $[N_x]$x$[N_{pml}]$.

Finally, the order of the finite difference stencil is denoted N_{hl}. The four differentiator functions in *Fd2dTimestep* do finite differencing within the total modeled domain, including the PML layers (i.e. the total area shown in Figure 4.2). This makes the differentiators span an area of $[N_x + N_{pml}]$x$[N_z + N_{pml}]$. The PML layer absorption steps of *Fd2dTimestep* only computes values that span the dimensions of the respective layers.

4.2 Implementing GPU Offloading

The source code of SplFd2dmod is mostly written in the Fortran 90 standard, with some older components written in Fortran 77. Since CUDA is a language extending ANSI C, as mentioned in Chapter 2, our implementation uses mixed programming, in which we use a Fortran 2003 feature which enables writing modules with interface declarations pointing to the CUDA source functions.

Our first approach is moving the computations performed for each time step to the GPU, i.e. porting the functionality of the *Fd2dTimestep* routine depicted in 4.3. The first step in order to achieve this is moving the relevant data values for the computations to the GPU. We have made Fortran interface declarations for CUDAs memory management functions, namely *cudaMalloc*, *cudaMemcpy* and *cudaFree*, which do allocation, copying and de-allocation, respectively. These interfaces are called within the *Fd2dModfd* routine, allocating and copying all the relevant values to the GPU device, and once modeling completes, de-allocates the associated memory. Once the necessary data values are moved to the GPU device, we proceed by implementing CUDA kernel functions for each of the *Fd2dTimestep* steps in Figure 4.3.

Common to all of the following implementations, the modeling routine, *Fd2dModfd*, downloads the relevant stress (σ) values from the device after each GPU invocation of the steps in the *Fd2dTimestep*. The dimension of the downloaded stress values is equivalent to a single line of the modeled area, i.e. N_x. Another common point is that all kernels have the same execution configuration, that is, they are all executed in a grid accommodating the total size of the model area ($[N_x + N_{pml}]$x$[N_z + N_{pml}]$), with thread blocks of dimensions 16x16. Figure 4.4 shows this configuration superimposed on the Figure of the total area (4.2). Finally, the "Add sources as stress" part of Figure 4.3, is implemented with the same kernel across the different implementations.

The kernel is very small, and only performs a simple vector-matrix addition.

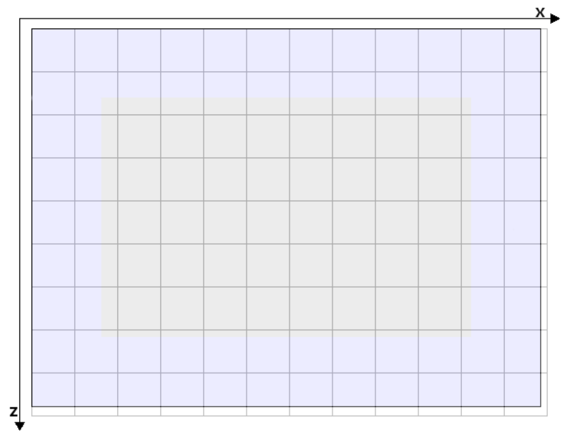

Figure 4.4: Execution configuration superimposed on the total working area of the model (compare with Figure 4.2)

4.2.1 First Implementation

In this implementation, we partition the steps in Figure 4.3 into 7 separate kernels as shown in Figure 4.5. The order in which these kernels are invoked is the same as in 4.3, i.e. "Kernel 1 → Kernel 6 → Kernel 2 → Kernel 7 → Kernel 5 → Kernel 3 → Kernel 6 → Kernel 4 → Kernel 7 ". The kernel invocation count for one time step of the *Fd2dTimestep* routine becomes 9 for this first implementation.

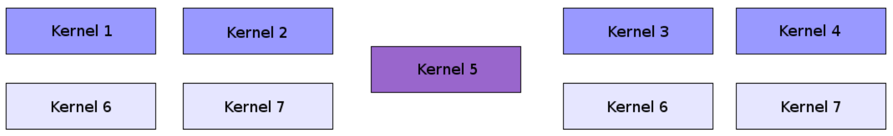

Figure 4.5: Logical kernel assignment for the first implementation of the *Fd2dTimestep* routine. See Figure 4.3 for reference

The tasks performed by the different kernels in Figure 4.5 can be summarized as follows:

- Differentiator Kernels

1. Kernels 1 and 2 are solving for the velocity (v) components in the horizontal and vertical directions respectively. The equation form of these operations is equivalent to that of Equation 4.7 expanded in both spatial directions, with σ substituted with v.
2. Kernels 3 and 4 are solving for the stress (σ) components in the horizontal and vertical directions respectively. The equation form of these operations is equivalent to that of Equation 4.6 expanded in both spatial directions.

- PML Kernels

1. Kernel 6 absorbs wave energy reflected off the horizontal borders resulting from the computations in kernels 1 and 3
2. Kernel 7 absorbs wave energy reflected off the vertical borders resulting from the computations in kernels 2 and 4

Looking back at the stencil shown in Figure 3.5, the 8th order centered differentiators of Kernels 1-4 have a similar shape, with the exception that there are a total of 16 contiguous values in either direction being evaluated for a point located half a grid point in front or behind the center of the stencil. Keeping in mind that the domain is a staggered grid, this means that the computed output point is located in a different, overlapping grid shifted half a grid point off the grid with the evaluated (old) values.

Figure 4.6 shows the 3 different paths threads within the horizontally differentiating kernels take. The Figure shows the complete domain of the model, including PML layers. Considering the stencil of the horizontally differentiating kernels (1,3), if a kernel thread is within region 2 of the Figure, all points in the stencil are used in computing the new point. If the thread is located within regions 1 or 3, some of the evaluated points of the stencil will be outside the domain. These points are treated as 0 in the computations, effectively shrinking the half of the stencil facing the border of the domain. The vertically differentiating kernels work in the exact same way, only in the z direction.

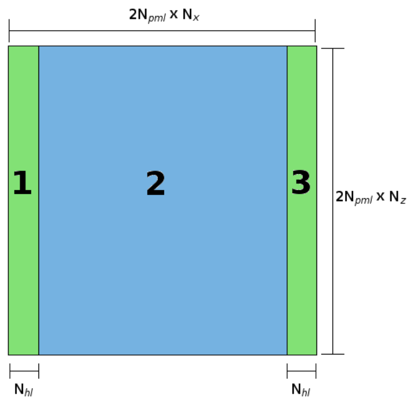

Figure 4.6: Horizontal differentiator regions of computation

All kernels of this implementation use only the non-cached global memory of the GPU device see Figure 3.4).

4.2.2 Second Implementation

Now that a working implementation has been achieved, its time to do optimizations. The related finite differencing done in [17] uses primarily the on-chip per-SM shared memory of the GPU. According to Chapter 5 of [10], reads from shared memory is a lot faster than global memory reads. According to [15] and [10], reading memory from the global memory space has in addition to the 4 clock cycles required to issue the read instruction for a warp, an added latency of about 400-600 clock cycles. Once shared memory is loaded into its memory space on the SM, [10] states that given no bank conflicts among threads of the same warp, accessing shared memory can be as fast as accessing registers, which generally cost no more than the 4 clock cycles to issue the read instruction.

Using shared memory in this implementation should prove beneficial due to the fact that several of the memory locations read by threads spaced within the half-length of the differentiator stencils read several common values.

We denote the differentiator stencil half length (order), N_{hl}, and the width and height of a thread block N_b. We prepare a 2D shared memory area of size $[2N_{hl}+N_b]\times[N_b]$ for the horizontal, and $[N_b]\times[2N_{hl}+N_b]$ for the vertical differentiator kernels. The threads of a thread block is logically placed at the center region of the shared memory area. Each thread reads a single value from its calculated location in the problem domain into its relative location in the shared memory region. For the horizontal differentiator kernels, threads which are at a distance $d <= N_{hl}$ from the left border, reads values located N_{hl} points to the left, and $N_{hl} + N_b$ points to the right into corresponding locations in shared memory. The same is done in the vertical direction with threads located N_{hl} points below the upper border. The loaded shared memory regions for blocks of both the horizontal and vertical differentiator kernels is depicted in Figures 4.7 and 4.8, respectively. Finally, the differentiator coefficient values (α) are loaded into a 1D shared memory area by the first N_{hl} threads of the thread block. At the end of all the preceding shared memory load operations, all kernels within the same block are synchronized, ensuring that all values are loaded beyond that point.

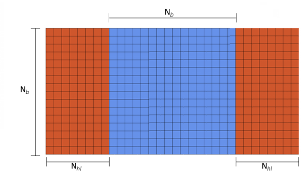

Figure 4.7: Shared memory region for thread blocks in the horizontal differentiators

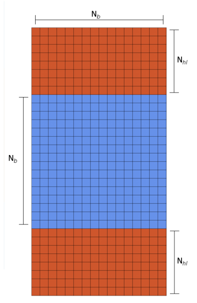

Figure 4.8: Shared memory region for thread blocks in the vertical differentiators

In the first implementation, the number of kernel invocations is 9. Since all threads of each of the kernels in that implementation is mapped to the same problem domain, more of the kernels could be combined, minimizing the number of invocations per time step. Looking back at the computational steps of each time step in Figure 4.3, we note that the PML operations after each differentiation can be combined with the preceding differentiator. The only requirement for doing this is adding conditionals to the differentiator kernels. For the horizontal differentiators, these conditionals would check whether the current thread of the kernel resides within the boundaries of either the left or the right PML layer, and perform the PML operations accordingly. An equivalent approach can be made for the vertical differentiators. By doing this, we reduce the kernel invocation count by 4, which makes the new total 5. The five kernels and their scope compared to Figure 4.3, can be seen in Figure 4.9.

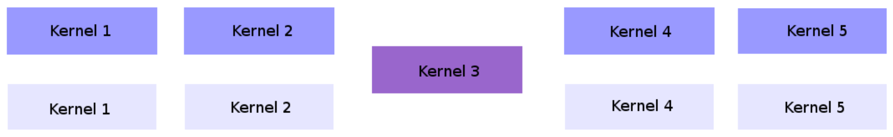

Figure 4.9: Logical kernel assignment for the second implementation of the *Fd2dTimestep* routine. See Figure 4.3 for reference

The whole modeling routine depicted in 4.3 performs memory transfers from the GPU to the host for every time step. We can optimize these transfers by using page locked host memory. What this means is that the page in host RAM containing the memory target of the transfer, is pinned to a fixed position, ensuring that the host operating system is unable to move it around. Doing this speeds up memory transfers from the GPU as the transfers are performed using Direct Memory Access (DMA), circumventing the CPU and thus lowering the transfer latency (see [23])

Summarizing the optimizations made in this second implementation, we have a total of 5 kernels, 4 of which perform combined differentiation and PML absorption. The differentiators uses shared memory for the stencil values, as well as the differentiator coefficients (σ). The PML calculations still only perform global memory read/writes. Finally, page allocated memory for the per time step GPU $\rightarrow$ host transfers has been employed.

4.2.3 Third Implementation

In this implementation, we further reduce the kernel invocation count by combining more of the steps in 4.3. All of the operations to the left of *add sources* are combined, and similarly, all operations at the right are combined. The total number of kernels now becomes 3, which can be seen in Figure 4.10. Both of these differentiator/PML kernels are now performing operations in both directions of the domain, doing the equivalent of Equation 4.7 for the kernel on the left (Kernel 1), and 4.6 for the kernel on the right (Kernel 3).

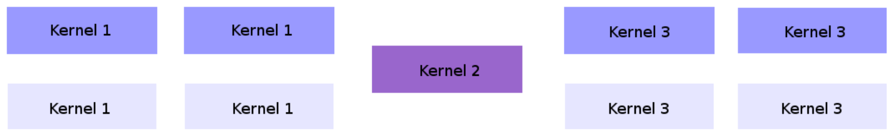

Figure 4.10: Logical kernel assignment for the third implementation of the *Fd2dTimestep* routine. See Figure 4.3 for reference

While the previous implementation used shared memory regions of $N_b \times (N_b + 2N_{hl})$ elements each. For the combined kernels of this implementation, the shared memory regions would have to be further expanded to dimension $(N_b+2N_{hl})\times(N_b+2N_{hl})$ in order to accommodate all necessary values of both directions. Another problem here is that the left kernel solves for the velocity (v_x) components in horizontal direction, followed by the separate velocity (v_z) components in the vertical direction. This would complicate shared memory usage by either needing to prepare two separate shared memory regions for each of the velocity fields, or by re-loading new values into shared memory between differentiations. Both cases would also further complicate thread indexing, and also requiring added registers and control flow.

Reverting back to global memory is not really an option in this implementation, since we know that reads from the global memory space is expensive, and that threads within a block tends to read several common values, something that would benefit from some kind of data caching. Looking at the architectural layout in 3.2, we see that the L1 texture cache looks promising for use with the differentiators. The texture memory, according to Chapter 5 of [10], is considerably faster than global memory reads, since it only needs to access the global memory on cache misses, i.e. when a thread tries to get a value not currently in the cache. It is also indicated that texture memory reads have less constraints on the memory access pattern compared to global and constant memory.

We implement texture memory with the two differentiator/PML kernels of this implementation. The way we do this, is to create 3 1D texture (for σ, v_x and v_z) with dimensions equal to the total area of the problem domain, i.e. $(2N_{pml} + N_x) \times (2N_{pml} + N_z)$. 1D textures is a natural choice for this implementation due to the fact that the memory areas of e.g. σ and v_x, v_z is located in linear memory where logical 2D addressing in the problem domain is stated as e.g. $\sigma[x][z] = \sigma[z \times (2N_{pml} + N_z) + x]$. Reads from the textures are performed through the CUDA API function, which for e.g. fetches in σ

looks like: "$\sigma[index] = tex1Dfetch(\ texture\_\sigma,\ index\ )$".

Summarizing the changes made in this implementation, we have joined Kernels 1-2 and Kernels 4-5 of the previous implementation (see Figure 4.9, and used texture memory for the evaluated values of the differentiator stencils. The differentiator coefficients remain, as in the previous implementation, in shared memory, since there are only 2 1D arrays of coefficients for each of the two kernels, each with dimension N_{hl} (i.e. 8).

4.3 Multi GPU

SplFd2dmod is, as mentioned in Chapter 2, MPI enabled. Since the framework for parallelism is already implemented in this manner, we have added a feature which enables the slave processes spawned in MPI mode to use GPU computational offloading if there is an available GPU for this process. The way this works is by passing an argument to the application, *gpn*, indicating how many GPU devices are available per node/workstation. The first *gpn* processes of each node runs the GPU version, while the remaining processes for that node runs regular CPU computations. By allowing a hybrid solution like this, resource utilization is effectively maximized. The updated flow diagram for the application can be seen in Figure 4.11. A requirement for this feature is that each node the application uses have the same fixed number of GPU compute devices. We have implemented this feature for all the 3 different GPU enabled implementations.

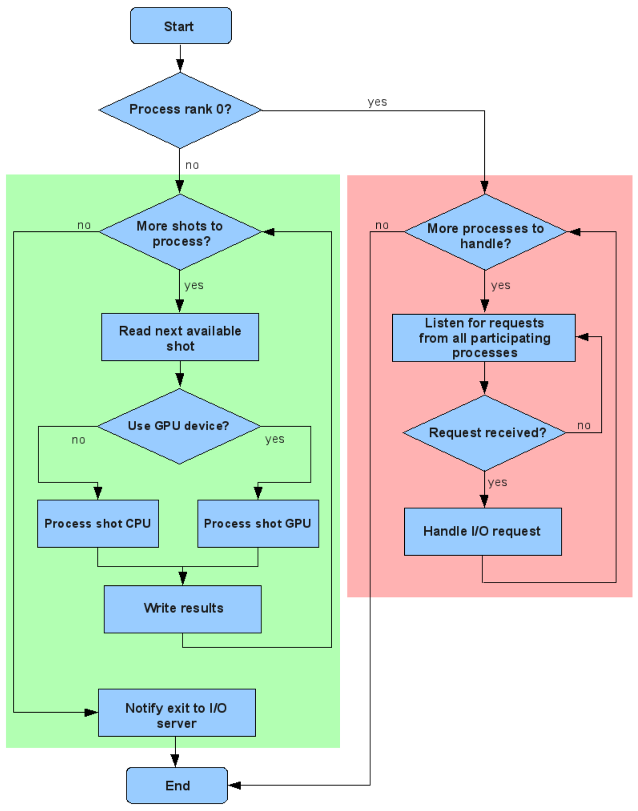

Figure 4.11: Updated work flow for hybrid CPU/GPU implementation

Chapter 5

Results & Discussion

In this Chapter we will compare the performance of our original application and the various optimized GPU versions discussed in the previous Chapter. The comparisons will be run on two different workstations, and we will end this Chapter with a discussion of the results obtained.

5.1 Methodology

For our benchmarking, time will be the metric used. Also, relative speedup with respect to the original application will be measured for each of the GPU versions mentioned in the previous Chapter. The speedup is calculated as follows:

$$S = \frac{t_{orig}}{t_{gpu}} \tag{5.1}$$

Where S is the speedup factor, t_{orig} is the run time of the original application, and t_gpu is the run time of the application with GPU offloading.

For each benchmark, we present a performance graph based on the average run time for a specific configuration over the course of 4 samples/runs. This average is calculated with the following equation:

$$t_{average} = \frac{1}{n}\sum_{i=1}^{n} t_i \tag{5.2}$$

The timings done in the tests are performed for the core computational kernel of the the *Fd2dModfd* routine, so data preparations are not included in the measurements despite the fact that they only incur an additional run time of a few seconds.

5.2 Test Systems

Our benchmarks will be obtained from two of the workstations available at the High Performance Computing Lab (HPC-Lab) run by Dr. Elster at the Department of Computer and Information Science (IDI). The hardware characteristics of these workstations are listed in Table 5.1.

System	CPU	RAM	Storage	GPU
HPC lab 1	Intel Core2 Quad Q9550 @2.83GHz	4GB	41GB	NVIDIA GeForce GTX280 NVIDIA Tesla C1060
HPC lab 2	Intel Core I7 965 @3.20GHz	12GB	1.4TB	NVIDIA Tesla S1070 (4X NVIDIA Tesla C1060)

Table 5.1: Hardware Characteristics

The operating system used for the two workstations in Table 5.1, is a 64 bit (x86_64) Ubuntu 9.04 distribution running a v2.6.28-11 Linux kernel. HPC lab 1 uses a v2.1 release of the CUDA software development toolkit, while HPC lab 2 uses a v2.2 release. The original version of SplFd2dmod has been compiled on both workstations using the Intel Fortran Compiler v10.1. For the GPU utilizing implementations, the NVCC compiler of the CUDA software development toolkit has been used. CUDA host code is compiled by NVCC using the systems' GCC compiler. For HPC lab 1, the GCC version is 4.2.4, while HPC lab 2 uses version 4.3.3.

Common to all GPUs listed in Table above, is that they are connected with the PCI Express 2.0 interface. The S1070 in HPC lab 2 is connected with 2 such interfaces.

5.3 Benchmark Results

5.3.1 Results from the HPC workstations

These results are obtained from the HPC lab workstations mentioned in Table 5.1. Each of these tests show results from modeling a single shot with the original, unmodified CPU based version in single processor mode, and the 3 GPU enabled versions. The total dimensions, including 20 points wide PML layers on all sides, is 1481x1101, requiring a total of 6072 thread blocks of dimension 16×16 for each kernel invocation of the different GPU implementations. The total thread count for each of these invocations becomes $6072 \times 16 \times 16 = 1554432$. The following 3 tests benchmark the performance of modeling a shot with these dimensions, doing a total of 13334 invocations of the Fd2dTimestep computations depicted in Figure 4.3.

The results shown in Figures 5.1 and 5.2 show the performance of the modeling on the HPC lab 1 workstation mentioned in Table 5.1 for the single processor mode of the original application, as well as single processor mode versions of the 3 different GPU implementations detailed in the previous Chapter. These two Figures show the performance of using the NVIDIA GeForce GTX280 and NVIDIA Tesla C1060 GPU, respectively. Figure 5.3, shows the same tests for the HPC lab 2 workstation, using one of the 4 NVIDIA Tesla C1060 GPUs of the NVIDIA Tesla S1070 rack unit. All 3 of the following test results show average timings in seconds for 4 runs (see equation 5.2). The Figures also show speedup (see equation 5.1) relative to the original CPU version.

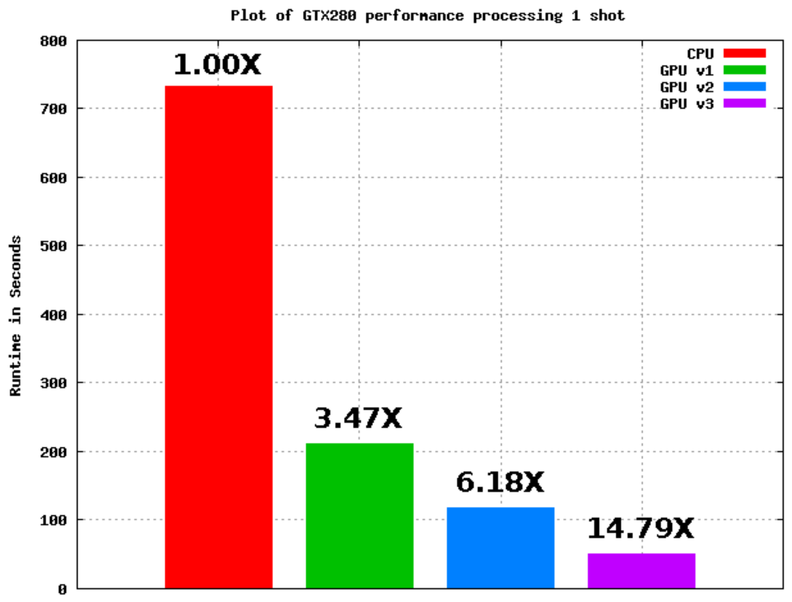

Figure 5.1: HPC lab 1 running NVIDIA GeForce GTX280

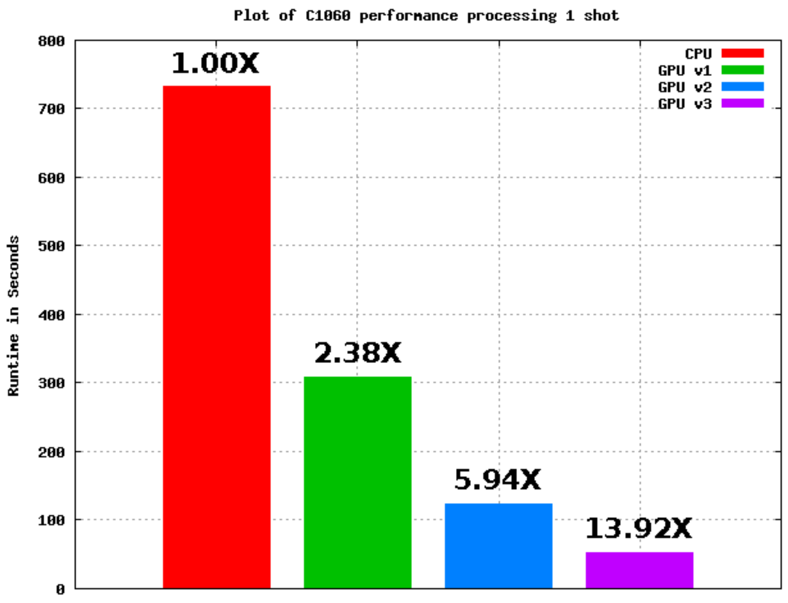

Figure 5.2: HPC lab 1 running NVIDIA Tesla C1060

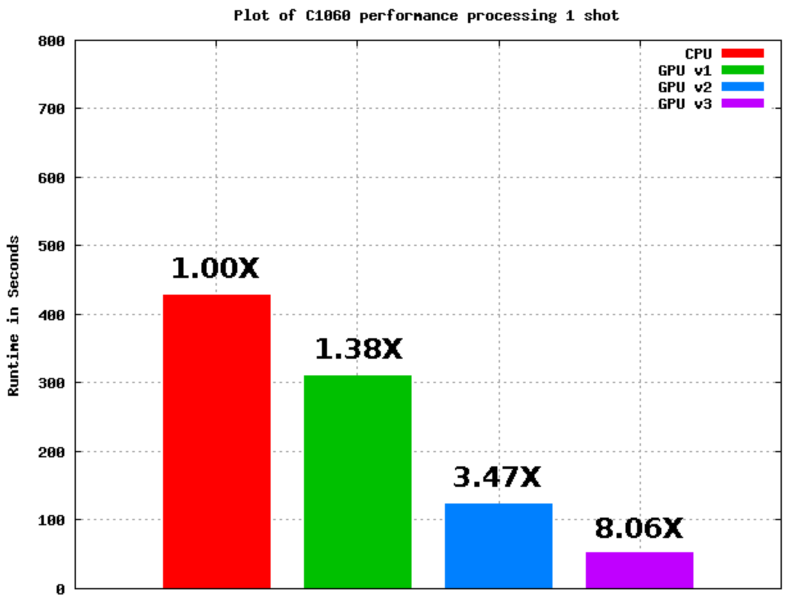

Figure 5.3: HPC lab 2 running single NVIDIA Tesla C1060

5.3.2 Scaling Characteristics

Here we show how well the CPU and GPU versions scale when adding more CPUs/GPUs to the computation on the workstations. Results are collected from the HPC lab 2 workstation listed in Table 5.1. While tests could have been run on HPC lab 1 as well, its GPU configuration is heterogeneous, which will make the results less predictable.

Both tests will run 4 times, adding 1 processing unit to the computation each time. The application will run with the I/O server as a separate process, distributing 12 shots for every run among the participating slave processing units. Running the application in this way will make the total range of processes running for each configuration $[2 - 5]$. The main reason for modeling 12 shots, is that this amount of work increases the chances of an even workload distribution among the participating slave processing units.

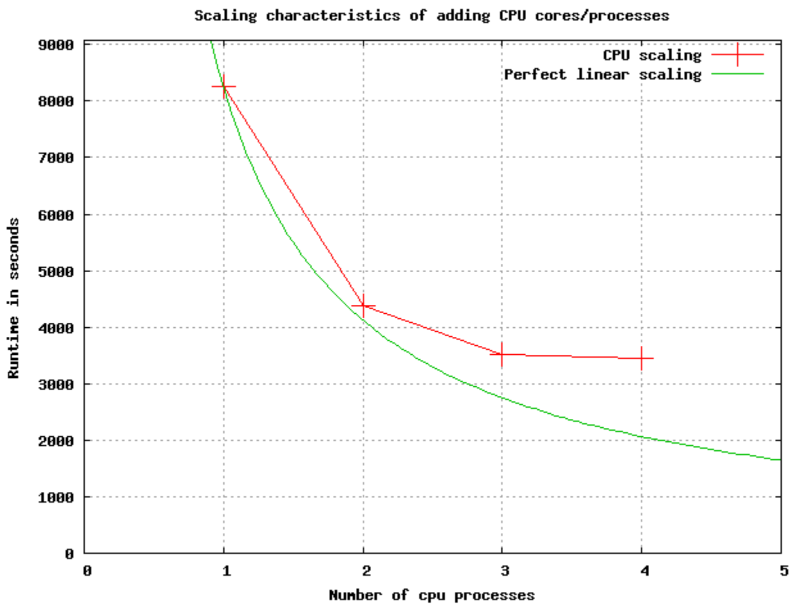

Figure 5.4: CPU Scaling results with {1,2,3,4} processes

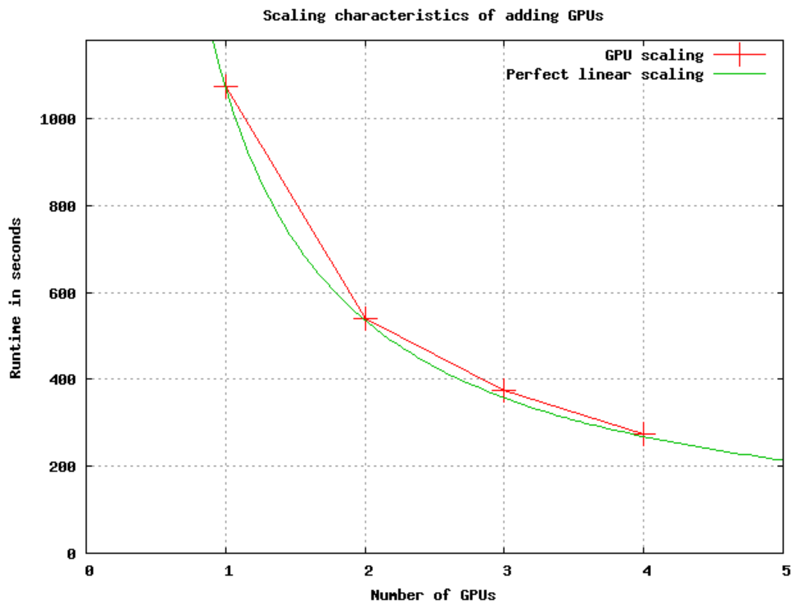

Figure 5.5: GPU Scaling results with {1,2,3,4} processes

5.4 Discussion

The results from the HPC lab 1 workstation shown in Figures 5.1 and 5.2, show a fairly significant performance decrease for the first GPU implementation (*GPUv1*) running on the Tesla C1060. The reason for this can be attributed to the memory bandwidth difference between the GTX280 and the C1060. For the C1060, memory bandwidth is 102GB/s (see [24]), while the GTX280 has a bandwidth of 141.7GB/s (see [25]) which amounts to a difference of approximately 28%. This performance difference can also be seen for the other 2 GPU implementations, but with less impact. The reason for this is that *GPUv2*, which uses shared memory, does far less reads from global memory compared to *GPUv1*. The relatively large performance differences between the three implementations are mainly attributed to the fact that the algorithms perform very few operations for each element read from memory; i.e. they are what is known as *memory bound*.

The performance jump between *GPUv2* and *GPUv3* is largely due to the use of texture memory, and implies that even fewer memory reads are performed due to the higher cache utilization of this final implementation.

The merging of Kernels performed for this implementation also means that some of the kernel invocation overhead from *GPUv2* is removed. *GPUv2* requires additional address computations for the shared memory reads, as well as some extra conditional branches. Both of these requirements have been removed in *GPUv3*, which has also had an impact on performance.

The performance results shown in Figure 5.3 are as expected, the same for the GPU versions as the results of Figure 5.2, this due to the fact that the C1060 is the GPU device running the tests for both workstations (S1070=4×C1060).

For the scaling tests, we observe that near linear scaling is achieved with the S1070, but not with the CPUs. This is not surprising, since the application parallelizes the work by assigning different shots to the different slave processes. Since the processes running on the CPU share the same cache and system memory, per-process performance will decrease as more processes are added. As an added note, we confirmed that our strategy for ensuring even workload distribution worked by looking at the application output for the participating processes. All slave processes modeled the same number of shots for each of the tests in both the CPU and GPU case.

The linear scaling of the GPUs can be attributed to the fact that none of them have to share any of the memory or computing resources with other processes. The tiny portion of interaction with the CPU and the host memory is too small to make an impact from the 1 GPU to the 4 GPU case.

Finally, as we have looked at the compiler output of all of the implementations, we see that the memory requirements (registers and shared memory) per kernel are sufficiently low to allow full SM occupancy. What this means is that each SM of the GPU is able to run 4 blocks, or 1024 threads (maximum of capacity for Compute Capability 1.3, see Appendix A of [10]), simultaneously. The effect of being able to accommodate this amount of threads, is that when a warp executed by the SM issues a long latency read operation, the thread scheduler switches to another warp while the memory operation is performed, effectively hiding memory latency.

Chapter 6

Conclusions & Future Work

In this thesis we analyzed a seismic application for shot processing and successfully developed and implemented a GPU-accelerated version. Our results achieved considerable speedups compared to the original implementation. Implementing the GPU off-loading was done with only minimal modifications to the original source code, primarily in the core modeling function. Since Fortran continues to often be the programming language of choice in the geophysical applications at StatoilHydro, this ability to keep the large portions of source code unchanged when implementing GPU off-loading, greatly reduced the programming efforts required.

An understanding of parallel programming and the C/C++ programming language are the most fundamental requirements for developing CUDA applications. From our experiences developing the GPU implementations of this thesis, the biggest challenge, not being familiar with CUDA programming beforehand, was getting a clear understanding of the targeted application. Learning to use the CUDA programming API was not that time consuming, since it is generally well structured and documented. The efforts put into learning CUDA programming was hence for us certainly a lot smaller than those required for understanding the geophysical aspects of the application. Based on these experiences, we gather that the development efforts required for the developers at StatoilHydro should also be reasonably small.

As was mentioned in the previous paragraph, understanding the targeted application was quite the challenge for this Computer Science student. Because of this, the first working GPU off-loading implementation was ready only 6 weeks prior of submitting this thesis. This fact should be a clear indication of the potential for positive results in the course of a relatively small developing period.

Applications with computational kernels performing data parallel computations are a good match for the data parallel nature of GPU devices. The computational kernel of our application proved to be fairly data parallel in nature, with few serial dependencies. This fact made it a good match for GPU off-loading.

Our benchmarks were done on two different systems in the NTNU IDI (Department of Computer and Information Science) HPC-lab. One system was a Intel Core2 Quad Q9550 @2.83GHz with 4GB of RAM and an NVIDIA GeForce GTX280 and NVIDIA Tesla C1060 GPU. Our second testbed was an Intel Core I7 Extreme (965 @3.20GHz) with 12GB of RAM hosting an NVIDIA Tesla S1070 (4X NVIDIA Tesla C1060). On this hardware, speedups up to a factor of 8-14.79 compared to the original sequential code were achieved, confirming the potential of GPU computing in applications similar to the one used in this thesis. The work done in e.g. [17], [4] and [3], also further supports our conclusion.

6.1 Future Work

Since a lot of the time spent completing this work went into understanding the workings of the targeted application, quite a lot remains to be explored. Following are some of the main issues to be explored further.

6.1.1 GPU Utilizing Mathematical Libraries

We would like to see the potential payoffs associated with using libraries such as the CUDA based *CUFFT* for computing Fast Fourier Transform of 1, 2 and 3 dimensional data, and the CUBLAS library (an adaptation of the Basic Linear Algebra Subprograms (BLAS)) for linear algebra operations in seismic processing related applications.

6.1.2 Further Improvements to SplFd2dmod

As we discussed in the previous Chapter, the computational kernel does few arithmetic operations per memory read, i.e. is considered *memory bound*. In order to achieve optimal speedups in such applications, additional attention to the memory access patterns of the GPU threads is required. We believe that there's still potential for higher performance if the memory access patterns of our kernels is further analyzed.

Another potential performance optimization could be to eliminate the memory transfers between time steps by keeping these values in GPU memory, only doing transfers to/from the GPU at the beginning and end of the modeling loop. We suspect that the performance benefits from this might be relatively small compared to better memory bandwidth utilization as mentioned earlier, but it is still a point of interest. Doing this would also require additional storage in GPU memory, which depending on problem size, may or may not prove to be an issue.

Also, the algorithm for the differentiation depicted in Figure 4.6 could potentially be improved by treating the entire problem domain as a single region. The texture fetch operations of our last implementation (*GPUv3*), has the property of returning a 0 value for fetches made on memory areas not covered by the texture. This could potentially make it possible for us to use the full width of the differentiator stencil in all regions of the problem domain, eliminating the need for adjusting the stencil length in the bordering regions, i.e. regions 1 and 3 in Figure 4.6. Treating the entire problem domain as a whole for the differentiators like this, would eliminate the conditional branches for selecting regions, making all threads of a block perform the same operations regardless of relative location in the problem domain, which should result in increased performance.

6.1.3 Further Testing

The tests performed in this thesis all model shots based on values of the same dimensions, requiring approximately 116 MB. By increasing the resolution of the input velocity and density models of the application, we could get a better view of the applications scaling characteristics with respect to problem size. The procedure to perform such a *re-grid* operation was presented to us too late to be included in this thesis. It would be of great interest to see the applications performance characteristics given larger problem sizes.

The multi-GPU functionality of the application has in this thesis only been tested on a single workstation. The reason for this is that we haven't had access to a compute cluster containing nodes with a distributed file system and CUDA enabled GPUs. Performance comparisons of the multi-GPU enabled implementation on such a cluster with the original running on e.g. one of StatoilHydro's production clusters would be interesting, and a likely added incentive for doing further research on GPU based computational offloading.

Bibliography

[1] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillps, “GPU Computing,” *Proceedings of the IEEE*, vol. 96, May 2008.

[2] D. Blythe, “Rise of the Graphics Processor,” *Proceedings of the IEEE*, vol. 96, May 2008.

[3] H. Nguyen, C. Zeller, E. Hart, I. Castao, K. Bjorke, K. Myers, and N. Goodnight, *GPU Gems 3*. Addison-Wesley Professional, August 2007.

[4] J. M. Cohen and M. J. Molemaker, “A Fast Double Precision CFD Code using CUDA,” 2009.

[5] O. Johansen, “Checkpoint/Restart for Large Parallel Seismic Processing Applications,” Master’s thesis, Norwegian University of Science and Technology (NTNU), January 2009.

[6] L. T. Ikelle and L. Amundsen, *Introduction to Petroleum Seismology*. No. 12 in Investigations in Geophysics, Society of Exploration Geophysicists, 2005.

[7] J. A. Haugen and B. Arntsen, “Spl fd2d module documentation.” Fortran source code documentation. StatoilHydro internal document.

[8] K. Fatahalian and M. Houston, “GPUs - a closer look,” *ACM QUEUE*, March/April 2008.

[9] “Glsl.” Wikipedia article. Website available at ”`http://en.wikipedia.org/wiki/GLSL`”, last visited: June 26, 2009.

[10] NVIDIA, *Cuda Programming Guide 2.1*, December 2008.

[11] The Khronos Group, "OpenCL Overview," February 2009. Website available at "http://www.khronos.org/opencl/", last visited June 26, 2009.

[12] A. L. Shimpi and D. Wilson, "NVIDIA's 1.4 Billion Transistor GPU: GT200 Arrives as the GTX 280 & 260," *AnandTech Web-site*, June 2008. Website available at "http://www.anandtech.com/video/showdoc.aspx?i=3334&p=1", last visited: June 26, 2009.

[13] NVIDIA, *NVIDIA GeForce GTX 200 GPU Architectural Overview*, May 2008.

[14] D. Kirk and W. Hwu, *Cuda textbook*. 2008-2009.

[15] R. Farber, "Cuda, Supercomputing for the Masses," *Dr. Dobb's Portal*, 2008-2009. Website available at "http://www.ddj.com/cpp/207200659", last visited: June 26, 2009.

[16] M. T. Heath, *Scientific Computing - An Introductory Survey*. McGraw-Hill, second ed., 2002.

[17] P. Micikevicius, "3D Finite Difference Computation on GPUs using CUDA," 2009.

[18] "Valgrind Instrumentation Framework." Website available at "http://valgrind.org", last visited: June 26, 2009.

[19] O. Holberg, "Computational Aspects of the choice of Operator and Sampling Interval for Numeric Differentiation in Large-scale Simulation of Wave Phenomena," *Geophysical Prospecting*, vol. 35, 1987.

[20] J.-P. Berenger, "A Perfectly Matched Layer for the Absorption of Electromagnetic Waves," *Journal of Computational Physics*, vol. 114, 1994.

[21] Q.-H. Lui and J. Tao, "The perfectly matched layer for acoustic waves in absorptive media," *Journal of the Acoustic Society of America*, vol. 102, October 1997.

[22] T. Wang and X. Tang, "A Nonsplitting PML for the FDTD Simulation of Acoustic Wave Propagation," September 2001. Society of Exploration Geophysicists International Exposition and Annual Meeting.

[23] NVIDIA, *Cuda Reference Manual 2.1*, November 2008.

[24] "Tesla C1060 Board Specification," September 2008. Website available at "http://www.nvidia.com/docs/IO/56483/Tesla_C1060_boardSpec_v03.pdf", last visited: June 26, 2009.

[25] "NVIDIA GeForce GTX280 Specifications." Website available at "http://www.nvidia.com/object/product_geforce_gtx_280_us.html", last visited: June 26, 2009.