TDT4590 Complex Computer Systems
Specialization Project, Fall 2009

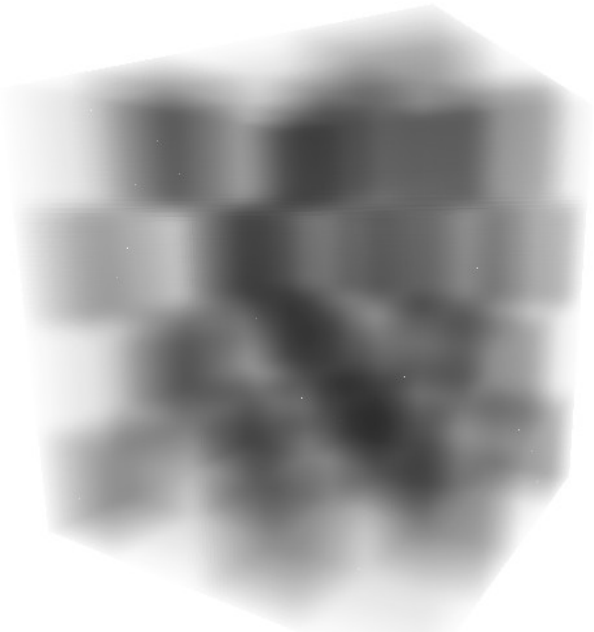
Ahmed Adnan Aqrawi

Three Dimensional Convolution of Large Data Sets on Modern GPUs

Supervisor: Dr. Anne C. Elster, IDI
Co-supervisor: Victor Aarre, Schlumberger Stavanger

Trondheim, December, 2009

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science



NTNU
Norwegian University of Science and Technology

Problem Description

In Petrel, Schlumberger's seismic software, one often comes across large seismic cubes that need to be filtered in order to generate clearer images. The seismic cubes are viewed from three dimensions, implying that one must filter in all three dimensions as well. However, this filtering is very computationally demanding and thus uses a lot of computational resources. This project's goal is to implement a Gaussian filter on a large three dimensional data set on the GPU using NVIDIA CUDA to off-load the CPU. A CPU version will also be developed for comparison and analysis. Since the size of the data to be transferred to the GPU memory is quite large, the calculations need to be performed on sub cubes. This implies that one must account for border data between cubes to avoid an edge effect. The implementations developed will be benchmarked and compared to evaluate performance gains.

Abstract

In the petroleum and gas industry, one of the main foci is using seismic processing to find new oil and gas reservoirs. Recordings of seismic waves can be used to create images representing the surface of the earth. To do so one has to filter the data collected. One of the methods for filtering this data is convolution in the spatial domain. Which is done in three dimensions (3D) because of the 3D nature of the data collected. The data collected can be of surfaces over several kilometers in length and are therefore very large is size.

This project focuses on implementing a 3D convolution algorithm on modern CPUs and GPUs with non-separable filters for large data sets, in the spatial domain. Our results demonstrate that the filtering mask should be placed in constant memory rather than shared memory because there is an overhead assosiated with the use of shared memory per kernel launch. The data in constant memory must be read coalecsed for it to be efficient. Shared memory should not to be used for the filtered data either due to the lack of communication between the threads in the convolution kernel. Again, the overhead of reading into shared memory only slows down the process. To compare our results, implementations on the CPU were performed in C. The platforms tested on are both a uni-core CPU and a quad-core CPU, as well as a single GPU and a system with up to 4 GPUs. The CPU used is a AMD phenom x4, whereas the GPUs used are the NVIDIA Tesla c1060 and NVIDIA Tesla s1070. Our work includes figuring out how to process large amounts of data most efficiently on both the CPU and GPU with the use of different blocking methods when accessing the disk.

Our results also show that the I/O time, which one would expect to be a bottleneck, is only 1-2% of the total execution time on a single CPU. This means that convolution is a computationally demanding task, but fortunatly a very parallelizable one. Our results indicate that compared to a single core a speedup of 3.57 is achieved on the Phenom x4 , a speedup of 17 is achieved on the Tesla c1060 (single GPU) and a speedup of 62 is achieved on the Tesla s1070 (4 GPUs). This led to the computation percentage being reduced by 5%, 25% and 90%, respectively for the three platforms. Further work regarding optimizations should hence focus on I/O.

Acknowledgement

This report, together with the prototype, is the result of a project assigned by the course TDT4590 at the Norwegian University of Science and Technology.

I would like to thank my supervisors Dr. Anne Cathrine Elster for invaluable support and feedback throughout the entire project. She has been an inspiration with her great understanding and dedication to the field. Given her generosity and encouragement all the resource needed for this project where made available. I would like to thank Victor Aarre of Schlumberger for his support in providing me with new ideas, example source code and a set of seismic data. I would especially like to thank NVIDIA for sponsoring of our group and our HPC-lab, and for making high-end graphics cards such as Tesla c1060 and Tesla s1070 available. I would also like to give thanks to the entire HPC group for their support, encouragement and enthusiasm for this project. And a special thanks to Jan Christian Meyer, Thorvald Natvig, and Holger Ludvigsen for all their help.

Trondheim, Dec 2009

Ahmed Adnan Aqrawi

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

In the oil and gas industry, there is always an interest in investigating potential oil and gas reservoirs. There are several ways in which to test for this, and one of these is seismic data collection. Seismic data is gathered by recording seismic waves (waves of force that travel through the earth). This data is used in the field of petroleum to discover the geological structures of the earth and find natural resources such as oil and gas. To help in this search seismic data is processed by many filters and filtering methods to get a clearer subsurface image and to view more relevant information such as faults and reservoirs, see FIgure **??** for an example of seismic data. These filters are like other image filtering processes very adaptable to the graphical processing unit (GPU), but are per today run on the central processing unit (CPU).

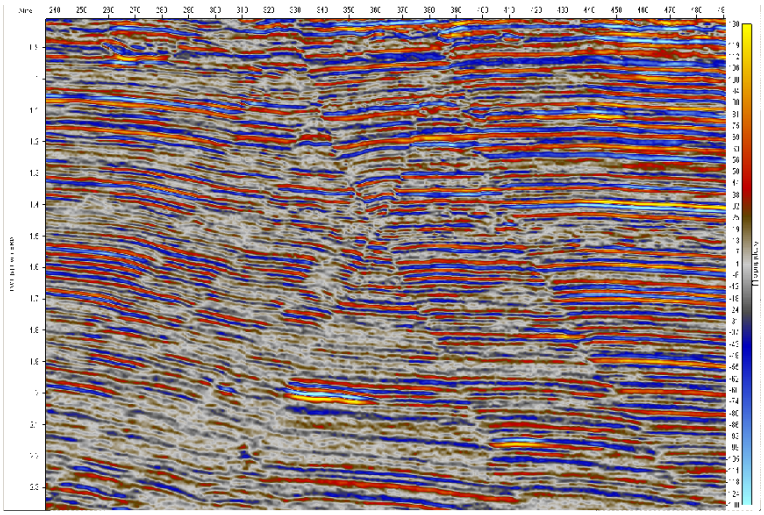


Figure 1.1: Figure illustrating seismic data from [**?**], with permission from Schlumberger

In recent years, it has been shown that the performance capabilities of the GPU, in some cases, has exceeded that of the CPU. Which in turn motivated the development of the general purpose graphical processing unit (GPGPU). This has lead to the use of the GPU not only in graphic applications, but also in scientific calculations. These trends have created a boom in the graphical processing architectures

and manufacturers have started introducing new product lines specific for scientific calculations. In Figure ?? one can see the trend of computational power measured in floating point operations pr. second (FLOPS) for the past 5 years. Another aspect worth noting is that the use of the GPU gives room to use the CPU for other tasks in parallel, functioning as an accelerator.

Given these advancements one is now often interested to see if it is possible to utilize the GPU for calculations and gain some increased performance for certain tasks. Such tasks as image processing, seismic processing and other physical modeling as well as linear programming applications have proven to be well parallelizable on the GPU. This gives the foundation of this projects existence in that we are to perform an image enhancement task on seismic data on the GPU.

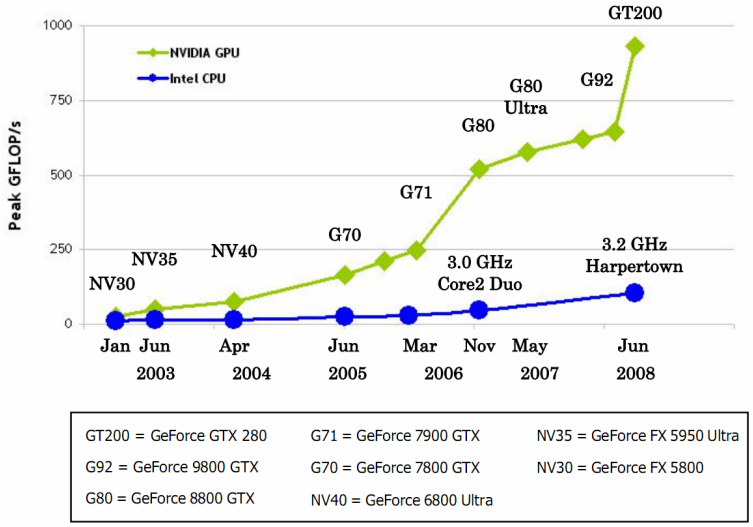


Figure 1.2: Figure illustrating CPU and GPU performance trends from [?], with permission from NVIDIA

1.1 Project Goals

The aim of this project is to implement convolution for non separable filters in the spatial domain in CUDA, for large three dimensional data sets. A large data set is defined as a data set that does not fit into modern system buffers, currently at sizes between 8-12 GB. The Gaussian filter is a filter used in seismic processing and implementing it on the GPU would introduce new possibilities in the field of pre-processing seismic data. The challenges here are how to handle large datasets. Meaning that one must compute the data set in intervals of sub-sets and must account for border information to compute the filters correctly. For comparisson implementations will be developed for both a single and quad -core CPUs. The goal is to benchmark the convolution implementations on modern CPU and GPU with different filter sizes and compare the two to see which is most efficient when it comes to large data sets. Possibilities to run on several GPUs to accelerate performance will also be explored, and speedup will be assessed.

1.2 Project Contributions

There are three main contributions in this project. The first is to perform convolution in three dimensions with non separable filters. It is a rare thing to find convolution performed in three dimensions not to mention on the GPU. This should be useful for anyone aiming at using the GPU for any similar tasks.

The second is to look at handling large data sets. This introduces many problems from disk access to transfer of data to memory. In this project the focus is on the retrieval of data to the GPU by blocking across different dimensions. The combination of using large data sets on the GPU is also a rare occurrence. Since usually the data used is exactly large enough to fit in the systems main buffer.

The third significant contribution here, is the use of the GPU and CUDA in seismic processing and how it can accelerate that process by experimenting with the different memories in the CUDA hierarchy (for example constant, shared and global -memory). There have been some studies regarding the topic accelerating seismic processing, but in our project the focus is on using convolution as a filtering method and the use of CUDA to program on the GPU. In our project there are also considerations regarding the use of multiple GPUs to accelerate the process, which is both rare and interesting to see how the algorithm scales on several hundered cores.

1.3 Outline

The rest of this report will structured as follows:

Chapter 2: Relevant researched background material and related work is emphasized and explained such that the reader has all the presumed knowledge to understand the rest of the work. It is also a way to show how this project builds upon existing work in the same field.

Chapter 3: A short introduction to which hardware and software is used in the project. A description of how the implementations in this project were performed, and why certain implementation choices were made are explained. Here one will also find the thoughts put behind each optimization and what the expectations are as to how they will perform and test.

Chapter 4: Results regarding I/O tests are presented and discussed. The main focus is the blocking techniques used to achieve good disk access times and explaining why they are so efficient. Results regarding convolution tests on various platforms are presented and discussed as well. The main focus here is on comparing the implementations and presenting speedup and computation percentage. An in depth analysis of the comparisons and traits are also shown.

Chapter 5: Here one will find the conclusion of the work performed and suggested futher work in the field.

Appendix: Tables of results gathered during the benchmarking process are included in the appendix. some of these results are summarized in graphs in Chapter 4.

CHAPTER 2

Background and Related Work

In this chapter, the focus is on introducing some of the main sources the reader might need to understand our work. The following sections summerize the main references read. Section 2.1 inroduces related work done in similar fields. Section 2.2 concerns spatial filtering. Section 2.3 Explains the concept of a filtering mask and the gaussian filter. Section 2.4 is a practical example of how the gaussian filter is used in seismic processing. Section 2.5 is about general parallel programming. Section 2.6 introduces OpenMP and the concepts of multithreading. Section 2.7 Explains the main apects of the CPU and GPU architectures. Finally, Section 2.8 Gives a short introduction to the CUDA programming model.

2.1 Related Work

This section will focus on introducing papers and theses chosen to be discussed with the intention to emphasize work done in a similar field before and how this project will build upon them. The main fields focused on here are image processing, convolution, GPU accelleration, three dimensional data and multiple GPU systems. All these topics are relevant to this project, and have been researched to lay a foundation for the implementations performed.

Image Convolution with CUDA, 2007 [?]

This is a paper written by NVIDIA to show how CUDA can be used to perform convolution in image processing. This is related to this project in that it also concerns convolution in the spatial domain and it is also implemented in CUDA. In contrast to this paper, the image processing performed in this project is on three dimensional data, and the data to be filtered does not fit in memory and so one must perform several communications in the memory hierarchy.

Accelerating 3D Convolution using Graphics Hardware, 1999 [?]

This is a paper published by IEEE Visualization in 1999 that approaches the subject of 3D convolution performed on a GPU. The main idea here is to use the graphical hardware to accelerate the convolution process. This is work done in this area pre CUDA and this is where it differs from this project. Since before the CUDA architecture the use of shared memory was not available and this can be a good enhancement/optimization. Other than the use of CUDA this project differs in that it also considers the use of multiple GPU to accelerate the process and is concerned with larger data sets.

Modeling Communication on Multi-GPU Systems, 2009 [?]

This is a master thesis concerning the use of communication and calculations on several GPUs simultaneously. Another important subject taken into account here is partitioning of data such that calculations can be done on several GPUs. This is relevant to this project because of the large amount of data to be filtered and the advantage of using multi-GPU. It is also interesting to see how one can partition data such that communication between GPUs is optimal. In contrast to this thesis the problem solved here is of image processing and not a solution to partial differential equations. Another difference is again the consideration of large data sets.

Paulius Micikevicius, 3D Finite Difference Computation on GPUs using CUDA, Nvidia, 2008. [?]

This is a paper by Nvidia that shows how one can process three dimensional data. The implementation is specific for a Tesla 10 series GPU, which is the same used in this project. The most interesting aspects here are that this is an ideal example of using CUDA on a three dimensional dataset and that its specificly developed by Nvidia for their 10 series GPU. All of these three elements are present in this project as well. There is also a discussion here on the use on multiple GPUs, which is also covered by this project.

Eirik Aksnes, Simulation of Fluid Flow through porous rocks on modern GPU, 2009. [?]

This is a masters thesis, which focuses on a physics simulation in three dimensions. This was used in relation to programming large three dimensional objects on the GPU. This a good source since here as well as in this project the GPU in use is the Nvidia Tesla c1060, and here too the focus was on three dimentional data. This is not a heavy cited reference, but nevertheless it is a useful source since it discusses similar aspects.

2.2 Spatial Filtering

There are two somewhat similar concepts that must be understood clearly when performing linear spatial filtering. One is correlation and the other convolution. Correlation, speaking from an image processing point of view, is when one moves a filtering mask across an image and computes the sum of products at each location. Convolution is similar, but the filtering mask is first rotated 180 degrees. This is best shown with a trivial example of 1-D filter on a 1-D image. See Figure **??**.

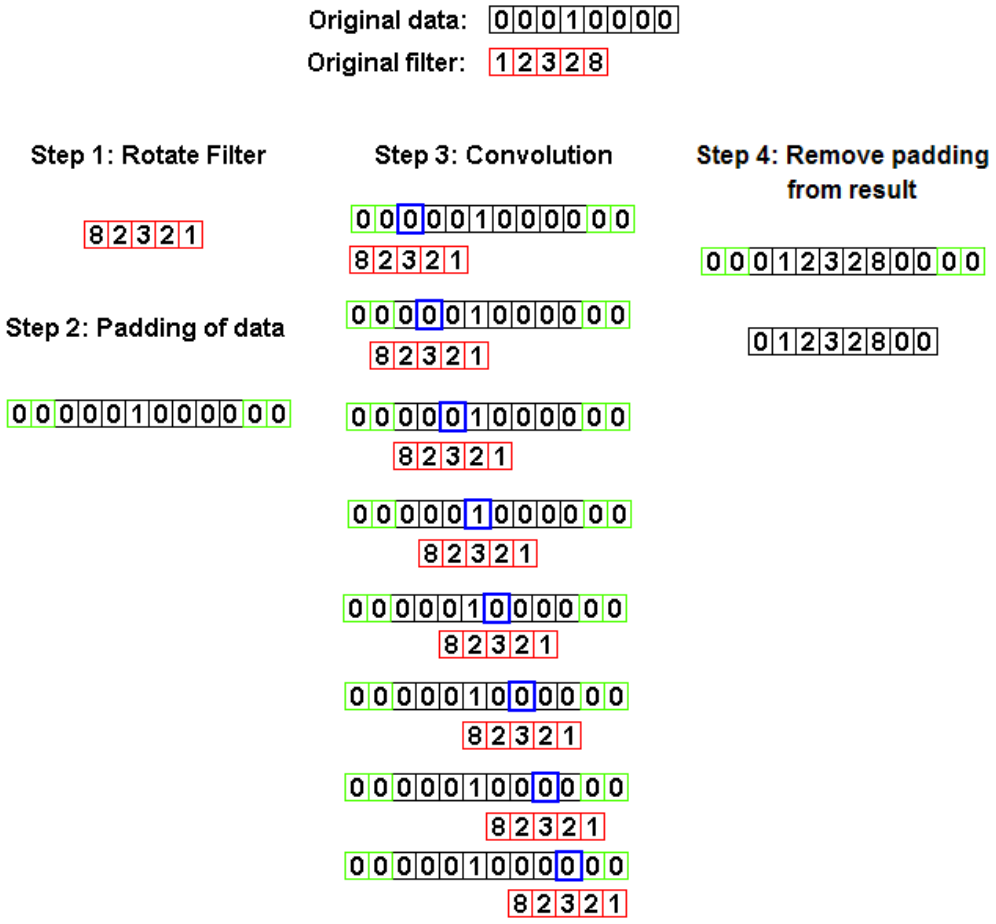


Figure 2.1: Image illustrating Convolution in 1D. Inspired by an example in [**?**]

The first thing one notices is that if one is to overlap the filter on all the value in the image, i.e. move each value of the filter on every pixel in the image, then we would have to pad the image with zeros such that it is possible. This means that if the filter is of size m, then we need a padding of size m-1 to the left and right of the image. In this case a padding of size four is added on each side of the image since the filter contains five values. Another aspect worth noticing here is that correlation and convolution are functions of displacement. Where one starts at a displacement of zero and increases by one as one filters the image.

Convolution

From a mathematical point of view, convolution is an operation involving two functions that produces a new function. This new function reflects to which extent the original functions match if their graphs were aligned with each other. Convolution is mathematically defined as Equation **??** [**?**].

$$(f \star g)(t) = \int_{-\infty}^{\infty} f(a)g(t-a)\,da \tag{2.1}$$

$$w(x,y,z) \star f(x,y,z) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} \sum_{r=-c}^{c} w(s,t,r) f(x-s, y-t, z-r) \tag{2.2}$$

The continuous mathematical definition is not that useful in our case since we would like to deal with discrete cases in programming. To convert the preceding in to a discrete function we would have to add discrete convoluted values from each function. If we have that $w(x,y)$ is the filter of size $m \times n$ that will be convoluted with an image $f(x,y)$, denoted as $w(x,y) \star f(x,y)$. This gives the Equation **??** [**?**], which is a discrete sumation equivalent to the continous mathematical definition of convolution.

2.3 Spatial Filter Masks and The Gaussian Filter

While discussing filtering in the spatial domain we mentioned a filtering mask, which is used to convolute with the image. These masks are generated depending on their purpose, such as smoothing, edge detection, edge enhancement, etc. The filters are decided by a given size $m \times n$ and with the use of a mathematical function one can calculate the values of the filter discretely. We will later see that the nature of the function used will result in how well the filter will perform.

Gaussian Smoothing

Using a Gaussian filter, also known as Gaussian smoothing, is an operator that is used to blur images and remove detail and noise. This is similar to the way a mean filter works, but the Gaussian filter uses a different kernel. This kernel is represented with a Gaussian bell shaped bump. This kernel has some special properties regarding separability that we will look at in detail.

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{2.3}$$

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.4}$$

The Gaussian distribution in the 1D and 2D cases are shown in Equations **??** and **??** (from [**?**]), where σ is the standard deviation of the distribution. The functions are illustrated in Figure **??**. These can be used to give a better idea of what a Gaussian distribution is. The idea of Gaussian smoothing is to use these distributions as a point spread function to create a filtering mask and by using convolution one is able to blur an image. Since images are usually stored as discrete pixel values one would have to use a discrete approximation of the Gaussian function on the filtering mask before performing the convolution. See Figure **??** to see an example of a discrete approximation of a gausian filter in two dimensions.

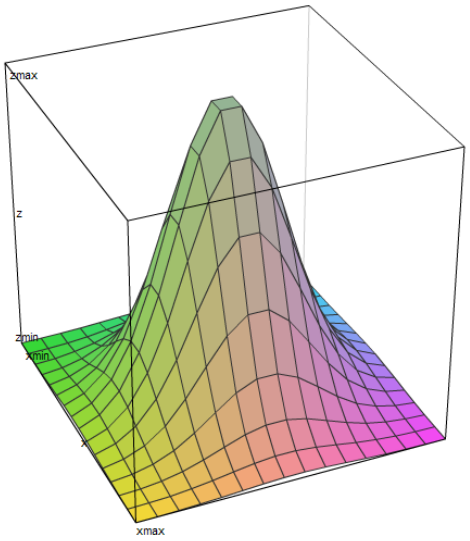


Figure 2.2: 2D Gaussian distribution graph drawn in online 3D grapher [1]

Gaussian Filter Mask in Practice

In theory the Gaussian distribution is non-zero, which would imply an infinitely large convolution kernel. But, in practice it proves to be 'almost zero' more than three standard deviations from the mean. This implies that the kernel can be truncated after three standard deviations. Once a suitable kernel has been calculated then the Gaussian smoothing can be performed using a discrete convolution method as explained earlier. The Gaussian filter is separable if *circularly symmetric* meaning that one can use a one dimensional filter to filter images. For example if the image is three dimensional then one can convolute three times using a one dimensional filtering mask, once in each dimension. If the Gaussian function is *elliptical* than it is not separable and this would result in using a three dimensional filter once in all three dimensions.

[1] http://www.livephysics.com/ptools/online-3d-function-grapher.php, accessed 2009-12-09. Available to all since it is public domain

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

Figure 2.3: Example of two dimensional Gaussian Filter Mask with discrete values

2.4 Gaussian Smoothing in Seismic Processing

Figure **??** illustrates and example of the practical use of gaussian smoothing in seismic processing. This is data made accessible by Schlumberger and is released into public domain. Here one can see how the gaussian filter is used to blur the image such that noise can be eliminated. This is best seen in the lower left corner of the image. The type of filtering done in this project will be used with the same purpose in seismic processing.

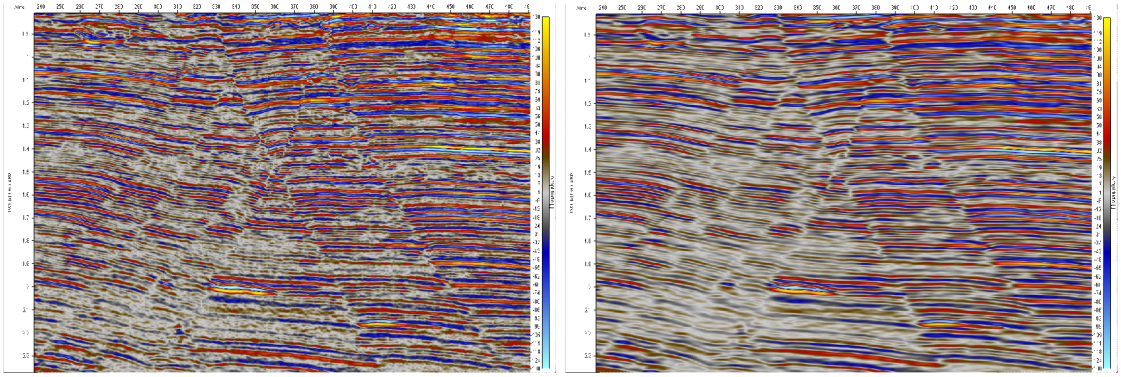


Figure 2.4: Figure illustrating seismic data before and after gaussian smoothing from [**?**], with permission from Schlumberger

2.5 Parallel Computing

Parallel computing is a form of computation where many calculations are performed simultaneously. This is depending on that large problems can be divided into smaller ones such that one can calculate them concurrently. There are several different forms of parallel computing bit-level, instruction level, data and task parallelism. Parallel computing has been around for quite a while now, but has lately been given more attention since the limitations of power consumption and physical limitations of

frequency levels on computational hardware have started to stagnate. This gave birth to the multi-core CPUs and created a paradigm in computer architecture.

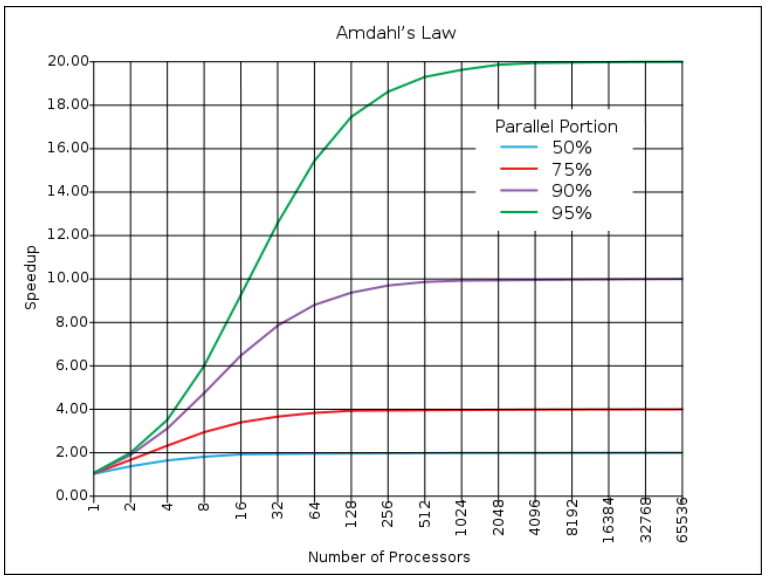


Figure 2.5: Image illustrating Amdahl's law [2]

Amdhal's Law

When generally speaking of how effective a parallel algorithm is one usually mentions the speedup factor (how much faster a parallel algorithm runs compared to a sequential one). The potential speedup of an algorithm on a parallel platform can be measured with the help of Amdahl's law. This law states that all parallel computations are limited with a sequential part of their code and thus their speed up is limited by this part as well. Amdahl's law is stated as in Equation **??** [**?**] and illustrated in Figure **??**. S is the speedup, and P is the portion of the code that is parallelizable and is a value lesser than 1 and greater than zero.

$$S = \frac{1}{1 - P} \tag{2.5}$$

Gustafson's Law

Another law closely related to Amdahl's law in computer science is Gustafson's law **??** [**?**]. Here S is the speedup, P is the number of processors and α is the non-parallelizable part of the program. Amdahl has made the assumptions that the sequential part of the program is independent from amount of processors one has while Gustafson believes that this is not the case.

[2] http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg, accessed 2009-09-08. Available to all under the license of creative commons

$$S(P) = P - \alpha(P - 1) \tag{2.6}$$

Speed Up Limitations

Ideally one should be able to gain linear speedup meaning that one would increase the speedup with the amount of processing units available. For example if one has two processors then it should take half the time to execute a program since two are working on the problem. But, this is rarely the case since there are some factors that limit speedup that one should look out for when programming in parallel. One is that not all problems can be parallelized at all time, they might have some sequential parts, which means that while one processor is executing the sequential part the other is idle. This results in sub-linear speedup. Another fact is that not all algorithms can be executed in parallel without extra computational cost. Some implementations might have this extra cost in the parallel version and this will also result in sub linear speedup.

Other speedup limitations are I/O and communication time between processors. In some systems not all processors are capable of reading and writing to I/O units which means that the other processors will have to wait, resulting in lesser speedup. Often processes have to communicate with each other like in the case of border exchanging. This will result in time spent communicating which is not present in the sequential algorithm. This will also result in sub linear speedup. These effects become more evident when dealing with large datasets since one can use a lot of time reading and writing data, and more data will have to be communicated between processes.

Data Dependencies

Data dependencies are a common issue in parallel programming and should be understood to perform good parallel implementations of algorithms. The fact that no program can run faster than its longest chain of dependencies will prevent a lot of speedup, this is known as the critical path. This is because some calculations normally depend on other calculations and if they are not completed then one cannot proceed to the next step i.e creating a delay. However, most algorithms do not consist of a long chain of dependent calculations and can therefore run concurrently.

$$I_j \cap O_i = \emptyset \tag{2.7}$$

$$I_i \cap O_j = \emptyset \tag{2.8}$$

$$O_i \cap O_j = \emptyset \tag{2.9}$$

A good description of dependencies are the Bernstein conditions [**?**]. These describe when two program fragments, denoted P_i and P_j, are independent and can be executed in parallel. For P_i let I_i be the input and O_i be the output for the program. For P_j the input will be I_j and output O_j. Now we can say that P_i and P_j are independent if they satisfy the following conditions **??**, **??** and **??** (from [**?**]).

Types of Parallelism

Instruction-Level Parallelism

Computer programs are in essence a collection of instructions. Some instructions have dependencies, such as an instruction should be performed before another can use its results and so on. The instructions of a program can be collected into groups and executed in parallel without effecting the results of the program. This is often seen in pipelining architectures. In addition to instruction level parallelism from pipelining, some processors can execute more than one instruction at a time. These are known as super scalar processors. Instructions can be grouped only if there are no data dependencies between them.

Data Parallelism

In data parallelism one is focusing on distributing work on several computing nodes, and this is often inherent in program loops. In loops one is often performing similar functions or calculations on a large data set and if these are independent of previous states one is able to calculate these concurrently. Many scientific and engineering applications have data parallelism, it is also present in GPU applications.

Classes of Parallel Computers

There are many classes of parallel computers and they are usually not mutually exclusive. The classes are as follows: multi-core computing, symmetric multiprocessing, distributed computing, cluster computing, massive parallel processing. There are also specialized parallel computers such as FPGA (field programmable gate arrays), GPGPU, application specific integrated circuits and vector processors. I will be focusing on Multi-core CPUs and GPGPU because these are within the scope of this project.

Multi-core CPU

A multi-core processor is a processor with several execution units, also known as cores. This is different from a super-scalar processor in the fact that super-scalar processors issue multiple instructions per cycle from multiple instruction streams. Each core in a multi core processor can be super scalar. An interesting attribute with these multi-core processors is that they can perform simultaneous multi-threading.

This means that while one execution unit is performing calculations on multiple threads and a thread is idle, another execution unit can do calculations on the idle threat.

GPGPU

General purpose computing on the GPU is a relatively new trend in computer engineering research. The GPU is an accelerator and co-processor that has been optimized to handle graphical tasks. Computer graphics processing is generally influenced by large data parallelism operations and in particular linear algebra matrix operations. Previously the GPGPU was normally programmed using the graphics API, such as the use of textures to perform calculations. Recently several new programming languages and platforms have been developed for the purpose of general purpose computing on the GPU. NVIDIA and AMD have introduced the CUDA and CTM programming environments respectively. Other GPU programming languages are BrookGPU, PeakStream and RapidMind. There is also the Introduction of a the Framework OpenCL for writing programs on heterogeneous platforms consisting of CPUs, GPUs and other processors. The scope of this project extends to the use of CUDA, the programing platform developed by NVIDIA.

2.6 OpenMP and Multithreading

OpenMP is an implementation of multi-threading. Multi-threading is when one parallelizes code with the use of threads. Here a master thread running the program creates slave threads, by forking threads, and the task is divided between them. The threads run concurrently with the runtime environment allocating threads to different processors. The sections of code that are to run in parallel are marked with a pre-processor directive which will cause the forming of threads before the section is executed. Each thread will be given an ID, where the master thread has an ID of "0". After the execution of the parallel code the threads are then joined to the master thread that continues onward with the program. This is shown in Figure **??**

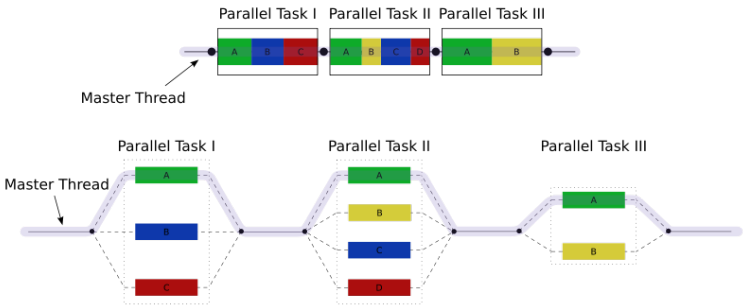


Figure 2.6: Image illustrating the Fork and Join of threads with OpenMP [3]

[3] http://en.wikipedia.org/wiki/File:Fork_join.svg, accessed 2009-09-11. Available to all under the license of creative commons

By default, each thread executes independently from the other hence giving good parallelism results. There are possibilities for work sharing constructs to divide the tasks between the threads such that each thread executes its allocated part of the work. Both task parallelism and data parallelism can be done using OpenMP in this way.

Main Features

The main OpenMP elements include thread construction, work sharing constructs, data environment management, synchronization of threads, user level runtime routines and environment variables. Thread creation happens when a thread is forked from a master thread, this is done by using the compiler directive "# pragma omp parallel". For a broad overview of OpenMP language extentions see Figure **??**.

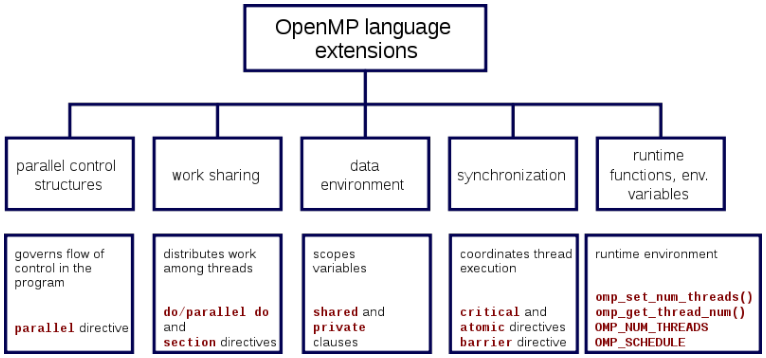


Figure 2.7: Image illustrating language extentions of OpenMP [4]

The work sharing constructs is used to specify how one can assign independent work to one or all threads given the code that will be run. Here one has the option to specify with commands whether one wants a single (master) thread to perform a code block. One can perform calculations on specific sections of independent code blocks to different threads or use loop constructs to split up loop iterations among the threads. This can be done by adding to the command like this "# pragma omp parallel single".

It is also possible to specify if the data is to be shared among the threads or if some of the variables are private by using data sharing attribute clauses. These are also specified in the command such as in this example "# pragma omp parallel for private(i,j,k)". The variables i ,j and k are not shared among the threads but rather each thread has a separate (private) copy.

With the use of OpenMP one is also in control of the scheduling one is to use by using the scheduling clauses. There are three types of scheduling to choose from static, dynamic and guided (which is a variation of dynamic). The static scheduler is used when one before execution decides which threads have which allocated iterations. Dynamic scheduling is when a thread is able to retrieve other iterations if its done earlier than others.

[4] http://en.wikipedia.org/wiki/File:OpenMP_ language_ extensions.svg, accessed 2009-09-11. Available to all since the author has released it into public domain

When multi-threading one might come across a situation where one thread is done earlier than others and one might have problems with synchronizing between the threads before approaching another task. That is why in OpenMP one will find synchronization clauses. Here one can use commands to create barriers such all threads synchronize to that point before moving on. One is also able to specify that a region is critical such that one can avoid deadlocks. In critical sections only one thread is allowed to perform instructions at a time.

Thread creation and work sharing constructs

By specifying "parallel" in the pragma one will fork into multiple threads that perform the instructions enclosed in the construct in parallel. By default OpenMP will set the amount of threads to that which matches the amount of cores available on the machine. Given that one is running on a Quad-core then 4 threads will be created. The original process will then be denoted as master thread and gets thread id 0. Once the threads are done they will be joined and the master process will continue running on a single core.

After indicating that code is to be run in parallel one can also specify how the threads should share the work load. An example would be the use of the "for" construct to specify that a loop is to be parallelized, and will result in that the loop iteration will be split among all the threads. There are also constructs that enable one thread to run while others are waiting or just signaling that a section of independent code can be run in parallel.

Pros and cons

One of the greatest strengths of OpenMP is that it is easy and simple to use, and that one does not have to deal with message passing. OpenMP directives automatically handle data layout and decomposition. The use of incremental parallelism results in no dramatic changes to ones code. Since one does not have to modify ones serial code much when using OpenMP makes it less likely to cause errors. OpenMP code can also be compiled for both serial and parallel code where in the serial compiler recognizes OpenMP syntax as comments. This results in that it will compile in both cases.

OpenMP is a great tool, but it also has its limitations. Such that it does not have reliable error handling. The scalability of an OpenMP program is bound by the memory architecture on which it runs. Currently it only runs efficiently on shared memory multiprocessor platforms. It also requires a compiler that supports OpenMP.

Performance expectations

One could expect that running code on n cores would result in n times the speedup in OpenMP, but this is not the case because OpenMP is also affected by the com-

mon problems that apply to parallel computing generally. Parts of the code in an OpenMP program run on only the master process resulting that the speedup theoretical upper limit is limited by Amdahl's law, in that parts of the code are sequential. Another aspect is that the memory bandwidth also limits speedup since in shared memory often the same path is used for all threads to get data, and so they must be interleaved. Other aspects resulting in overhead are synchronization and load balancing between all the threads.

2.7 GPU and CPU Architectural Features

The graphical processing unit is a processor dedicated to rendering graphics and functions as an accelerator such that it offloads the CPU when processing graphical data. The graphical processors architecture has mainly focused on SIMD instructions. Such that it can run several similar instructions simultaneously on several threads. Recently these graphical accelerators have become of interest in the field of high performance computing (HPC). Driven by the gaming industry and their never ending demands for more realistic computer graphics, the GPU has evolved from a primitive processor only able to preform restricted graphics rendering operations to being a programmable processor with huge performance capabilities. The theoretical floating-point processing power of the graphical processor has greatly exceeded that of a CPU.

Figure 2.8: Image illustrating CPU and GPU transistor usage and layout from [**?**], with permission from NVIDIA

Main aspects of CPU architecture

The GPU is mainly thought of as an accelerator, and it is therefore important to mention that it is intended to be used in cooperation with a CPU. It is merely an addition and not a substitute. The CPU has different capabilities and as such is a more flexible processing unit. It is designed to maximize the performance of a single thread of sequential instructions. The CPU is able to perform instruction level parallelism, and as such it can optimize its performance by executing several different

instructions simultaneously (SPMD). It also supports flow control that allows it to take advantage of the instruction level parallelism (See Figure **??**). Another interesting aspect of the CPU is the use of many transistors to avoid memory latency with the use of caches, see Figure **??**. This is important in the combination of flow control and instruction level parallelism such that one does not have to access main memory every time one changes the instruction performed. And as such it is very important to optimize caching and memory access when programming on the CPU to obtain maximum performance.

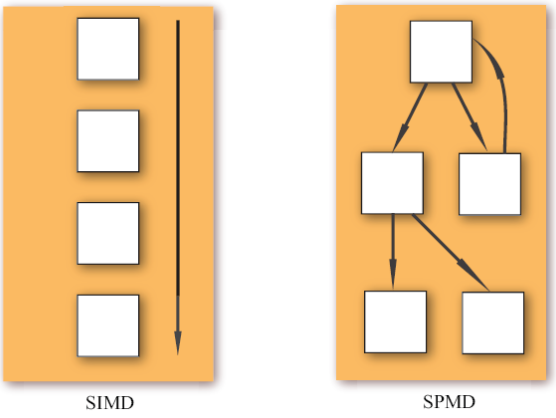


Figure 2.9: Image illustrating SIMD and SPMD from [**?**], with permission from NVIDIA

Recently even CPUs have been adopting parallelism and run with several cores. In this project a Quad-core processor is used, meaning that there are 4 cores in the CPU that can perform calculations. The special feature here is the way in which the caches are used. Each core has a L1 and L2 cache, and they share a L3 cache. Again one sees that the focus is on memory latency and caching of data, but it is important to know that there are three levels of caching here rather than two when programming. This also introduces more advancd scheduling and flow control units, given that the different cores must co-operate and share data. See Figure **??** for a closer look at the architecture of the AMD Phenom used in this project.

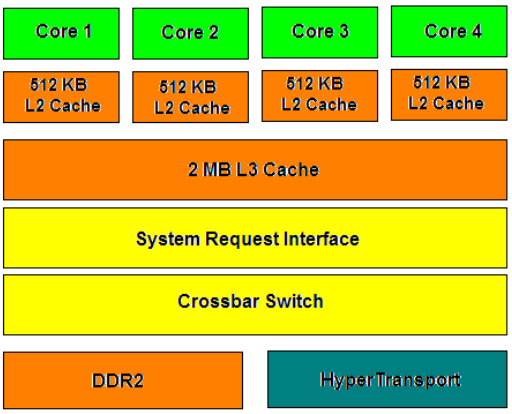


Figure 2.10: AMD Phenom Quad-Core Architecture. With data from [**?**]

Main aspects of GPU architecture

The GPU is different in that it focuses the use of the majority of its transistors for data processing and less on cache and flow control. This comes from the nature of the tasks one can perform on the GPU. Since it is mainly used for image processing one tends to perform similar operation on all pixels to be displayed. Pixels are independent from one another and so they can be processed separately, which explains the use of SIMD architecture. The main differences in the GPU and CPU is the use of memory and the access patterns used to access them. On the GPU memory is accessed coherently in the sense that when one pixel reads or writes a value to memory the next/neighboring pixel will do so as well in a few cycles. This means that by arranging the memory correctly one can hide the time of memory access with computations. Because if this simple access pattern the need of a complex flow control disappears resulting in that more transistors can be used for data processing. This is of course both an advantage for tasks that are parallelizable, but it is also limiting for tasks that demand instruction level parallelism or a lot of non coherent memory access.

The modern GPU is a mixture of programmable and mixed-functions units, and all programmable units in the graphics pipeline now share a single programmable hardware unit. Graphical processors differentiate themselves from the computational processors in that they focus on high throughput and of many parallel process with low latency execution of a single processor. Quite often in scientific and multimedia application one is to preform similar operations on many data. The GPU supports a tremendous amount of threads that execute similar operations in parallel. The combination of low cost, high performance and programmability of the recent GPUs make them an attractive approach in an HPC context. These progressions and developments lead to the GPGPU general purpose graphical processing unit.

Since the GPU is at first and foremost a graphics rendering hardware it is natural that the first attempts to program on it used the graphics API, such as CG, which introduced challenges since the programmer had to familiarize onesself with the API and the API also introduces overhead in communication with the GPU. The interest in being able to program on the GPU in a non graphical context have lead to the development of several programming platforms that overcome the delays of the graphics API and give the programmer more control. Examples of recent platforms are OpenCL, CUDA, Brook for GPU, etc. In this project I will be focusing on the use of the Compute Unified Device Architecture (CUDA), a programming model for GPGPU developed by NVIDIA.

2.8 CUDA Programming Model

CUDA is a parallel programming architecture and model, which includes a C complier plus support for OpenCL and DirectCompute, see Figure **??**. It is designed such that it naively supports multiple computational interfaces such as standard languages and APIs. The main intention of this programming model is to simplify

programming on the GPU. When using CUDA to program one will avoid the overhead of programming using a graphical API such as CG. Another feature worth mentioning here is how CUDA enables the programmer to directly program to the GPUs components such as direct memory access, using the shared memory and specifically manipulating threads.

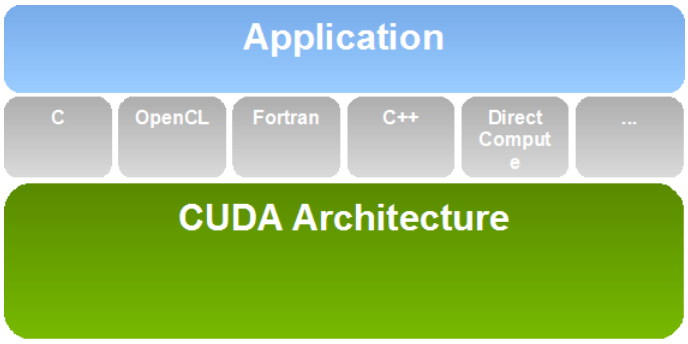


Figure 2.11: Image illustrating CUDA architecture from [?], with permission from NVIDIA

One has the option of running CUDA as a C extension, such that those familiar with C programming can easily use it to accelerate their code. As mentioned earlier the GPU functions as an accelerator and is therefore used with the CPU where one can run non parallelizable code in a program on the CPU and the highly parallelizable code on the GPU to accelerate execution and distribute the workload. The NVIDIA CUDA programming model consists of a "host" that is a traditional CPU, and one or more compute devices that are massively data-parallel co-processors (GPUs). Each device is equipped with a large number of arithmetic execution units that has its own DRAM, and runs many threads in parallel.

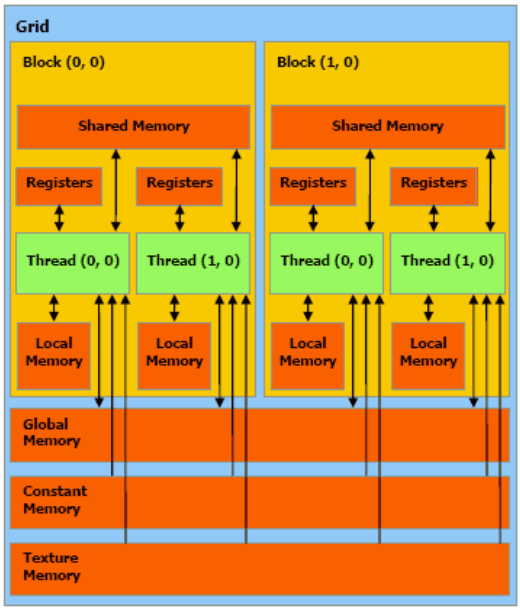


Figure 2.12: Illustration of the CUDA memory hierarchy from [?], with permission NVIDIA

To invoke calculations on the GPU one has to perform a kernel launch, which is basically a function written with the intent of what each thread on the GPU is to perform. The GPU has a specific architecture of threads where they are divided into blocks and where blocks are divided into a grid, see Figure **??**. The grid has two dimensions and can contain up to 65536 blocks in each dimension. While each block contains threads in three dimensions, and can contain up to 512 threads in two dimensions and 64 in the third. When executing a kernel one specifies the dimensions of the grid and blocks to specify how many threads will be executing the kernel.

A mention has already been made about the GPU having its own DRAM and that this is used in communicating with the host system. To accelerate calculations within the GPU itself there are several other layers of memory such as constant, shared and texture. Table **??** shows the relation between these. Here the ocus is on showing the different memory options available, and to map their attributes. For more details regarding CUDA and how to program on it read the NVIDIA programming guide [**?**]

Table 2.1: Table showing CUDA Memory Hierarchy, with data from [**?**]

Memory	**Location**	**Cached**	**Access**	**Scope in Architecture**
Register	On-chip	No	Read/Write	Single thread
Local	Off-chip	No	Read/Write	Single thread
Shared	On-chip	No	Read/Write	Threads in a Block
Global	On-chip	No	Read/Write	All
Constant	On-chip	Yes	Read	All
Texture	On-chip	Yes	Read	All

CHAPTER 3

3D Filter Implementations

This chapter will focus on our implementations of convolution on modern CPU and GPUs, and the framework on which they run. The details performed throughout the project and the main ideas behind the way the implementations perform will be discussed. There will also be explanations of how code has been optimized to run on the different platforms given their structure and architecture. The platforms experimented on in this project are running a single core CPU, a Quad-Core CPU, a single GPU and multi-GPU. All the 4 implementations of convolution on the different platforms will be presented in details in this chapter. One can also find details on how these implementations have been tested within a certain framework, which produces the convoluted images both before and after they have been processed. This framework has been supplied by Schlumberger and later modified to meet the needs of this project.

Schlumberger being the sponsor and co-supervisor of this project have set some rules to make the project more challenging and explorative. As convolution is a common technique to filter images, there have been several implementations and projects that address optimizing it. Even NVIDIA has an example of how to convolute on the GPU in their CUDA SDK [**?**]. But, most implementations are concerned with images which are two dimensional, and they usually optimize by separating the filter to make memory optimization easier. To set some boundaries in this project only three dimensional filtering is considered, and the filtering mask used is to be non-separable. This introduces challenges in memory since there are many jumps that occur when reading and writing three dimensional data. Something that can be avoided when using a separable filter and running the filter once in each dimension.

Another aspect to address in this project is the use of large data-sets, something that is uncommon for the GPU, The impact of several disk accesses on the execution time, and how one can block data to optimize disk access. The main condition is that the data to be filtered does not fit into memory and therefore it must be blocked for several disk access. This concerns both reading and writing to disk. A small problem is Introduced in edge cases where one must include more data from a neighboring block such that a border effect will be avoided and the filtered data will be more correct.

There are 4 main steps in the pipeline of filtering an image. When blocking, this

will be repeated for the amount of blocks. The steps are: create filter, read data block w/boarder values, convolute and update result table, write result table to file. These steps will be discussed in detail later on in this chapter, and so will ways to parallelize these steps to gain speedup including using several CPU cores, the graphical processor or several graphical processors.

Some other interesting aspects when optimizing code is of course the programming language used and how it is compiled to machine code. The framework introduced by Schlumberger is written in C# and during the project the use of C# has been considered and tested against the use of C. C is a more common language in the case of low level optimization. The implementations in these programming languages are mostly similar, but differ in the way they are compiled and run. C# code might even run on a virtual machine. Both implementations have been performed and run in Visual Studio, and if the preference is to use C# in the future one can even use the functions written in C as functions in a dynamic library and access them in a C# program. Meaning that it is not a problem to use C in the implementations, but it should be interesting to see how these languages differ in execution time. Therefore this will be tested and documented. The read and write algorithms are written in both C# and C. While all other implementations that proceed this are only written in C, as it is clearly more optimal and runs both faster and more stable, which will be shown in the next chapter with the results.

The Sections in this chapter are organized as follows. Section 3.1 shows which hardware is used to implement on. Section 3.2 explains the programming framework developed. Section 3.3 Explains how the 3D gaussian filter is implemented. Section 3.4 explains the methods used to read and write to disk. Section 3.5-3.8 explains convolution on the folowing platforms in teh following order uni-core CPU, quad-core CPU, single GPU, and multi-GPUs. Section 3.9 explains how tests are implemented.

3.1 Hardware and Software Used

While benchmarking it is worth noting that not only the algorithm has been modified to run more efficiently, but also the platforms it runs on have changed. The systems used for testing have in common are the use of a 64 bit system, which enables the addressing of more than 4 GB of memory. All implementations except the one where 4 GPUs are used have been on the same system. The reason for this is the fact that the Tesla s1070 is a rack solution that has been put together on a machine that can only be accessed remotely.

Table 3.1: Table showing the system components used

CPU	Phenom X4 2.81GHz
RAM	8 GB DDR2 memory
Disk	Samsung 500 GB 7200 rpm Disk
GPU1	NVIDIA Geforce 9800 GTX
GPU2	NVIDIA Tesla c1060

The main system where the majority of the tests are run on the Windows 7 64-bit operating system. The CPU in this system is a AMD Phenom X4, with 4 cores, three levels of cache where the level 3 cache is 2 MB large and is shared by all the cores. The level 2 cache is 512 KB large and is private for each core. This is one of the high end processors from AMD on the market (until Phenom2 will be released) and it supports the newest SSE instructions SSE4a. The system also has 8 GB of DDR2 memory. The GPU used in this project is a Tesla c1060, which is created for scientific calculations and has no video output. That is why it is combined with a GeForce 9800 GTX. The GeForce card is used for video output only, while the calculations are done on the Tesla card. The Tesla c1060 has 4 GB of global memory and is of the 1.3 generation of NVIDIA cards. The Tesla s1070 system that is used to run on multiple GPUs is basically a combination of 4 Tesla c1060 on a rack solution. Below one can find all relevant tech information on the hardware used.

Table 3.2: Table showing AMD Phenom X4 details [**?**]

Addressing	Supports 64 bit
L3 cache size	2MB(shared between all cores)
L2 cache size	512KB
L1 cache size	128KB
Memory bandwidth	33.8 GB/sec peak
CPU frequency	2.8 GHz

The machine where the Tesla s1070 is running is operated by a Linux operating system. The machine is an Intel i7 extreme system with 12 GB of DDR3 memory. This is obviously a better system than the AMD, but since there is an interest in developing in C# from Schlumberger it was not considered compatible in the begining because of the operating system. After the realization of using C# would only be limiting, a change was made to use C instead, but optimization were already made for running on the AMD processor. The change of platform would require a re-implementation of the CPU code and therefore the AMD, even though being inferior, was used to benchmark the CPU code. The AMD machine is also more available since one does not have to use it remotely and can reboot and tweak with ease. A future development in this project should consider using the i7 extreme and compare it to the GPU results.

Table 3.3: Table showing NVIDIA GPUs details [**?**]

GPU model	NVIDIA Tesla c1060	NVIDIA Tesla s1070
Num. cores	240	960 (4 x 240)
Core speed	1.3 GHz	1.3 GHz
Global memory	4 GB GDDR3	16 GB GDDR3
Memory bandwidth	102 GB/sec peak	102 GB/sec peak
System I/O	PCIexpress	PCIexpress

Using the AMD processor also has some positive features such as a profiler that can be integrated into Visual Studio 2008 Proffesional Edition, making it easy to

optimize for cache and test the developments of cache misses. The results of the AMD profiler [?] will be included in the sections regarding results of the convolution implementations.

The use of different operating systems gave some varying results in I/O time. Since it is common that operating systems buffer their disk access and after running a test several times this could result in that the operating system becomes better at retrieving the specific data resulting change in I/O time. If this feature were stable then one could continue to use it, but it is not. That is why it has been turned off such that a stable I/O time can be registered. In the final system it is recommended that it be turned on because it is averagely faster.

3.2 Programming Framework

At the start of the project a simple framework to read and produce bitmap (BMP) images from disk was made available by Schlumberger. The framework produced sliced two dimensional images of a three dimensional seismic cube from 3 different views. It produces all images of (x,y) dimensions along the z axis, all images of (x,z) dimensions along the y axis, and all images of (y,z) dimensions along the x axis (See Figure ?? for an illustration.)

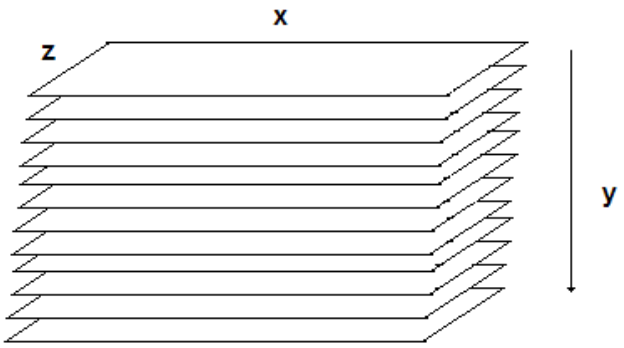


Figure 3.1: Image illustrating slices of x,z dimensions along the y axis

The major drawback with this framework is that it is too time consuming to produce all the images to black box test if the results are successful. Instead modifications where made to the methods such that they produce requested BMP images that are specified as a parameter in the functions. To produce all the images is just a waste of time because usually one only needs one image in each dimension to see if the convolution is working successfully. The original algorithm has low performance because it accesses the disk to read one byte at a time and when producing images of large dimensional scale this would mean several million disk accesses. Since the production of images is considered to be outside the scope of this project, because it

is not involved in the convolution process, the only modification made is the selection of single images and no optimization was done to these algorithms.

The framework is also modified to produce three dimensional cubes such that one can test for several sizes of data, which is crucial in this project given the large data set requirement. The data produced is similar to a three dimensional chess board pattern, but follows the format produced by Schlumberger. This is good for testing because of two aspects: One, it is easy to imagine what is expected of the results unlike seismic data which is more scattered data, and second it is better to see the results of a Gaussian filter on sharp edges and transitions (from a dark to a brighter color). The chess board pattern has sharp transitions in all three dimensions making it easy to check for correct behavior of the convolution algorithm. Since the datastructure is similar to that of 3D seismic one can easily interchange later.

To summarize, the framework is mainly used to produce images from bin files located on the disk and to produce bin files with three dimensional data such that it can be used in the convolution and filtering algorithm. Both these functionalities have not been optimized because they are not considered as the main scope of this project, and are rather tools created to help in the testing and development process. This means that they are not accounted for in the execution time and benchmark.

3.3 3D Gaussian Filter Implementation

Part of the convolution process is to use a filter mask that will convolute with the original data. Which mathematical distribution one chooses in the filter generation does not effect the execution time much since the result is a set of discrete values in a three dimensional cube, which is only calculated once for the whole execution. In this project, with the agreement of Schlumberger, a Gaussian filter is chosen for the implementation. A Gaussian filter is used to blur images and therefore the results expected on the images to be produced should be a blurry chessboard pattern where all the sharp edges are now gradients between the light and darker color.

$$G(x, y, z) = \frac{1}{(2\pi)^{3/2}\sigma^{\frac{1}{2}}} e^{-\frac{x^2+y^2+z^2}{2\sigma^2}} \tag{3.1}$$

To produce a Gaussian filter one can use a three dimensional Gaussian distribution and calculate discreet values for each element in the array of the filter. The mathematical formula for the three dimensional discrete distribution is **??** [**?**]. The problem here is that the distribution is infinitely long and most of the values calculated after 3 standard deviations are close to zero. That is why one only needs to calculate values up to 3 standard deviations in each dimension and truncate the other values from the filtering mask because their contribution will be minimal. In this project the Gaussian filter is calculated up to 4 standard deviations in each dimensions to obtain an even more precise filter and results from the convolution. The code implementation of the filter in C is as follows:

```
1  void createFilter(){
2    const float PI = 3.14159265358979323846f;
3    int len = standardDiv * 4 + 1;
4    filter = (float *)malloc((len*len*len)*sizeof(float));
5    float temp;
6
7    for (int i = 0; i < len; i++) {
8      for (int j = 0; j < len; j++) {
9        for (int k = 0; k < len; k++) {
10         temp= 0;
11         temp= (1 / (pow(2 * PI, (float)1.5) * sqrt(standardDiv)));
12         temp= temp* (pow(exp((float)1), -((pow((i - (standardDiv * 2)),
               2) + pow((j - (standardDiv * 2)), 2) + pow((k - (
             standardDiv * 2)), 2)) / (2 * pow(standardDiv, 2)))));
13         filter[(len*len*i+len*j+k)] = temp;
14       }
15     }
16   }
17 }
```

Code in C for calculating Gaussian Filter values

3.4 Reading from and Writing to Disk

General disk access is one of the most bandwidth bound processes in this project where one transfers data from the disk to buffer to the CPU for calculations and then writing the results back to disk. The data is of course passed through levels of cache and memory which are faster at transferring data. This implies that disk access should be minimal. In this project the data to be filtered is larger than memory available, 8 GB, that it needs to be separated and transferred one block at a time. It is then important that the blocking is done such that one does not need to jump back and forth on the disk to read and write data, but rather that the data is aligned and is read sequentially, for optimal reads.

In this implementation two files are used one to read from and another to write to. They both have the same layout and format such that one can use the result file to read as well if it is preferred. Both have the same header information in the same format as suggested and used by Schlumberger. After the header the data is presented in the form of a one dimensional line of floats. The data is to be read such that each row is read until the count of rows and then several rows are read until the count of columns, and this is to be repeated until the count of depth of the cube. This is how it can be represented as a triple for loop. But the data on the disk is stored aligned as a one dimensional array of float values.

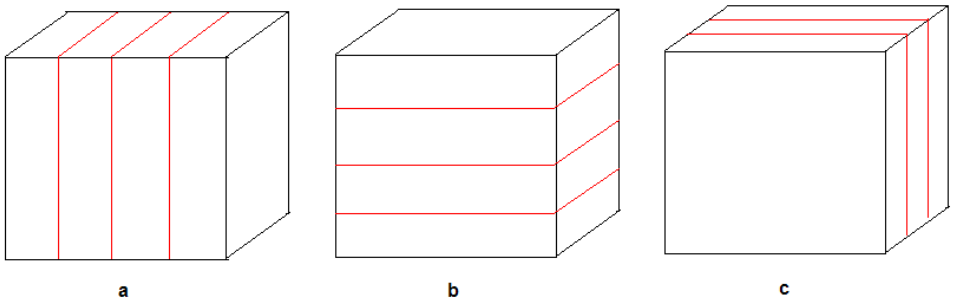


Figure 3.2: Image illustrating blocking. (a) across x dimension, (b) across y dimension, and (c) across z dimension

Since there are three dimensions, then one can block data across all three dimensions. To figure out which method is most useful, blocking is done across each dimension one dimension at a time and then tested to see the rate at which data is read and written (see Figure **??**). This is also compared to reading and writing one float at a time just for comparison, and to illustrate how crucial alignment can be. The read and write methods in the developed framework only need to know what to read and the blocking parameters that are sent in as a pointer. While in the main method for-loops are used to iterate over the whole cube (See code segment below).

```
1 void readCube(FILE *file){
2   _fseeki64(file, fileOffset, SEEK_SET);
3   data = (float*)malloc(blockLength * sizeof(float));
4   fread(data, sizeof(float), blockLength, file);
5 }
```

Code in C for reading a block of data from bin file

In both C and C# versions of the code, predefined functions are used to read and write to files on the disk. These methods need to know the path of the file to be read, a pointer in memory to where the data is to be read, a mode in which to open the file such as binary, and the amount of data to be read. The read method used can be found on line 4. To set an offset as to where to read and write in the file, one can use another built in function to seek in a file. This can be found on line 2. Notice that for large data set the conventioal seek function cannot be used because it does not address 64 bit of memory. The use of the offset is important when blocking such that one can set the pointer to where one wants to continue reading from disk as the next block is about to be read. For more documentation on these functions see Microsoft MSDN [**?**]. Similar functions where used in the C# version of this code. In both cases the data is read into an array in memory that will further be used in the convolution algorithm.

As mentioned before when blocking the data one needs to consider the shared data between blocks, which grows as the filter size grows. But, is overall of little significance since there is such a vast amount of data to be processed. To include this data in the borders one can just adjust the offset sent into the read method to read

earlier and stop later in all dimensions depending on the size of the filter. We will have a closer look at this later when discussing the convolution algorithm.

3.5 Convolution Implementation on Uni-Core CPU

Convolution in the spatial domain is calculated per pixel by multiplying corresponding values in the filter and the data set and adding them together, then dividing by the number of elements in the filter. See Figure **??** for an illustration. This is done per point in the data set. It is a way to calculate how all neighboring pixels effect the pixel one is looking at now given a mathematical distribution. One cannot perform convolution in place because this would effect the results for the neighboring pixel when calculated later. Therefore a result table must be used to contain the updated values.

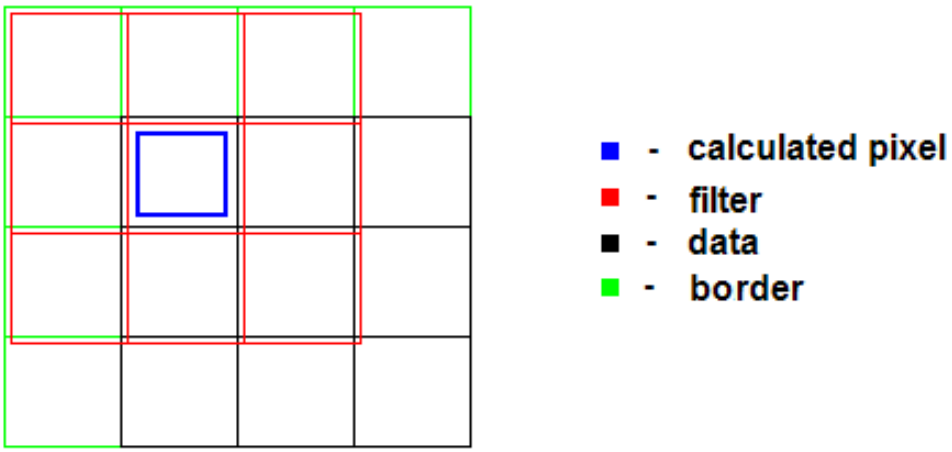


Figure 3.3: Image illustrating the different pieces used in convolution

Having a result table is an issue for large data sets because this means that one can only use half the memory available to read from disk and the other half for the result table. Another limitation as to how much data that can be read at a time is that the operating system usually uses some of the memory and it may even vary depending on the processes running on the machine. The memory available in this implementation is 8 GB. Where 1.5 GB is used for the operating system, that is why only a maximum of 3GB data is read at a time, which correspond to 6 GB allocated in memory because of the duplication of the table to calculate results.

When reading several blocks of data to be convoluted it is crucial that one accounts for the border values that should be read as well to avoid border effects between the blocks read. The size of the border is dependent on the size of the filter since when the outer most point is to be filtered then half the filter, in all dimensions, will be outside the frame of the data. This is why one must have a boarder along

the whole data set as well as between all the blocks of data. This means that there is a boarder along all edges of the data set and between each block, and that the border size is regulated by the filters size.

In the case where the filter to be convoluted is separable, then there is no need to optimize memory if the filter is kept constantly in the L1 cache then just normal traversal through each dimension would be optimal with few cache misses. When it comes to non-separable filters in two or three dimensions then one must jump in memory to reach the next row of data, which means that the pre-fetcher of the controller, using the SSE4a instructions [**?**], should retrieve the next row into the cache to perform the calculations. The pre-fetcher is able to get the next row as long as the stride is constant for each read from memory because it will be able to understand the pattern. In this case it is always constant, meaning that the rows needed are able to be pre-fetched without any problem. Cache misses can be checked using a computation profiler [**?**], which will be shown later in the result chapter. The filter size also regulates the amount of computation performed per pixel. Since as mentioned one is to traverse the filter size and perform calculations using all the values in the filter array. This means given that one has a 5x5x5 filter one would perform 125 multiplications and additions per pixel to in the cube. A 3x3x3 filter would give 27 multiplication and additions, and so on.

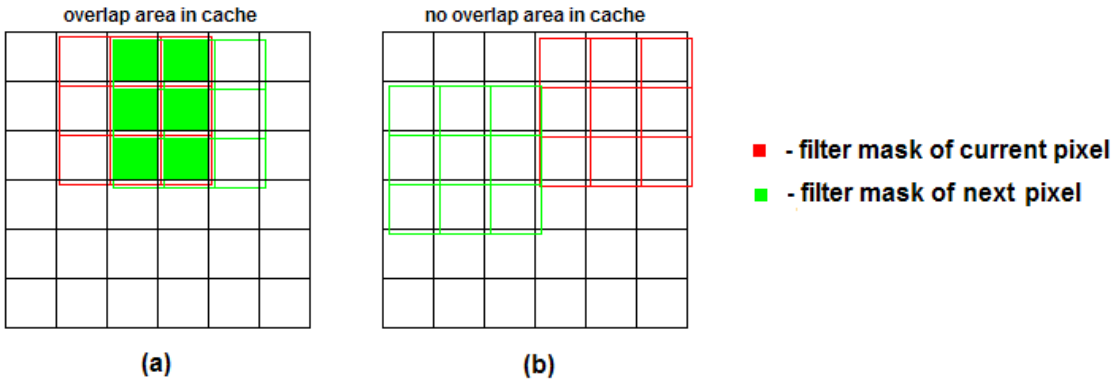


Figure 3.4: image illustrating overlap in memory between calculations of neighboring pixels

If one is to optimize for this type of data then one should do as many calculations as possible while data is in the cache. See Figure **??** to see how data from one pixel overlap with data for the calculation to the next pixel. This means that a lot of the values needed in the next calculation are already in the cache, but once one traverses to the end in the x direction then the data that will be needed for the next calculations have no overlap from the previous. Such that all the values in the cache need to be changed. Since all data is aligned and all jumps in memory have similar strides, this should result in the pre-fetcher being able to optimize against cache misses because the strides are constant. This can be shown in the profiler where this resulted in a cache miss percentage close to 0 %. This shows that there

is no more a need to optimize for memory and cache misses beyond this point.

```
1  float counter = len*len*len;
2  float imagevalue;
3  //performing convolution
4  for (int i = 0; i < z; i++) {
5    for (int j = 0; j < y; j++) {
6      for (int k = 0; k < x; k++) {
7        imagevalue = 0;
8        for (int l = 0; l < len; l++) {
9          for (int m = 0; m < len; m++) {
10           for (int n = 0; n < len; n++) {
11             imagevalue += token[(len+x)*(len+y)*(i+l) + (len+x)*(j+m) +
                   (k + n)] * filter[(len*len)*l + (len)*m + n];
12           }
13         }
14       }
15       data[y*x*i+ x*j + k] = imagevalue / counter;
16     }
17   }
18 }
```

C code for three dimentional convolution

3.6 Convolution Implementation on Quad-Core CPU

When parallelizing on the CPU where one has 4 cores one would have a shared memory system, this means that one does not need to use communication time on distributing data. Rather the read and writes can occur simultaniously to different addresses, and when it comes to calculations the work load can be distributed such that each core gets equal work that can be retrieved from the shared memory (L3 cache). On the system used in this implementation there is a 2 MB shared memory L3 cache that all the cores share. They have equally large level 2 caches that are local per core and have the size of 512 KB. This way when a portion of the data is sent to the shared memory it can be equally distributed and calculation in convolution are independent and therefore each core can perform calculations without having to wait. When all cores are done calculating, then the shared memory can be updated such the next 2 MB can be calculated.

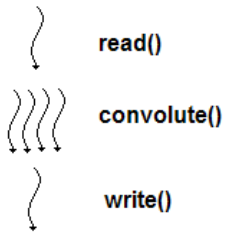


Figure 3.5: Image illustrating the amount of threads running when calculations are performed

In this implementation OpenMP was used to parallelize the code. 4 threads where used since there are 4 cores such that each core can run a thread. The directive used was "#pragma omp parallel for", which is the one that unrolls for-loops and dedicates separate instructions for each core to perform. It is also specified that the calculated variable be private in each thread. The calculated value for each pixel is stored in a temporary variable before inserted into the result array , and as such it is kept private for each thread such that they do not all write to the same variable. The filter on the other hand is placed in constant memory and is read by all.

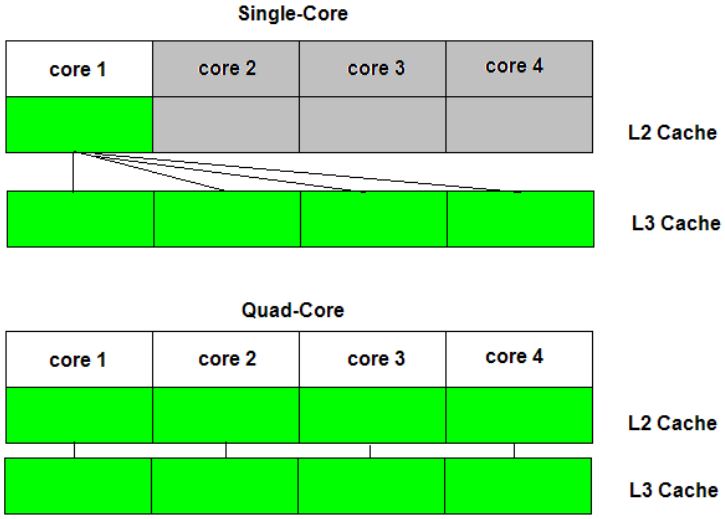


Figure 3.6: Image illustrating the work distribution on a Quad-core compared to a single-core

When it comes to optimizing caching and memory, the use of already optimized code for one core duplicated to several cores sharing memory is already optimal. See Figure **??** that explains how calculations are just distributed in the same optimal fashion. This means that other than making sure that some critical variables are kept private for each thread no other optimizations were made to the implementation. Given that reads and writes are done sequentially the expected speedup compared to running on one core should be close to 4. But since some parts of the code are non-parallel, like I/O, it should be a bit less.

```
1  float counter = len*len*len;
2  float imagevalue;
3  #pragma omp parallel for private(imagevalue, filter)
4  for (int i = 0; i < z; i++) {
5    for (int j = 0; j < y; j++) {
6      for (int k = 0; k < x; k++) {
7        .
8        .
9        .
10     }
11   }
12 }
```

C code with OpenMP for three dimentional convolution

3.7 GPU Implementation in CUDA

As mentioned before, the GPU is an accelerator and therefore it must cooperate with the CPU. This introduces some delays in sending the data from disk to the global memory of the GPU, Since it has to go through the main memory and be streamed over to the GPUs memory. There is also a delay after the computation is done since the data has to be sent back to main memory from the global memory of the device (GPU). This means that there is some extra work that has to be done in order to perform computations and may result in no speedup in the case where little computation is needed. The communication between CPU and GPU is done across a PCI-express buss which has a limited bandwidth of about 6GB/s, which is less than optimal. But, since we are looking at large data sets and a computationally heavy algorithm this communication time can be shadowed by computation time. Meaning that the use of the GPU should be good for performance in this case.

When using the GPU for calculations one uses several threads running on the device to perform calculations. There is a time constraint of 5 seconds that should not be exceeded by a kernel invocation, which means that the tasks performed per thread should be of a simple nature. In this case each thread can be performing calculations on one pixel of the data cube at a time. This should be a simple task even for a larger filter. Here similarly to when running threads on the multi-core CPU the memory is shared by all the threads. Thereby making it easy to allocate global memory, transfer the data there, invoke the kernel and get the results.

The main bottleneck in this procedure is the transfer of data to and from the GPU since it is limited by the capacity of the bus and is surely slower than the rate data can be retrieved from the global memory for calculations on the GPU, 102 GB/s [**?**]. That is why it is a good idea to transfer as much data as possible at a time and then invoke the kernel several times to perform all the necessary calculations. Here as well we have the issue that convolution needs a equally large array to store its results therefore forcing the use of only half the memory for data transfer at a time. When it comes to the memory usage internally on the GPU one has several options. One could allocate the data in the shared memory, which is useful if one is to reuse data between threads. Or one can use the texture memory, which is optimized for two dimensional texture calculations. In this implementation none of these intermediate memory banks are used because of the lack of communication between threads given the nature of the task. The threads do not need to share memory in order to solve their tasks since there are no dependencies. The filter, on the other hand is located in constant memory since it will not be changed during the process and such that all the threads can read it. It is smart to use the constant memory becaused it is actually cached. The only limitation is that it is read only by the device. This is not an issue because one is only interested in reading the filter values and not change them during execution. Another interesting aspect that should be noted is the fact that the best case lookup in constant memory is one cycle, which is similar to shared memory.

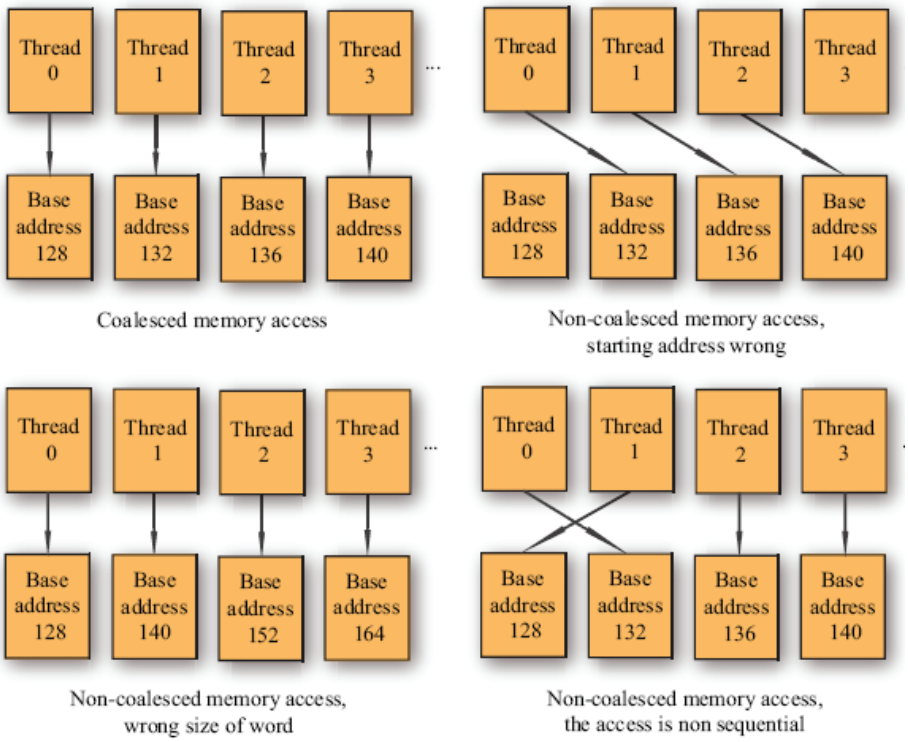


Figure 3.7: Image illustrating memory coalesced reads [?]

In this implementation the amount of threads created per block has been decided with the use of the occupancy calculator developed by NVIDIA [?]. To use the calculator a cubin file was compiled with the use of the CUDA file containing the convolution kernel. The cubin file can be opened as a text file and contains relevant information such as the use of shared memory and the amount of registers used. After inserting this information into the occupancy calculator it indicated an amount of 100 threads would give the most occupancy, and was at 63%. The bottle neck here is the amount of registers available per thread, and since one needs 24 to run the kernel one cannot run it on all the cores available on the GPU. The reason being the maximum amount of registers available in total. One could force the compiler to use only 16 threads, but this will force the threads to save intermediate data to global memory for calculations, which will greatly decrease performance. Given that 100 threads would give the maximum occupancy in this case means that one can have several blocks. Since the data is in three dimension the grids two dimensional nature was combined with one dimensional threads to access all three dimensions in the problem. This is a similar approach to the one performed in [?].

To copy data over and from the global memory of the GPU standard CUDA functions where called, and even when allocating memory on the main memory on the host CUDA non device functions where used. The reason for this is because it optimizes the transfer and locality of the data read from the disk to match that of the GPU device. This is because of the two dimensional nature of the device memory [?]. When the data is in global memory and the threads are calculating each their pixel the data locality, by default, will have them avoid writing to the same address in memory avoiding bank conflicts and it will also have them read data coalesced.

Since each neighboring thread is working on the neighboring pixel in the result array they will never write to the same value and avoiding bank conflicts. And since each thread will start to read values with an offset of 1 from its neighboring thread they will be reading data coalesced.

Looking at the big picture, the data is now read from the disk to the main memory and then moved to global memory on the device before performing calculations on the GPU. Then the convolution kernel is called as many times as needed given the size of the sub cube in the x dimension (this is because using 100 threads is optimal [**?**]). Then data is written back to main memory and further on back to disk before the next block of data is read. The GPU used in this implementation has 4 GB of memory and so only 2GB of memory are read at a time because of the result table. This means that there are several reads from disk needed here compared to the CPU version where one could read 3GB at a time. This could be a possible source of delay since reading from disk and transferring data to memory is time consuming.

```
1  __global__ void increment_kernel(float *d_data, float *d_token, int len
       , int x, int y, int offsetnrx, int offsetnry) {
2    float imagevalue =0;
3    for (int l = 0; l < len; l++) {
4      for (int m = 0; m < len; m++) {
5        for (int n = 0; n < len; n++) {
6          float temp = d_token[(blockIdx.y+l)*( x + len)*(y + len) + (
               blockIdx.x+m + (gridDim.x*offsetnry))*( x + len)+ (
               threadIdx.x+n +(blockDim.x*offsetnrx))];
7          float temp2 = d_filter[(len*len)*l + (len)*m + n];
8          imagevalue += temp * temp2;
9        }
10     }
11   }
12   d_data[(blockIdx.y)* x * y + (blockIdx.x +(gridDim.x*offsetnry))* x +
         (threadIdx.x)+(blockDim.x*offsetnrx)] = imagevalue / (len*len*
        len);
13 }
```

Kernel code for three dimentional convolution

3.8 Multi-GPU Implementation in CUDA

In the case of using multi GPUs one is trying to hide computation time by running computation simultaneously on all GPUs. This is of course not quite possible because of the restrictions of the PCI bus connection when communicating and transfering data to the GPUs. In this implementation the system running has 4 GPUs available with 4 GB of memory each. The code to run and transfer data to the GPUs is run in threads with the help of OpenMP just like in the multi core CPU version. Here similarly to the quad-core implementation there are 4 threads running where each thread is responsible for a GPU and they have corresponding IDs.

The idea is to run each thread in parallel to invoke all the GPUs simultaneously. The only issue here is how the threads should share the bus. This needs some clever pipelining. In this implementation one thread at a time transfers data (both to and from the device) to utilize the buses capacity and not interleave data on the bus. Then by running the kernel invokation in parallel one can hide the transfer times for the 3 other GPUs since the first GPU is already doing calculations while the others are transferring data. This way the only overhead compared to running on a single GPU would be the the final transfers from the last 3 GPUs, which occur after the first GPU is done computing and transferring the data back. Here all reads and writes are done by a single thread and are as large as possible. To make sure that each thread was accessing the bus separately the use of the "critical section" directive in OpenMP was used. This works as a semaphore and avoids deadlocks and collisions in memory access. See Figure **??** for details around the pipeline. It is worth mentioning that the larger the calculations the less signinficant the cudaMemCpy() functions will be in terms of delay. Therefore the size of each block in the figure is not representative for larger sets, but should give an idea of the concept of hiding calculations by runnig the code in threads.

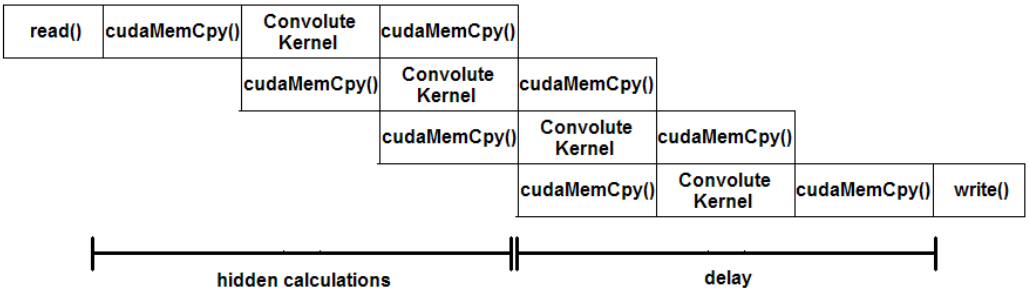


Figure 3.8: Pipeline representing running code on 4 GPUs

On the machine running the 4 GPU solution there is only 12 GB of main memory available, which should be an issue since the quad GPUs together have 16 GB of global memory. In theory one could not transfer all the data from one read of disk to memory to device memory. But, in the case of convolution and the need for a result table, one only needs to transfer 8 GB of data at a time and the rest should be allocated for the result array. Therefore in this case one read from disk would suffice to utilize all the memory available on the graphical processors.

Another issue worth mentioning is that there is a restriction in CUDA that if several devices are invoked. A CUDA *non-device* function cannot be called between every time one sets the device i.e chooses a device. The use of non-device functions, such as cudaMallocHost(), help because they optimize reads from host to device and not being able to use them means that one should use standard methods of allocating memory. This is a potential cause for less speedup because the memory is not structured for optimal transfer, but this is just one of the limitations of CUDA. One could potentially try to restructure the data without using the functions to utilize

for the GPU.

```
#pragma omp parallel default(shared)
{
    float *d_data;
    float *d_token;
    int th_id;
    int newtoken;
    int newdata;
    dim3 threads = dim3(100, 1, 1);
    dim3 blocks  = dim3(100, z/num_th, 1);

    th_id=omp_get_thread_num();
    cudaSetDevice(th_id);
    int device;
    cudaGetDevice(&device);

#pragma omp barrier

    cudaMalloc((void**)&d_data, d_dataLength);
    cudaMalloc((void**)&d_token, d_tokenLength);

    newtoken = (d_tokenLength/4)*th_id;
    float *temptoken = (float*)(token + newtoken);

    cudaMemcpy (d_token, temptoken , d_tokenLength,
        cudaMemcpyHostToDevice);
    increment_kernel(d_data, d_token, len, x, y, blocks, threads);

#pragma omp critical
{
    cudaMemcpy (tempdata , d_data, d_dataLength, cudaMemcpyDeviceToHost
        );
}
    cudaFree(d_data);
    cudaFree(d_token);
    cudaThreadExit();
  }
}
```

C code with OpenMP for three dimentional convolution on Multiple GPUs

3.9 Testing and Benchmarking implementation

The main aspects that need testing for this code are the time used for I/O operations and the time used for computations. These will then be compared between different versions of calculated speedup. It is important to see if the convolution algorithm is computationally dominant or if the I/O time is the main bottleneck here. If the I/O time is dominant, then there is no need to parallelize the computation portion of the code because the speedup will be minimal and limited by the time used on communication time.

In the implementation an include "time.h" is used for the function clock() to take the time. The function is used in the main method and the whole operation from start to end is timed. This involves creating a filter, reading data from disk, performing convolution, and writing to disk. When the IO time is measured then the functions to create a filter mask and convolution that are commented out. This way only the read and write times are measured. Then by using these values one is able to calculate the computation rate and how much of the time is used on computation, which will be discussed in detail in the next chapter.

```
1   clock_t start = clock();
2   readDims();
3   setOffset();
4   createRes();
5   createFilter();
6
7   int sub_ic = dimensions;
8   int sub_xc = dimensions;
9   int sub_vc = dimensions/numofblocks;
10  fileLength = sub_ic * sub_xc * sub_vc;
11  nbytes = fileLength * sizeof(float);
12  counterread= 0;
13
14  FILE *fileread;
15  FILE *filewrite;
16  fileread = fopen(name, "rb");
17  filewrite = fopen("C:\\test\\res.bin", "wb");
18
19  for(int i=0; i<numofblocks; i++) {
20    fileOffset = fileLength * sizeof(float) * i + originOffset;
21    readCube(fileread);
22    convolute(sub_ic,sub_xc,sub_vc);
23    writeCube(filewrite);
24  }
25
26  fclose(fileread);
27  fclose(filewrite);
28
29  clock_t stop = clock();
30  printf("Time elapsed: \%f\n", ((double)stop - start) / CLOCKS_PER_SEC
      );
```

C code for three dimentional convolution

CHAPTER 4

Results and Discussion of Benchmarkes

In this chapter the focus is on testing the different implementations and documenting the results. The method of testing is timing the execution time of only I/O and the total execution time. Since all parallelism is done in the convolution segment of the code, the timing of I/O is done separately and is considered as the I/O time for all implementations because they all use the same code for reading and writing operations to disk. The way in which time is recorded in the implementation is shown in section **??**. The complete data collection can be found in the appendix **??** in table form. In this chapter only the essential results are shown and discussed. Results will be represented in both graphs and tables.

For some implementations the time taken to perform convolution for larger sets of data takes several hours. Therefore a limit of one hour has been set such that if an implementation takes longer than one hour it times out. This is because each implementation is tested 10 times and the median execution time is noted to assure quality of the sampled data. In other words if an implementation takes one hour to execute, then it would take 10 hours to completely test. Because of the time constraints on this project one cannot test for several large sets if they are this time consuming. Yet the results are tests of several sizes that belong to the time range. The execution times are compared between the different implementations for speedup analysis, which proves to be sufficient in showing trends. This can also indicate if the implementations scale and how they scale given the problem size.

Later in this chapter, the focus will be on explaining and discussing the results obtained form the various implementations, exploring the reasons for the speedup obtained on the different platforms. There will also be considerations for possible improvements to gain even more speedup. This will be done by attempting to describe the way in which the algorithms behave and by comparing crucial elements of this behavior. This will involve in depth analysis of memory interaction and the way in which the executions are pipelined and performed. An analysis of the convolution algorithms on the multiple platforms will be considered.

The sructure of this chapter is as follows. Sections 4.1 and 4.2 are results and discussion of I/O. Section 4.3 shows the visual results of convolution. Section 4.4 shows the AMD and CUDA profilers results. Sections 4.5 and 4.6 are results and discussion of convolution implementations.

4.1 I/O and Blocking Results

As mentioned in earlier chapters blocking of the data cubes is necessary and in this project some experimentation has been made by trying out blocking across different dimensions. And this has been compared to the use of no blocking and just reading 1 float value at a time. The results show that as long as one reads across other dimensions than the x dimension, then it will result in good read and write rates to disk. It is also important to find a blocking method that makes it easy to add border values without jumping on disk. That is why a blocking across the z dimension is chosen. The results for I/O time given different blocking methods are shown in Table **??**.

Table 4.1: Execution time in seconds of reads given a data size and blocking method

Dimensions	No blocking	Blocking x axis	Blocking y axis	Blocking z axis
200x200x200	40	31	2	2
400x400x400	314	317	18	18
800x800x800	2533	2524	150	149

As mentioned earlier the I/O time tends to be unstable depending if it is buffered first or not and to stabilize it all buffering and operating system help is turned off. Then by benchmarking the C# version and C version one can see that the C functions are better at reading from disk. This is the reason as to which a shift to C was made. I/O is one of the slowest processes since it is communicated to at a rate of MB per seconds in contrast to memory which has a bandwidth of several GB per second. Saving time at this stage can be crucial.

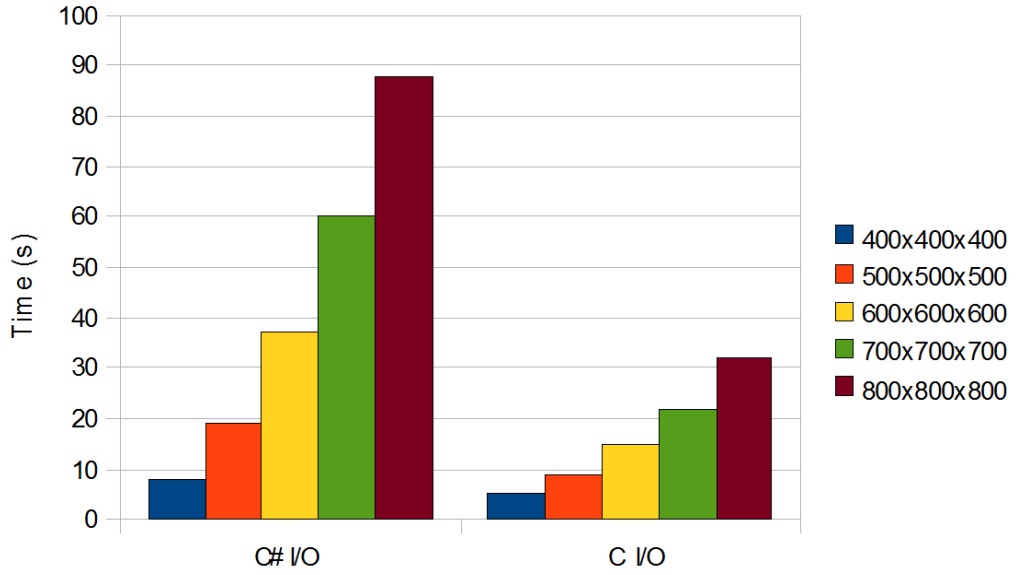


Figure 4.1: Execution results for I/O

When testing I/O operations different sizes of blocks were considered. The same sizes

are then used when filtering to compare execution times and calculate computation percentage of the different operations. Figure **??** shows a graph of the time for each cube run on different programming platforms.

4.2 I/O and Blocking Discussion

4.2.1 Disk Access

The data filtering pipeline starts with obtaining data from disk to buffer such that calculations can be performed on them. This is the slowest segment of the pipeline, which makes each disk access very costly. The disk bandwidth is typically about 70 MB per second. Comparing this to the memory bandwidth of the buffer which is typically 35 GB per second, each disk access is about 500 times slower. And when considering the vast amount of data one would expect from a seismic processing application then disk access should be done efficiently.

Looking at the results after having used blocking across the z axes I/O time resulted in being between 1-2 % in the case of a single core, and has the trend of being less significant as the data set grows. This means that the I/O time even through slow is still less time consuming than performing the convolution operations. This is good because it means that one can focus on optimizing computations.

4.2.2 Blocking

The reason for such a time efficient disk access is the type of blocking performed. This is also seen in the results. When the disk is forced to perform jumps before each read or write the performance of the I/O is decreased significantly, As seen when blocking across the the x dimension of the data set. The requirement for jumps is typically evident in large multiple dimensional data sets that need to be blocked. Given the data-structure of the seismic data by reading along the z axes one would not need to jump on the disk, but rather read a whole block of data at a time, which has several positive aspects. The first is that one only needs to access the disk once for each read to buffer. The second all the data is aligned as it should be when filtered, making it faster to transfer along the bus and easier to convolute. Thirdly, it is faster to transfer because when transferring a large data set of aligned data one will use the full capability of both the bandwidth of the disk and capabilities of the bus.

Blocking across the z axes makes it easier to fill in borders, because in this way only two border need to be filled such that the necessary data is present. This is because one only has two neighboring blocks along the z dimension, and not in the other dimensions, that need to exchange data. This is of course possible until a certain size of data. Because sooner or later there will not be enough memory to fit all the data in all three dimensions at a time and therefore one would have to block along another dimension. This can also be the case if one has uneven dimensions on the

cube. The rule of thumb here is to read as much aligned data from disk as possible. Given the data-structure of the seismic data one should try blocking across the z dimension first if this is not possible then a combination of blocking across the y and z dimension. Then for really large sets, for example several tera bytes of data, one could perform blocking across all three dimensions.

4.2.3 Effects of Operating System on I/O

Some unexpected results showed up when the first tests of I/O where performed. To begin with the tested I/O time was high and as the testing contimued it got shorter and shorter for each test. When testing a sample of ten tests is taken and a median is returned as the final result. The fact that the tests got shorter gave unreasonable results. The main reason for this is that the operating system has a way of buffering its reads and writes such that it can hide disk time from the user. This can be a useful feature if one is working on a similar set and reading it several times, but for this project a standardized disk access time has to be established such that a comparison between the distribution of computation and I/O time can be established. The feature of buffered I/O is of course useful, but is undetermenistic depending on if the data read is in buffer or not. It might give an average of better reads and writes, but is harder to establish a correct estimate therefore it is not used further in the project. In a real life situation it is recommended to have this option turned on.

4.2.4 Why Use The Chessboard Pattern as A Data Set?

The correlation between seismic data and the chessboard pattern used are. Since in this project a Gaussian filter is implemented it is easier to see the effects and test its results on data with sharp edges in all three dimensions. This is accomplished with the use of the chessboard pattern. The way in which the data is stored including the header of the file in which they are stored is a replica of the seismic data example given by Schlumberger. This means that the application can easily use seismic data instead of the chessboard pattern such that it is compatible with the data structure standards of Schlumberger. Since the result file also shares these standards it can also be used to run a second iteration for more blur effect.

4.3 Visual results of Convolution

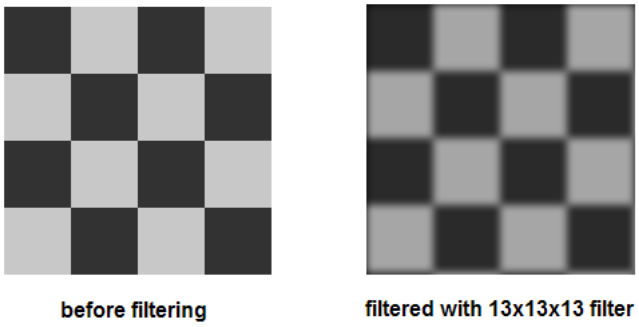


Figure 4.2: Image illustrating images of filtered data with filter size 13

The framework developed produces BMP images of slices along the different dimensions of the data cube. In Figure **??** one can see the results of filtering with a size 13 filter. The filter sizes used in testing are 5, 9, 13, 17 and 21. This is common for all results further on. The size of the filter introduces more work since it means that each pixel is now effected by more neighboring pixels, but it also enhances the effect of the filter. In this case the bigger the filter the better the blur.

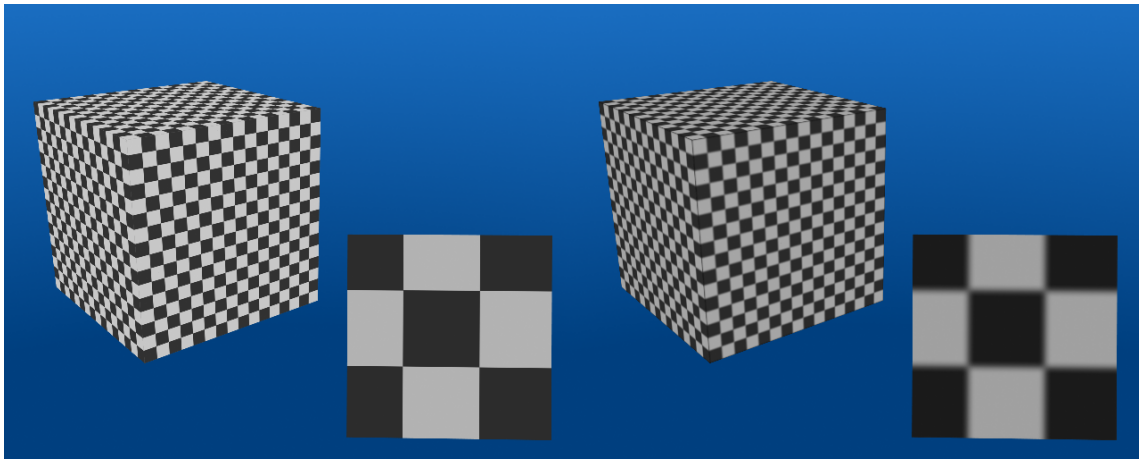


Figure 4.3: Image illustrating images of filtered data in 3 dimensions

Other than the standard two dimensional BMP images, there has been some experimenting with some other representation of the data. Such as a three dimensional representation of the cube before and after it is filtered, see Figure **??**. This might not be very interesting because the data of the cube here is not represented as a volume but rather a shell, since one can only see the outer most values. Another project this year is experimenting with ray-tracing and volume representation [**?**]. Using the volume representation code and inserting the filtered data into it gives

the Figure **??**, which shows a clear 3 dimensional blur effect. The filter used here is 21x21x21 floats large.

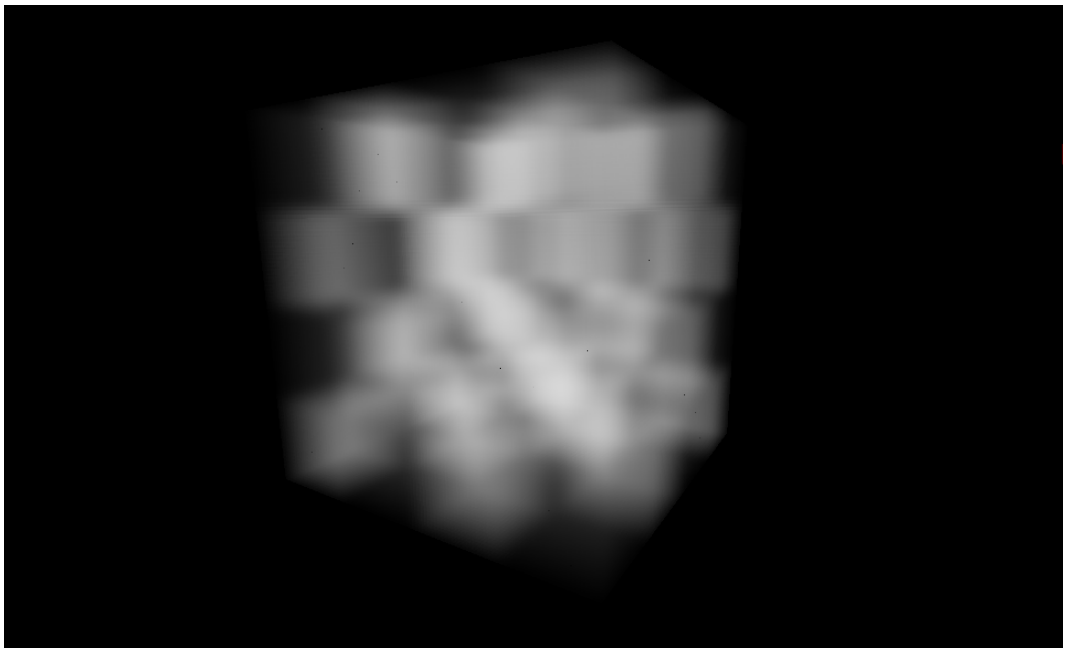


Figure 4.4: Image illustrating a ray traced volume rendering of filtered data. With permission from [**?**]

The concern of border effects has been repeated often in the report with special emphasis on how it effects the output data. Figure **??** shows the visual effcts of not accounting for border values. One can clearly see that the borders add noise and corrupt the values of the data such taht they are no longer valid.

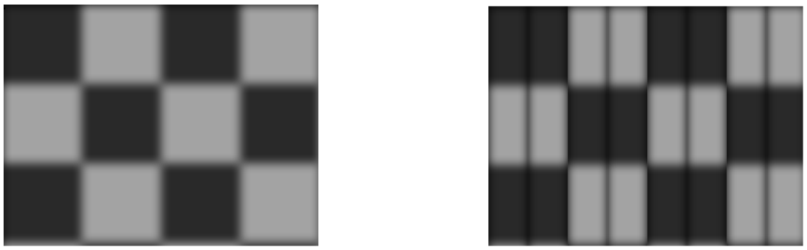


Figure 4.5: Image illustrating the border effect one can obtain if border data is not accounted for

4.4 AMD and CUDA profiler Results

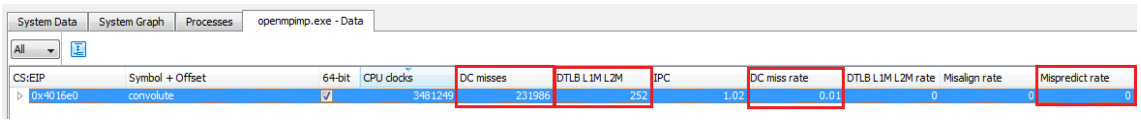


Figure 4.6: AMD profiler [?] results of running 800x800x800 cube with a size 13 filter on a single core CPU

The AMD profiler has been mentioned before in this report and has been used quite frequently while developing the CPU implementations as it helps to supervise memory and cache behavior. The profiler is developed by AMD [?] and is integrated in Microsoft Visual Studio 2008 Proffesional Edition. Below one can find Figures ?? and ?? taken from running the profiler and the results returned from both the single-core implementation and the quad-core implementation. Notice the close to zero miss rate for caches and prefetcher. The fields marked in these figures from left to right are memory misses, L1 and L2 misses, memory misses in precentage and the amount of mispredictions the prefetcher had during execution

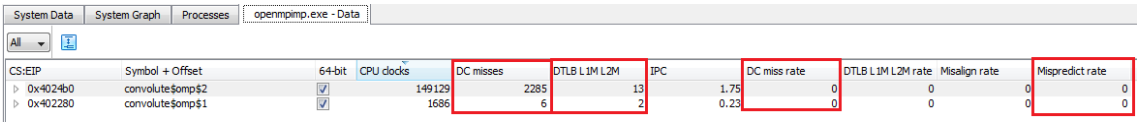


Figure 4.7: AMD profiler [?] results of running 800x800x800 cube with a size 13 filter on a quad-core CPU

Other than using the AMD profiler, the CUDA visual profiler was used to analyze the GPU executions. The CUDA profiler is Developed by NVIDIA to analyze CUDA code run on their GPUs. The Profiler is able to illustrate the time used for copying data from host to device and vice versa, and the time used on running kernels. In the case of the Convolution kernel 96 % of the time is used on convolution, which is not surprising since it is computationally demanding, and 4 % of the time on copying memory back and forth from host and device. This is illustrated in Figure ?? where the conv_kernel is the kernel used to convolute.

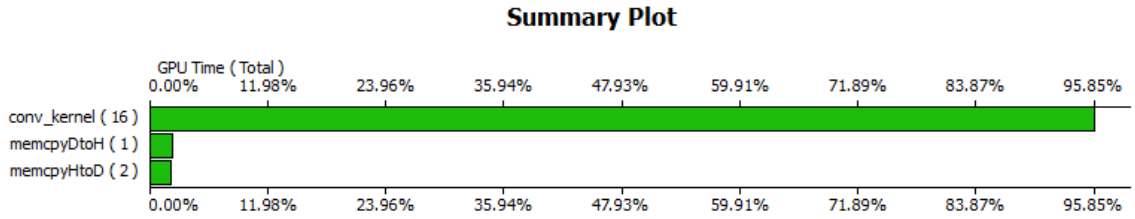


Figure 4.8: CUDA profiler [**?**] results of running 800x800x800 cube with a size 13 filter on a NVIDIA Tesla c 1060

4.5 Benchmarked Convolution Results

When it comes to testing convolution in contrast to I/O, the size of the filter has a major effect on performance. This is because the larger the filter, the more computation there is to do per pixel. Therefore it is not only interesting to increase the problem size, but also increase the size of the filter and compare on both levels. The filter sizes the majority of tests explore are 5, 9 and 13. One can clearly see the effects of using a bigger filter and the increase in computation time.

Two important factors for analysis that must be calculated from the results are speedup and computation percentage. When calculating speedup the total execution time of a serial implementation is devied by other parallel implementations. To calculate how large percentage of the time used in the execution is calculated by subtracting the I/O time from the total time and then dividing by the total time. This way one gets the percentage of time used to perform computation. The computation time is important because it is an indication of how much one can gain from parallelizing code. See Tables **??** and **??** for results for different platforms and data sets.

$$Speedup = \frac{execution\,time\,of\,serial\,algorithm}{execution\,time\,of\,parallel\,algorithm} \tag{4.1}$$

$$Computation\,percentage = \frac{(total\,exec\,time - I/O\,time)}{total\,exec\,time} \tag{4.2}$$

Because of the limit of 10 hour maximum execution time there are no tests for larger sizes than a 32 GB seismic block on the CPU. While on the GPU and multi GPU one is able to compute much larger seismic blocks, but only blocks up to 32 GB were tested for comparison reasons with the CPU. All data is blocked into 5 blocks to have a constant effect of I/O time on all implementations. This way one can focus on the effects of memory and cache usage. As one can see from the results in Tables **??** and **??**, convolution is computationally heavy and I/O is actually a minimal part. With large data sets and large filters on the single-core CPU almost 99% of all time used on computation. Therefore optimizing what occurs during computation is more important that the 1% time of I/O.

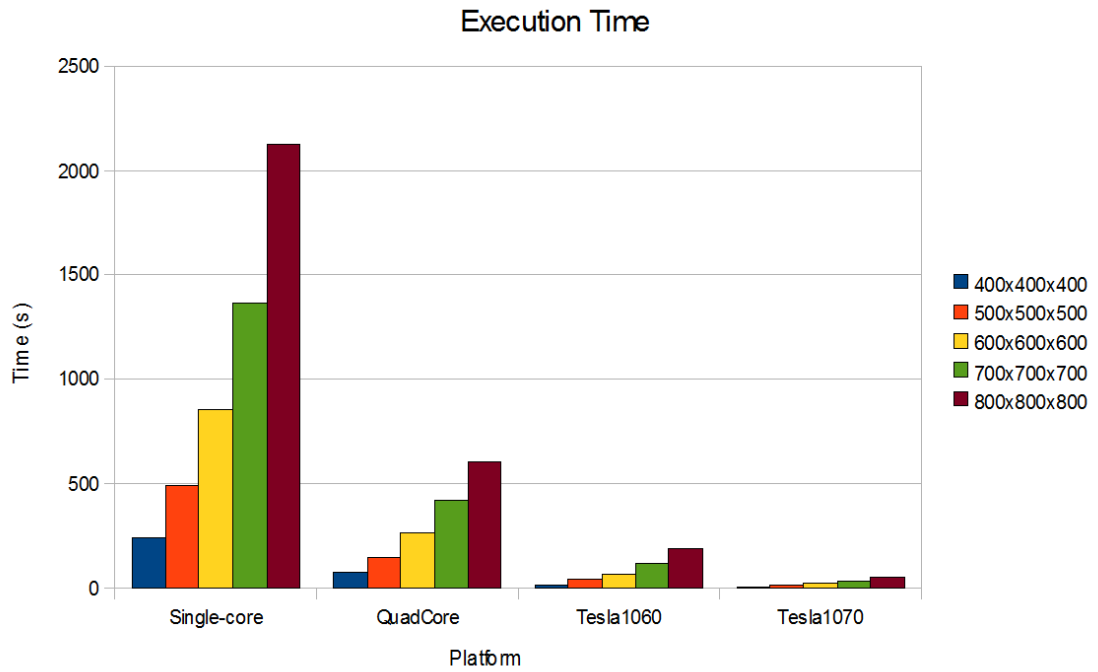


Figure 4.9: Graph illustrating execution time on different platforms for various data cubes

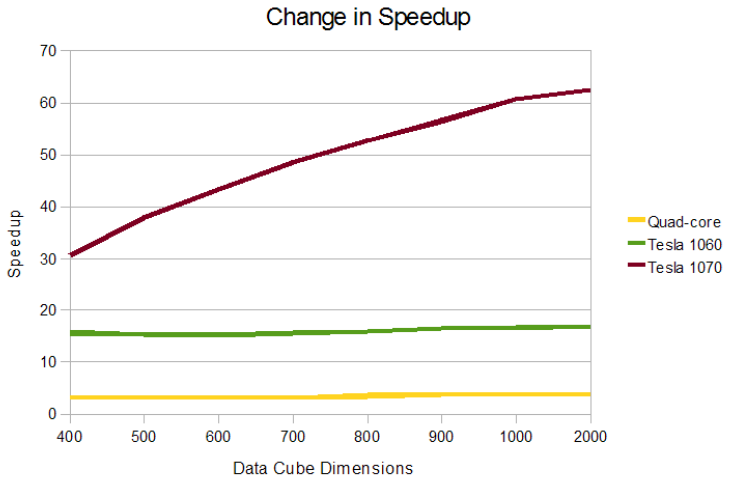


Figure 4.10: Graph illustrating speedup different platforms for various data cubes, values from Table **??**

In figures **??** and **??**, The values of Tables **??** and **??** are presented such that it is easier to see behavioural patterns. The results that are of main interest here are that most results are somewhat constant in both computation time and speedup for all platforms except the Tesla 1070. This comes from a distinct relation between transfer of data contra calculations, which will be discussed later. For larger data sets even the Tesla 1070 starts to converge and get a stable constant value like all other platforms.

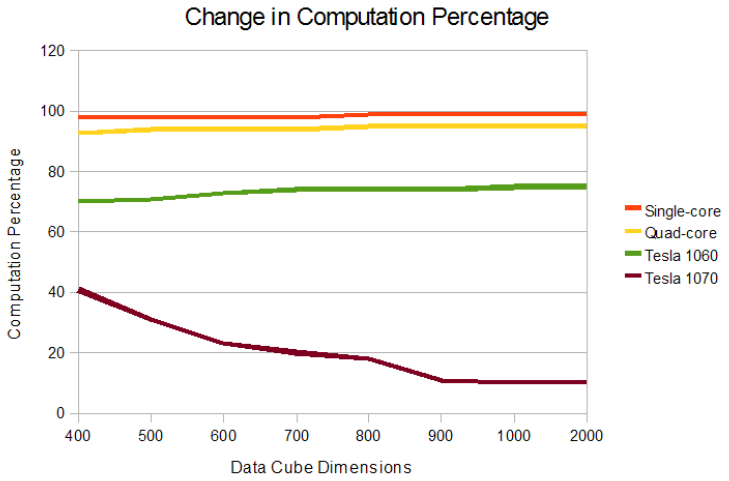


Figure 4.11: Graph illustrating computation percentage on different platforms for various data cubes, values from Table **??**

Table 4.2: Table of speedup for filter of size 13x13x13 given cube size

	Quad-Core AMD	**Tesla C 1060**	**Tesla S 1070**	
Dimensions	Speedup	Speedup	Speedup (CPU)	Speedup (GPU)
400x400x400	3.1	15.8	30.6	1.9
500x500x500	3.2	15.4	38.0	2.5
600x600x600	3.2	15.2	43.4	2.9
700x700x700	3.2	15.6	48.5	3.1
800x800x800	3.5	15.9	52.9	3.3
900x900x900	3.6	16.4	56.5	3.5
1000x1000x1000	3.6	16.6	60.8	3.7
2000x2000x2000	3.6	16.9	62.7	3.7

Table 4.3: Table of computation % for filter of size 13x13x13 given cube size

Dimensions	**Single-Core AMD**	**Quad-Core AMD**	**Tesla C 1060**	**Tesla S 1070**
400x400x400	98	93	70	41
500x500x500	98	94	71	31
600x600x600	98	94	73	23
700x700x700	98	95	74	20
800x800x800	99	95	75	18
900x900x900	99	94	74	11
1000x1000x1000	99	95	75	10
2000x2000x2000	99	95	75	10

4.6 Convolution and Optimization Discussion

4.6.1 What Makes Convolution Parallel Friendly?

Before attempting to parallelize an algorithm it is good to know that it is parallelizable. Convolution is an imaging technique, and like most imaging techniques one is performing similar actions on several pixels [**?**]. If these actions are independent as in the case of convolution, then one can run actions of each pixel in parallel with other neighboring pixels. This is actually part of the motivation behind the structure of the GPU and the way it is used (considering its SIMD architecture). Convolution also meets all the requirements of Bernstein's conditions (as mentioned in Section **??**) in that each pixel is completely independent of the others. Another interesting aspect in convolution is that calculations performed per pixel are actually quite time consuming in that each pixel must travers the length of the filter mask. The fact that being able to mask other computations behind each calculation should show great speedup. These are some of the motivation behind parallelizing convolution and why it is such a parallel friendly technique. The major bottle neck here is that since so many calculations are performed per pixel is the correct use of caches such that calculations are done in an optimal fashion.

4.6.2 Spatial vs. Fourier Domain

This project has a scope only accounting for spatial domain filtering as it is the most expensive of the two and because of time limitations. Convolution can be perormed in the fourier domain, but has its limitations. For example, The fourier domain is asyptotically faster than the spatial domain, but has limitations such as the data set dimensions have to be a power of two [**?**]. This is usually not the case for seismic data, and therefore one would have to pad the data, which could cause distorsion or noise in the filtered image. This gives justification that performing convolution in the spatial domain is also usedfull in some cases, and to account for the limitations of the time available, our project focuses on the spatial domain.

4.6.3 Memory Access Patterns in Uni-Core CPU Implementation

In the first implementation only the use of one CPU core was in focus. This of course has the advantage of having to narrow the focus on optimizing memory use for one core, which then should be easy to expand to the use of quad cores with the use of threads, more precisely in this case the use of OpenMP.

Lets have a closer look at how the memory works in this case. After having read to the buffer there are three levels of cache that can be utilized [**?**]. The data in the buffer is aligned according to rows read from disk. This is a problem because when filtering for a pixel we will be looking at all its neighbors in all three dimensions. Which means that one must perform several jumps in memory to obtain the next

values. This is of course not ideal, but since the jumps in memory are regulated by the dimensions of the filter, which are constant, the pre-fetcher should be able to recover them while calculations are being perform resulting in close to no cache misses. This is seen in the profiler results where when running on one CPU the cache misses are 0.01% of the time. Because of the nature of how the data is aligned and the constant jumps in memory the prefetcher is actually able to memory optimize the convolution algorithm.

This is the case of fetching from the buffer to the third level of cache. In the case of when the correct data is retrieved in the third level of cache. One is then retrieving a subset of that data to the level 2 and level 1 cache for calculations. The nice thing here is that now the data is aligned such that each sub cube, of an equal size as the filter, are placed after one another. And when performing calculations in the level 1 cache then the data needed is aligned such that it is read in a straight line resulting in well aligned data on all levels of the architecture. The only cache misses present are when one reaches the end of a the data set in the x dimension, and proceeds to jump to the next row below it. This cache miss occurs on the level 2 and level 1 cache and is a result of non-overlapping data between the two pixels being filtered. In the normal case there is overlap between neighboring pixels and one can perform calculations on these values since they are in the cache while the pre-fetchers attains the next values, but in the case where there is no overlap one must wait for the values to be gathered before proceeding. This is a rare case, because of the blocking pattern, and that is why it does not effect the overall cache misses of the application.

4.6.4 Memory Access Patterns in Quad-Core CPU Implementation

Knowing that one has an optimal serial implementation one can then justify comparisons with other implementations to analyze gained speedup. In the quad-core implementation one can use this optimal structure of memory and aligned data to run 4 threads where all the data in the level 3 cache is shared. This should result in that the implementation be more bound by memory bandwidth since memory access is optimized. Given that there are now 4 CPU performing the same task as one did before and are totally independent this should result in a theoretical 4 time speedup. Some bottlenecks here are of course the memory transfer rates, but the implementation should come quite close for large amounts of data and computation. In this project, the results obtained are a 3.57 speedup, which is close to the theoretical value and also underlines the fact that the implementations are memory optimized. The profiler also indicates this in the same way as for the uni-core implementation.

4.6.5 Effects of Operating System on Convolution

In the case of running on the uni-core one has 3 other cores that the operating system can use to perform tasks other than that of convolution. But, in the case of utilizing all the capacity of the CPU by running the convolution process on all

4 cores, tasks related to the OS will interrupt or interleave with the convolution process. This could be one of the major issues related to the speedup limitations noticed in the results. The fact that a speedup of 3.57 and not an ideal 4 times speedup is achieved when running on several cores could have many reasons. One could be the fact that only convolution is run in parallel and not I/O. But the trend that even for large sets, where I/O is less significant, no greater speedup was seen could be because large sets have greater execution time and the operating system is bound to interrupt the process, creating another bottleneck. This is avoidable by running on an accelerator such as the GPU.

4.6.6 Effects of the PCI-express Bus

Even though the same operation is performed on the GPU the memory access patterns differ. First of all there is a an addition to the pipeline, which is actually quite time consuming [**?**], and is the transfer of data from the system buffer to the GPUs buffer (global memory). This is done on the PCI express bus which has an average bandwidth of about 1 GB per second. This can prove to be troublesome if one has to perform several transfers such as in the case of large data sets. Another issue is that on the GPU there will be more transfers because of the size of the memory. The system buffer has a size of 8 GB ,where 2 are occupied by the OS. This allows 3GB (since convolution requires duplication) to be read from disk at a time as discussed earlier. While the GPU memory only allows 2GB. Therefore the combination of both having to transfer more often to process the same amount of data and that the transfer is slower because of the bottle neck the PCI express bus one cannot expect linear speedup between the amount of cores on the GPU and the CPU. Not to mention that the GPU processing units have a lower frequency. Nevertheless given that convolution is very parallelizable a speedup is expected.

4.6.7 Results of CPU vs GPU

The results indicate that for large data sets that the GPU is actually about 17 times faster than a uni-core CPU. And is expected to be faster as long as each thread has a lot to do i.e. the size of filter is large. Because each pixel has to calculate values along the filters dimensions. Nonetheless a 17 time speedup is actually a good result in the case of pre-processing large amounts of data. In the case where one would use 16 hours on a uni-core. On the GPU it would run in about 54 minutes. Thereby one has reduced the problem dimensions from hours to minutes.

4.6.8 GPU Occupancy

To utilize the processing capabilities of the GPU one has to make sure that there are enough threads available to actually perform all the calculations and obtain maximum occupancy on the GPU. NVIDIA have produced an occupancy calculator [**?**] that helps developers to analyze their implementations and show how much of

the GPU is actually occupied at a time. Here there is a fine balance between the number of threads, blocks and registers one uses. Because there is a limited number of registers per kernel then restrictions are introduced to the max amount of threads. In the case of this convolution algorithm there is a need for 23 registers (information retrieved from the cubin file), which results in only 63% of possible threads can be run at a single kernel launch. Meaning that only 63% occupancy can be obtained. The alternative is to push some values from the registers to global memory and run more threads. But since the register values are used often and forced lowering of register use in the complier would be place values in global memory will result in lower performance. This is because registers are accessed much faster than global memory and the values in these registers are used constantly.

4.6.9 Memory Access Patterns in GPU Implemenatation

Other than transferring the data to the graphical processor one has to optimize the kernel (algorithm running on the GPU) such that it utilizes the performance capabilities of the GPU. As mentioned before, convolution is a very bandwidth bound problem and on the GPU there are several layers of memory that can accelerate bandwidth, but they need to be explicitly programmed to. The shared memory layer is a private memory for each block such that threads within the block can communicate and cooperate without having to access global memory. This is memory that can be accessed within one cycle and is on chip, but needs to be transfered from global memory [**?**]. In the case of convolution there is no communication or cooperation within the threads and therefore it is not used. There is another layer of memory that can prove to be useful in this case, and that is constant memory. The issue here is that constant memory is small in size. One could use the global memory to run ones code from, but the memory latency would ruin performance [**?**].

4.6.10 Why use Constant Memory and Not Shared Memory?

Shared memory is the fastest of all buffer layers on the GPU. It can be accessed within one cycle of computation. The only other buffer that has this attribute is constant memory, and even here it is only valid for best case when the data is coalesced. The trouble with both these buffer is their size. Both are in the range of KB, even though constant memory is 4 times larger. The issue with shared memory is that it needs to be updated per kernel launch and one cannot send data directly to it since it is on chip. To use shared memory one has to retrieve data from the global memory and then do calculations on them. The trouble is that since it is so small (16KB) not much data will be buffered there and the overhead work of transferring the data from global memory might result in a decrease in performance. Making its use both complicated and with no guarantee of speedup.

Constant memory on the other hand is off-chip and can be updated directly without having to go through global memory, and it does not require to be updated per kernel

launch. But, it must be allocated statically. Meaning that one must at compilation time decide how much memory should be allocated. This is ideal for values that are constant throughout the computations, hence its name. Another limitation here is that constant memory is read-only from device. Therefore it cannot be used to store convoluted data, but in this implementation it is used to store the convolution filter. This is ideal since the convolution filter is used in all the calculation and retrieving results from it is half the job in convolution. Therefore if coalesced it can be retrieved in one cycle (best case) resulting in great speedup. And since the filter values are read only this is ideal. If the filter is to be placed in shared memory then it would have to retrieved from global per kernel launch, which is a great overhead compared to the use of constant memory. Shared memory also has to be coalesced to perform well.

4.6.11 What if Convolution Filter is Larger Than Constant Memory?

For filter sizes larger than 25x25x25 there will not be enough space in constant memory. There are two solutions here either one can run everything from global memory, which the results show gives a 12 times speedup. Or, one could perform several iterations with a smaller filter. The result would be visually similar, in the case of blurring, and computationally there are similar amounts of operations performed. Mathematically however it is not similar and a larger filter should be used. A larger filter would only result in more computations, but the trouble with using smaller filters is that one would double communication time and time spent transferring data to the GPU, since the operation is done several times. Therefore placing all data in global would most likely give a better execution time. Since disk access and memory bandwidth are the main bottlenecks and doubling their time of execution would probably result in a sub-optimal solution.

4.6.12 Bank Conflicts and Coalescing

Another aspect of memory usage on the GPU that is worth noting is the possibility of bank conflicts that happen during writes to the memory. The way this is solved here is that each thread solves for a value and that is why each neighboring pixel writes in the neighboring address in memory. This results in coalesced writes, which are recommended by NVIDIA [**?**] to optimize performance. Not only does the GPU algorithm have coalesced writes, but it also reads coalesced. This is because if the current thread is reading a value, its neighboring thread is reading the value beside it at all times. Again this is recommended for high performance.

4.6.13 The Use of Multi-GPU

Now that all the steps are taken such that the kernel is running optimally on a single GPU one could progress to running the code on several GPUs. The results show that

running on 4 GPUs gives a speedup that is 3.7 times faster than on a single GPU for larger sets of data. This is the result of running the kernels from 4 threads on the host where each invoke their own GPU, and the pipeline structure in which data is transfered to run is also optimal. The fact is given the way that the application executes on the multiple GPUs, with running GPUs one after the other and not interleaving at the data transfer stage, hides a lot of computation and transfer time. And as the problem size grows the more evident this becomes. Because the only delay evident that will not be there when running on a single GPU is the three cudaMemCpy instructions run at the end of the execution. For large data sets this takes some time, but is not a very heavy task that is why one can obtain a speedup of close to 4. This is even more evident in the reduction of computation time as the data size grows for the 4 GPUS. since when there is little data, calculations will be fast and the transfer of data to and from the GPU will be the bottleneck. While in the case where there are larger amounts of data, the calculations are larger and the time of the transfer of data is smaller in comparison to calculations. The transfer of data to and from the GPU is counted as computation time since it is an overhead of using the GPU contra the CPU, and since its a new introduction to the computation pipeline then it should be accounted for in computation.

4.6.14 Communication vs. Computation time

One of the largest motivations to parallelize the convolution algorithm is the amount of time that is spent on computations rather than retrieving the data from disk. When running on a single CPU 99% of the time was used for computations. When trying to run the algorithm on several cores the amount of computation time was reduced to 94% on the quad-core and 75% on a single GPU. It was not before running on multiple GPUs a 10% computation time was achieved. This shows that convolution as a problem scales well with several cores. This is because of the independence of calculations. When reaching a distribution of 10% computation time and 90% I/O time then one should realize that further paralleizing and optimizing the 10% computation time will result in little gain, and the focus should now be shifted at improving I/O access. Maybe the use of parallel I/O could be useful, but given the architecture of the machines used here there is only one node (CPU) able to access disk and buffer. This means that there are no options to run disk access in parallel, but it would be an interesting aspect for further work.

CHAPTER 5

Conclusions and Future Work

In the case of processing large seismic data sets, there is always a concern of how to efficiently transfer the data in sub blocks from disk to buffer and back. However when running on a single CPU core, our results showed that I/O time was limited to only 1% of the execution time. This means that 3D convolution studied here in this report are computationally demanding. However, convolution is very parallelizable since the calculations in neighboring pixels are very independent. Our results showed that a speedup of 3.57 was gained when running on a quad-core platform, which is close to the theoretical speedup of 4.

Our results showed that the GPU could accelerate the task significantly. The results showed that running the application on a GPU resulted in a speedup of about 17. The main limitations were the bottleneck of having to transfer the data from system buffer to the global memory of the GPU and the fact that the convolution kernel used 22 registers for computations and thereby only a maximum of 63% occupancy was possible. Nevertheless, by having used the GPU we showed that it was possible to reduce the execution time from hours to minutes even for larger sets. We discovered that this was achieved by using constant memory for the filtering mask and global memory for the dataset, and both were accessed coalecsed. Shared memory was not used because it introduced an overhead per kernel launch, which was avoided by using constant memory.

Our results also showed that convolution was easy to distribute across several platforms. When using the Tesla s1070, which is similar to using 4 seperate GPUs, the problem scaled nearly perfectly. Comparing our results of running on a single GPU to running on four achieved a speedup of 3.7, which is very close to the theoretical four times speedup. This was accomplished by pipelining the executions on the GPUs such that they hide each others computation and I/O time. The speedup gained from this compared to the uni-core implementation is 62 with a computation time of only 10%. Hence the major bottleneck now has become the time used reading from disk.

5.1 Future Work

In this project, the focus has been on using non separable filters such that one must filter in all three dimensions per pixel. A suggestion for further work in this field is to explore what effects a separable filter would have on speedup and performance.

Another focus in this project has been at looking into filtering in the spatial domain. Therefore an improvement of further work could look into using the fourier domain

When maximizing performance on the GPU the use of constant memory was used in this implementation, for reasons given in the discussion. An interesting aspect for further work would be using the shared memory instead and compare performance between the two. Or even the use of NVIDIAs new GPU architecture Fermi where one can use caches.

As the performance is optimized the computation time is decreased and I/O is dominant again. This is seen when using the Tesla s1070 on large data sets where the computation time is only 10%, which means that 90% of the time is used on I/O. Further work here would involve further optimization of I/O maybe even experimenting with parallel I/O.

All code for the GPU is written in CUDA. Another option would be using OpenCL. This way one could compare using NVIDIA hardware to the use of AMD hardware. One can also compare the performance obtaind by simply changing the programming platform to OpenCL as well.

Bibliography and References

[1] AMD. http://AMD.com, last accessed 2009-11-03.

[2] Michael Allen Barry Wilkinson. *Parallel Programming*, pages 3–26. Prentice-Hall PTR, second edition, 2005.

[3] Comparing Cg and CUDA Implementations of Selected Transform Algorithms. Robin Eidissen. Norwegian University of Science and Technology, 2008.

[4] Microsoft MSDN C++ Library. Microsoft Corporation. http://msdn.microsoft.com/en-us/library/default.aspx, accessed 2009-12-02.

[5] Nvidia CUDA Zone. Nvidia Corporation. http://www.nvidia.co.uk/object /cuda_what_is_uk.html, accessed 2009-08-31.

[6] Bernard Flury. *A First Course in Multivariate Statistics*. Springer Verlag, first edition, 1997.

[7] Erwin Kreyszig. *Advanced Engineering Mathematics*. Peter Janzow, 8th edition, 1999.

[8] Ray Tracing Using Nvidia Optix. Holger Ludvigsen. Norwegian University of Science and Technology, 2009.

[9] Simulation of Fluid Flow Through Porous Rocks on Modern GPU. Eirik Aksnes. Norwegian University of Science and Technology, 2009.

[10] Edge Detection on GPUs Using CUDA. Aasmund Eldhuset. Norwegian University of Science and Technology 2009.

[11] 3D Finite Difference Computation on GPUs using CUDA. Paulius Micikevicius. Nvidia Corporation, 2008.

[12] GPGPU: General Purpose Computation on Graphics and Interactive Techniques. Leubke. Harris. Kruger. Purcel. Govindaraju. Buck. Woolley. Lefohn. ACM, SIGGRAPH, 2004.

[13] Modeling Communication on Multi-GPU Systems. Daniele Spampinato. Norwegian University of Science and Technology, 2009.

[14] Nvidia CUDA Occupancy Calculator [online]. Nvidia Corporation. http://developer.download.nvidia.com/compute/cuda /CUDA_Occupancy_calculator.xls, accessed 2009-08-31.

[15] Nvidia CUDA Programming Guide [online]. Nvidia Corporation. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs /NVIDIA_CUDA_Programming_Guide_2.3.pdf, accessed 2009-08-31.

[16] Nvidia CUDA Reference Manual [online]. Nvidia Corporation. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit /docs/CUDA_Reference_Manual_2.3.pdf, accessed 2009-12-02.

[17] OpenMP. http://openmp.org, accessed 2009-09-11.

[18] Richard E. Woods Rafael c. Gonzales. *Digital Image Processing*. Prentice-Hall PTR, third edition, 2008.

[19] Accelerating 3D Convolution using Graphics Hardware. Matthias Hopf. Thomas Ertl. IEEE Visualization, 1999.

[20] Schlumberger Stavanger. Personal Communication with Victor Aarre.

[21] Scallable Parallel Programming with CUDA. Nickolls. Buck. Garland. Skadron. Nvidia Corporation, 2008.

[22] Image Convolution with CUDA. Victor Podlozhnyuk. Nvidia Corporation, 2007.

APPENDIX A

Benchmarking Tables

READ PER BYTE

Test on only IO read/write without blokking

Dimension	Dim	Data MB	Time	Sec	Trans rate Kb/s
100x100x100	100	3.81	00,00,05	5	781.25
200x200x200	200	30.52	00,00,40	40	781.25
400x400x400	400	244.14	00,05,12	314	796.18
800x800x800	800	1953.13	00,42,13	2533	789.58
				Average	***787.06***

Test of IO read/write with blokking along xc X5

Dimension	Dim	Data MB	Time	Sec	Trans rate Kb/s
100x100x100	100	3.81	00,00,05	5	781.25
200x200x200	200	30.52	00,00,38	38	822.37
400x400x400	400	244.14	00,05,06	306	816.99
800x800x800	800	1953.13	00,40,53	2453	815.33
				Average	***808.99***

Test of IO read/write with blokking along vc X5

Dimension	Dim	Data MB	Time	Sec	Trans rate Kb/s
100x100x100	100	3.81	00,00,05	5	781.25
200x200x200	200	30.52	00,00,39	39	801.28
400x400x400	400	244.14	00,05,20	320	781.25
800x800x800	800	1953.13	00,42,04	2524	792.39
				Average	***789.04***

Test of IO read/write with blokking along ic X5

Dimension	Dim	Data MB	Time	Sec	Trans rate Kb/s
100x100x100	100	3.81	00,00,05	5	781.25
200x200x200	200	30.52	00,00,38	38	822.37
400x400x400	400	244.14	00,05,17	317	788.64
800x800x800	800	1953.13	00,42,04	2524	792.39
				Average	***796.16***

READ PER BLOCK

Test on only IO read/write without blokking

Dimension	Dim	Data MB	Time	Sec	Trans rate Kb/s
200x200x200	200	30.52	00,00,01,83	2	15625
400x400x400	400	244.14	00,00,17,39	17	14705.88
800x800x800	800	1953.13	00,02,31,00	151	13245.03
				Average	***14525.31***

Test of IO read/write with blokking along xc X5

Dimension	Dim	Data MB	Time	Sec	Trans rate Kb/s
200x200x200	200	30.52	00,00,01,80	2	15625
400x400x400	400	244.14	00,00,18,52	18	13888.89
800x800x800	800	1953.13	00,00,02,29	149	13422.82
				Average	***14312.24***

Test of IO read/write with blokking along vc X5

Dimension	Dim	Data MB	Time	Sec	Trans rate Kb/s
200x200x200	200	30.52	00,00,01,80	2	15625
400x400x400	400	244.14	00,00,17,50	18	13888.89
800x800x800	800	1953.13	00,02,30,74	150	13333.33
				Average	***14282.41***

Test of IO read/write with blokking along ic X5

Dimension	Dim	Data MB	Time	Sec	Trans rate Kb/s
200x200x200	200	30.52	00,31,31	31	1008.06
400x400x400	400	244.14	00,05,17	317	788.64
800x800x800	800	1953.13	00,42,04	2524	792.39
				Average	***863.03***

READ PER BLOCK MED MINNE OPTIMERING

Test of only IO read/write with blokking along vc X5

Dimension	Dim	Data MB	Time	Sec	Trans rate Kb/s
200x200x200	200	30.52	00,00,00,87	0.87	71839.08
300x300x300	300	103	00,00,03,06	3	70312.5
400x400x400	400	244.14	00,00,07,70	7.7	64935.06
500x500x500	500	476.84	00,00,18,89	19	51398.03
600x600x600	600	823.97	00,00,37,38	37	45608.11
700x700x700	700	1308.44	00,01,00,32	60	44661.46
800x800x800	800	1953.13	00,01,27,74	88	45454.55
900x900x900	900	2780.91	00,02,03,35	123	46303.35
1000x1000x1000	1000	3814.7	00,03,26,42	206	37924.76
				Average	***55064.02***

READ PER BLOCK MED MINNE OPTIMERING I C

Test of only IO read/write with blokking along vc X5

Dimension	Dim	Data MB	Time	Sec	Trans rate Kb/s
200x200x200	200	30.52	00,00,00,83	0.41	76219.51
300x300x300	300	103	00,00,02,61	1.59	66332.55
400x400x400	400	244.14	00,00,06,30	4.61	54229.93
500x500x500	500	476.84	00,00,11,93	8.95	54556.56
600x600x600	600	823.97	00,00,21,39	15.05	56063.12
700x700x700	700	1308.44	00,00,33,34	22.46	59654.66
800x800x800	800	1953.13	00,00,51,82	32.9	60790.27
900x900x900	900	2780.91	00,01,13,65	48.6	58593.75
1000x1000x1000	1000	3814.7	00,01,39,95	64.44	60618.4
2000x2000x2000	2000	30517.58	00,01,39,95	519.4	60165.58
				Average	***58854.91***

As of 2008, a typical 7200rpm desktop hard drive has a sustained "disk-to-buffer" data transfer rate of about 70 megabytes per second

READ PER BLOCK W/Filter

Test of IO read/write and filtering with blokking along vc X5

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %
5x5x5	400x400x400	1219.51	205	96.1
	500x500x500	1352.58	361	94.74
	600x600x600	1422.85	593	93.76
	700x700x700	1343.88	997	93.98
	800x800x800	1211.39	1651	94.67
			Average	***94.65***

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %
9x9x9	400x400x400	265.39	942	99.15
	500x500x500	242.93	2010	99.05
	600x600x600	241.62	3492	98.94
	700x700x700	239.6	5592	98.93
	800x800x800	236.41	8460	98.96
			Average	***99.01***

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %
13x13x13	400x400x400	87.23	2866	99.72
	500x500x500	86.42	5650	99.66
	600x600x600	87.42	9651.22	99.62
	700x700x700	86.3	15525	99.61
	800x800x800	85.02	23525	99.63
			Average	***99.65***

READ PER BLOCK W/Filter I C

Test of IO read/write and filtering with blokking along vc X5

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %
5x5x5	400x400x400	11332.73	22.06	79.1
	500x500x500	12056.33	40.5	77.9
	600x600x600	12433.69	67.86	77.82
	700x700x700	12294.4	108.98	79.39
	800x800x800	11946.72	167.41	80.35
			Average	***78.91***

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %
9x9x9	400x400x400	2824.86	88.5	94.79
	500x500x500	2625.17	186	95.19
	600x600x600	2528.24	333.73	95.49
	700x700x700	2547.13	526.02	95.73
	800x800x800	2539.97	787.41	95.82
			Average	***95.4***

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %
13x13x13	400x400x400	1045.94	239.02	98.07
	500x500x500	995.88	490.3	98.17
	600x600x600	991.22	851.22	98.23
	700x700x700	983.6	1362.19	98.35
	800x800x800	941.69	2123.84	98.45
	900x900x900	922.68	3086.3	98.43
	1000x1000x1000	901.55	4332.8	98.51
	2000x2000x2000	875.48	35694.7	98.54
			Average	***98.26***

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %
17x17x17	400x400x400	497.02	503	99.08

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %
21x21x21	400x400x400	283.19	882.8	99.48

READ PER BLOCK W/Filter I C OPENMP

Test of IO read/write and filtering with blokking along vc X5

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %	speedup
5x5x5	400x400x400	24557.96	10.18	54.72	2.17
	500x500x500	23577.08	20.71	56.78	1.96
	600x600x600	22841.09	36.94	59.26	1.84
	700x700x700	22469.29	59.63	62.33	1.83
	800x800x800	20383.2	98.12	66.47	1.71
			Average	***59.91***	***1.9***

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %	speedup
9x9x9	400x400x400	7861.64	31.8	85.5	2.78
	500x500x500	7617.49	64.1	86.04	2.90
	600x600x600	7407.81	113.9	86.79	2.93
	700x700x700	6662.57	201.1	88.83	2.62
	800x800x800	5876.3	340.35	90.33	2.31
			Average	***87.5***	***2.71***

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %	speedup
13x13x13	400x400x400	3298.15	75.8	93.92	3.15
	500x500x500	3248.71	150.3	94.05	3.26
	600x600x600	3127.32	269.8	94.42	3.16
	700x700x700	3159.86	424.02	94.7	3.21
	800x800x800	3316.75	603	94.54	3.52
	900x900x900	3293.61	864.6	94.38	3.57
	1000x1000x1000	3219.79	1213.2	94.69	3.57
	2000x2000x2000	3136.83	9962.3	94.79	3.58
			Average	***94.33***	***3.26***

READ PER BLOCK W/Filter I CUDA

Test of IO read/write and filtering with blokking along vc X5

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %	speedup
5x5x5	400x400x400	45454.55	5.5	16.18	4.01
	500x500x500	46064.27	10.6	15.57	3.82
	600x600x600	45855.98	18.4	18.21	3.69
	700x700x700	45112.58	29.7	24.38	3.67
	800x800x800	44444.44	45	26.89	3.72
			Average	***20.24***	***3.78***

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %	speedup
9x9x9	400x400x400	30487.8	8.2	43.78	10.79
	500x500x500	28554.46	17.1	47.66	10.88
	600x600x600	28994.85	29.1	48.28	11.47
	700x700x700	28813.84	46.5	51.7	11.31
	800x800x800	28208.74	70.9	53.6	11.11
			Average	***49***	***11.11***

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %	speedup
13x13x13	400x400x400	16556.29	15.1	69.47	15.83
	500x500x500	15354.76	31.8	71.86	15.42
	600x600x600	15066.96	56	73.13	15.2
	700x700x700	15365.18	87.2	74.24	15.62
	800x800x800	15048.91	132.9	75.24	15.98
	900x900x900	15122.98	188.3	74.19	16.39
	1000x1000x1000	15012.49	260.2	75.23	16.65
	2000x2000x2000	14744.74	2119.4	75.49	16.84
			Average	***73.73***	***25.59***

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %	speedup
17x17x17	400x400x400	8503.4	29.4	84.32	17.11

Filter size	Dimension	Comp Rate Kb/s	Sec	Computation Time %	speedup
21x21x21	400x400x400	4816.96	51.9	91.12	17.01

READ PER BLOCK W/Filter I CUDA MULTI GPU

Test of IO read/write and filtering with blokking along vc X5

Filter size	Dimension	Comp Rate Kb/s	Sec	Comp Time %	speedup CPU	speedup GPU
5x5x5	400x400x400	46382.19	5.39	14.47	4.09	1.02
	500x500x500	46502.98	10.5	14.76	3.86	1.01
	600x600x600	47005.57	17.95	16.16	3.78	1.03
	700x700x700	46603.26	28.75	21.88	3.79	1.03
	800x800x800	44943.82	44.5	26.07	3.76	1.01
			Average	***18.67***	***3.86***	***1.02***

Filter size	Dimension	Comp Rate Kb/s	Sec	Comp Time %	speedup CPU	speedup GPU
9x9x9	400x400x400	35161.74	7.11	35.16	12.45	1.15
	500x500x500	35002.24	13.95	35.84	13.33	1.23
	600x600x600	35436.79	23.81	36.79	14.02	1.22
	700x700x700	35074.44	38.2	41.2	13.77	1.22
	800x800x800	35149.38	56.9	42.18	13.84	1.25
			Average	***38.24***	***13.48***	***1.21***

Filter size	Dimension	Comp Rate Kb/s	Sec	Comp Time %	speedup CPU	speedup GPU
13x13x13	400x400x400	32051.28	7.8	40.9	30.64	1.94
	500x500x500	37880.62	12.89	30.57	38.04	2.47
	600x600x600	43048.47	19.6	23.21	43.43	2.86
	700x700x700	47698.25	28.09	20.04	48.49	3.1
	800x800x800	49875.31	40.1	17.96	52.96	3.31
	900x900x900	52154.88	54.6	10.99	56.53	3.45
	1000x1000x1000	54863.06	71.2	9.49	60.85	3.65
	2000x2000x2000	54357.28	574.9	9.65	62.09	3.69
			Average	***20.55***	***78.61***	***3.5***