



Norwegian University of
Science and Technology

Effects of Compression on Data Intensive Algorithms

Ahmed Adnan Aqrawi

Master of Science in Computer Science

Submission date: June 2010

Supervisor: Anne Cathrine Elster, IDI

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

Problem Description

Seismic 3D data typically exceeds given caches and memory available on modern systems. GPUs offer a lot of computational power, albeit also have their limits, especially regarding I/O.

In this master thesis project, we will investigate what advantages compression can offer as a means to bridge the increasing gap between processor speeds and memory and disk speeds by reducing communication in exchange for compression/decompression computations. In particular, we will study at least two types of problems, one with overlapping borders, and one without and see what compression can offer for given 2D and 3D domain sizes, multi-core CPU vs GPU (or a combination of both), as well as I/O media (e.g. HDD vs SSD).

The goal is to provide a predictive I/O model for several classes of seismic algorithms. This model will be derived from our empirical studies of both lossy and lossless compression algorithms used in combination with selected filtering algorithms such as the Hough transform and convolution.

Assignment given: 26. January 2010
Supervisor: Anne Cathrine Elster, IDI

Abstract

In recent years, the gap between bandwidth and computational throughput has become a major challenge in high performance computing (HPC). Data intensive algorithms are particularly affected. by the limitations of I/O bandwidth and latency. In this thesis project, data compression is explored so that fewer bytes need to be read from disk. The computational capabilities of the GPU are then utilized for faster decompression. Seismic filtering algorithms, which are known to be very data intensive, are used as tests cases.

In the thesis, both lossless and lossy compression algorithms are considered,. We have developed, optimized and implemented several compression algorithms for both the CPU and GPU using C, OpenMP and NVIDIA CUDA. A scheme for utilizing both the CPU and GPU using asynchronous I/O to further improve performance is also developed. Compression algorithms studied and optimized include RLE, Huffman encoding, 1D-3D DCT, 1D-3D Fast DCT AAN algorithm, and the fast LOT. 3D convolution and the Hough transform filtering algorithms are also developed and optimized.

Lossy compression algorithms using transform encoding are also studied. Using these transforms for compression include: 1) transformation, 2) quantization and 3) encoding. Transformation and quantization are shown to be especially suitable for the GPU because of their parallelizable nature. The encoding step is shown to be best done on the CPU because of its sequential nature. GPU and CPU are used in asynchronous co-operation to perform the compression on seismic data sizes (up to 32GB). Transform coding is lossy, but the errors we experience are minimally visible and are within acceptable loss given the type of data (a max. of 0.46% ME and 81 rMSE for our seismic data sets).

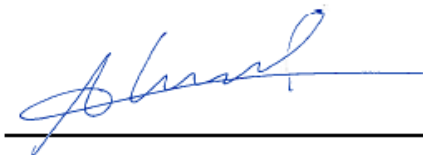
HDD disk with 70MB/s transfer rate, and a speedup of 3.3 for a modern SSD with a 140MB/s transfer rate. Several other results on both the recent NVIDIA Tesla c1060 GPU and the new NVIDIA Tesla c2050 Fermi-based GPU, as well as results for using CPU and GPU together using asynchronous I/O is included. The major bottleneck now is the PCI express bus limitations, and for files that do not compress well, the I/O bandwidth and latency is still an issue.

Acknowledgement

This thesis, together with the prototype, is the result of a Master thesis project assigned by the Department of Computer and Information Science at the Norwegian University of Science and Technology.

I would like to thank my supervisors Dr. Anne C. Elster for invaluable support and feedback throughout the entire thesis. 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 Dr. Victor Aarre and Mr. Cristian Larsen of Schlumberger for their 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 our group and 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. A special thanks to Jan Christian Meyer, Thorvald Natvig, Dr. John Ryan for their support, and a special thanks to my good friend and colleague Holger Ludvigsen for his help and guidance.

Trondheim, June 2010



Ahmed Adnan Aqrabi

Contents

Abstract	i
Acknowledgement	iii
Contents	v
List of Figures	ix
List of Tables	xii
1 Introduction	1
1.1 Goals	3
1.2 Contributions	3
1.3 Thesis Outline	4
2 Parallel Computing	7
2.1 Parallel Computing Theory	7
2.1.1 Amdahl's Law	8
2.1.2 Gustafson's Law	8
2.1.3 Speed Up Limitations	8
2.1.4 Data Dependencies	9
2.1.5 Types of Parallelism	10
2.1.6 Classes of Parallel Computers	10
2.2 OpenMP and Multithreading	11
2.2.1 Main Features	12
2.2.2 Pros and Cons	12
2.2.3 Performance expectations	13
2.3 GPU and CPU Architectural Features	13
2.3.1 Main aspects of CPU architecture	14
2.3.2 Main aspects of GPU architecture	15
2.4 CUDA Programming Model	16
3 Data Compression, Filtering and Seismic Data	19

3.1	Previous Work on Compression	19
3.2	Lossless Compression Algorithms	21
3.2.1	Run Length Encoding (RLE)	22
3.2.2	Huffman Encoding	23
3.2.3	Arithmetic Encoding	25
3.2.4	Lossless Compression of Floating-Point Data	26
3.3	Lossy Compression Algorithms	27
3.3.1	Compression Using Transforms	28
3.3.2	DCT (Discrete Cosine Transform)	30
3.3.3	Fast DCT: The AAN Algorithm	31
3.3.4	Lapped Orthogonal Transform	31
3.4	Filtering Algorithms	32
3.4.1	Convolution	32
3.4.2	Hough Transform	34
3.5	Seismic Data	36
4	Compression and Filtering Algorithm Optimizations and Implementations	39
4.1	The File Compression Format	40
4.2	Our Testing Frameworks	41
4.2.1	Producing Images	41
4.2.2	Producing Seismic Cubes	42
4.2.3	Synchronous and Asynchronous I/O	43
4.2.4	Benchmarking Framework	43
4.3	Optimizing Lossless Compression Algorithms	44
4.3.1	Optimizing RLE w/ Dictionary lookup	44
4.3.2	Optimizing Huffman Encoding	46
4.4	Optimizing Lossy Compression Algorithms	50
4.4.1	Optimizing the Naive DCT Algorithm	51
4.4.2	Optimizing Fast DCT: The AAN Algorithm	52
4.4.3	Optimizing Fast LOT	54
4.5	Optimizing Image Processing Algorithms	56
4.5.1	Optimizing 3D Convolution	56
4.5.2	Optimizing Hough Transform	59
4.6	Our AESC Library	61
5	Predictive Model for Seismic Processing I/O	63
5.1	Synchronous Model	64
5.2	Asynchronous Model	65
5.3	Compression Computation and I/O Tradeoffs	66

6	Results, Discussion and Analysis of Benchmarks	69
6.1	Hardware & Platforms Used for Testing	70
6.2	Data Sets for Tests	71
6.3	Compression Algorithms Performance and Visual Results	72
6.3.1	Modified RLE Benchmarks	72
6.3.2	Huffman Encoding Benchmarks	75
6.3.3	Naive DCT Benchmarks	78
6.3.4	AAN Implementation Benchmarks	84
6.3.5	LOT Implementation Benchmarks	90
6.4	Image Processing Algorithms Performance and Visual Results	93
6.4.1	3D Convolution Benchmarks	94
6.4.2	Hough Transform Benchmarks	100
6.5	Effects of Compression on the Seismic Filtering Process	103
6.5.1	I/O speedup	103
6.5.2	Predicted Model	107
6.5.3	Seismic Filtering Process Speedup	110
7	Conclusions and Future Work	115
7.1	Conclusions	115
7.2	Future Work	117
7.3	Closing Remark	119
	Bibliography and References	121
	Appendices	125
A	Orthogonal Transform Theory	125
A.1	DCT (Discrete Cosine Transform)	125
A.2	Fast DCT: The AAN Algorithm	127
A.3	Lapped Orthogonal Transform	129
B	Annotated Bibliography	135
C	Benchmarking Tables	141
D	Source Code	153
D.1	RLE	153
D.2	Huffman	155
D.3	Naive DCT	158
D.4	Fast DCT AAN	161
D.5	Fast LOT	165
D.6	Hough Transform	170

D.7 Convolution	170
E AESC Library Overview	173
F Short Paper for PARA 2010	175
G Poster ISC 2010	181

List of Figures

1.1	Graph from Hennessey and Patterson [21] portraying the increasing gap between memory bandwidth and computational throughput, with permission from David Patterson [34]	1
2.1	The Fork and Join of threads with OpenMP ¹	11
2.2	Language extentions of OpenMP ²	12
2.3	CPU and GPU transistor usage and layout from [23], with permission from NVIDIA	13
2.4	SIMD and SPMD from [23], with permission from NVIDIA . . .	14
2.5	CUDA architecture from [23], with permission from NVIDIA . .	17
2.6	CUDA memory hierarchy and the GPU architecture from [23], with permission NVIDIA	18
3.1	Huffman compression flow graph	23
3.2	Huffman compression with illustrated steps	24
3.3	Arithmetic compression flow graph	26
3.4	The process of compressing floats	27
3.5	The transformcoding process, inspired by [27]	29
3.6	2D Gaussian distribution graph drawn in online 3D grapher ¹ .	33
3.7	Example of two dimensional Gaussian Filter Mask with discrete values	34
3.8	Three point represented in the (x,y)- space and (r, θ)-space, obtained from ²	35
3.9	Byte stream structure of the SEG-Y Format with N textual headers and N traces and trace records. Image inspired from [41]	36
3.10	Seismic data example made available by Schlumberger from [1] .	37
4.1	Illustration of our compression format of the RLE algorithm . .	40
4.2	Image illustrating slices of x,z dimensions along the y axis . . .	42
4.3	Graphs of data distribution of seismic data given values of length 2-, 4-, 8-, and 16-bit	47
4.4	CUDA profiler snapshot of the LOT execution	55
4.5	NVIDIA CUDA profiler results for 3D convolution	59

4.6	CUDA profiler snapshot of the Hough transform execution . . .	61
5.1	Asynchronous I/O Pipeline	65
5.2	Showing advantages in execution time for combinations of fast/s- low compression, high and low compression rates and asyn- chronous compression	67
6.1	Seismic BMP images generated by our framework, based on the Westcam raw data set [1]	71
6.2	Execution time results for RLE algorithm	73
6.3	Execution time results for Huffman encoding algorithm	76
6.4	Before and after images of the transform encoding process using the naive DCT algorithm, generated by our framework	79
6.5	Execution time results for naive DCT 1D algorithm	80
6.6	Execution time results for naive DCT 2D algorithm	81
6.7	Execution time results for AAN DCT 1D algorithm	85
6.8	Execution time results for AAN DCT 2D algorithm	86
6.9	Execution time results for AAN DCT 3D algorithm	86
6.10	CUDA profiler snapshot of the DCT AAN 1D execution	87
6.11	CUDA profiler snapshot of the DCT AAN 2D execution	87
6.12	CUDA profiler snapshot of the DCT AAN 3D execution	87
6.13	Before and after images of the transform encoding process using the AAN DCT algorithm in several dimensions, generated by our framework	88
6.14	Execution time results for LOT algorithm	91
6.15	Seismic data after transform coding LOT and AAN in 1D, gen- erated by our framework	92
6.16	Execution time results for 3D convolution algorithm with filter size 13^3	94
6.17	Blur using 3D convolution with filter size 13^3 , generated by our framework	96
6.18	Execution time results for Hough transform algorithm	100
6.19	Visual results of the Hough transform, generated by our framework	101
6.20	Execution time results for synchronous I/O	104
6.21	Execution time results for Asynchronous I/O	105
6.22	Predicted execution time for synchronous model	107
6.23	Predicted execution time for asynchronous model	108
6.24	Effects of compression formats on seismic process for the seismic filtering algorithms 3D convolution and Hough transform	111
6.25	Speedups of the seismic process given a seismic filtering algo- rithm for compression format 3	112

A.1	Flowgraph of the AAN algorithm based on Pennebaker [35] and Arai et.al. [5] design, and obtained from [27], legends were added to simplify understanding	128
A.2	Flowgraph of the LOT algorithm based on Malvar and Staelin [30] design, and obtained with permission from [27], legends were added to simplify understanding	131
A.3	Flowgraph showing the Z matrix based on Malvar and Staelin [30] design, and obtained with permission from [27], see legends from Figure A.2 to simplify understanding	132
A.4	Flowgraph showing the Y rotaion matrix based on Malvar and Staelin [30] design, and obtained with permission from [27], see legends from Figure A.2 to simplify understanding	132
A.5	Flowgraph showing the larger process of the transform based on Malvar and Staelin [30] design, and obtained with permission from [27], see legends from Figure A.2 to simplify understanding	133

List of Tables

2.1	Architectural difference between NVIDIA CUDA architectures 1.3 and 2.0 (Fermi) as seen on www.nvidia.com	16
2.2	CUDA Memory Hierarchy, with data from [23]	18
4.1	Intel Vtune results for RLE	45
4.2	calculated compression rates from study of data of 2GB	46
4.3	Intel Vtune results for Huffman encoding	49
4.4	Intel Vtune results for Naive DCT transform encoding	52
4.5	Intel Vtune results for AAN DCT transform encoding	53
4.6	Intel Vtune results for 3D Convolution	58
4.7	Intel Vtune results for Hough Transform	60
6.1	Table showing the system components used in machine 1	70
6.2	Table showing the system components used in machine 2	70

CHAPTER 1

Introduction

One of the major challenges on modern computer platforms is overcoming the memory and I/O bandwidth and latency. In recent years, the gap between bandwidth and computational throughput has grown even larger causing a further challenge. As stated by Hennessey and patterson [21] current trends show that the gap will only wideb in the furutre as illustrated in Figure 1.1. This is especially the case for data intensive algorithms.

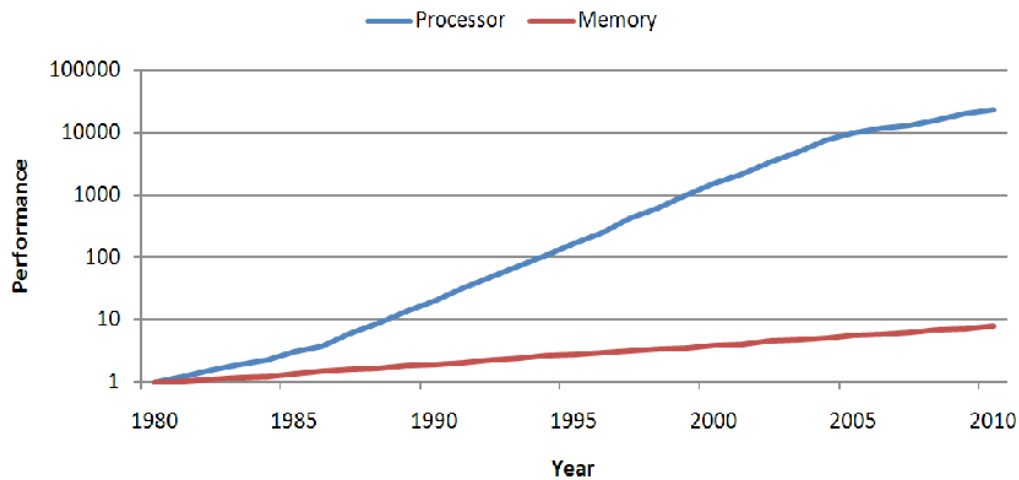


Figure 1.1: Graph from Hennessey and Patterson [21] portraying the increasing gap between memory bandwidth and computational throughput, with permission from David Patterson [34]

There are both hardware and software alternatives for optimizing the I/O bandwidth. One can try changing the hardware platform one is running on such as using SSD (solid state disk) rather than a HDD (Hard disk drive), as it performs at a much higher bandwidth. However, even this alternative has

its limitations. For this reason we are looking into compressing the data so that less data needs to be transferred from disk, and by using the computational power of the system to accelerate the I/O process. As we have already discovered in our previous work, and as it has been shown in several other cases, the computational power of the GPU is superior to the CPU in some cases. We are attempting to use these computational capabilities of the GPU for compression both as an accelerator and a supporter for the CPU during computations.

The case of using compression to improve I/O time has some unique properties. In the usual compression scenario, one is aiming for high compression rates and neglect the execution time cost to achieve this compression, which is typical of effective lossless compression methods such as LZMW [40] or lossy methods such as the GenLOT [12] with transform coding [35]. However, when optimizing for I/O, we not only need efficient compression rates, but also fast compression algorithms. Lossless compression is when one does not lose any data during compressing, which is optimal in cases where one is concerned about losing data. In lossy compression on the other hand, some data is lost during compression. In seismic filtering, it is acceptable to lose some data, as long as one maintains two decimal accuracy. In this thesis, both lossless and lossy compression algorithms are considered. Our focus will be on simpler faster lossless algorithms and transform coding because of its proven usefulness on seismic data [14].

The exploration process in the oil industry is heavily dependent upon Seismic imaging, which is considered as very data intensive. Seismic imaging is a method of exploring the layers of earth by using signal technology, which like ultrasound imaging includes recording waves, reconstruction, filtering and analysis. It is the common case that seismic data is recorded, reconstructed and then stored for later filtering and analysis. In this thesis, we will focus on the data after that it is reconstructed. This part of the seismic process that we are focusing on is termed seismic filtering. The filtering process is built upon two parts: 1) the transfer of data to a computation platform and 2) the actual filtering. In our previous work [4], the use of the GPU was explored on a typical filtering algorithm, namely 3D convolution. these results showed that to being with the transfer time consumed 2% of the execution time. As filtering was optimized by utilizing the computations capabilities of the GPU, the transfer time was 90% of execution time, making it the biggest limiting factor for further optimization. Another aspect worth noting is that this is not only a limiting factor for our work, but for all data intensive algorithms such as seismic processing.

1.1 Goals

Here are our Three main goals with this work:

1. To explore ways of using data compression to overcome memory and I/O bandwidth limitations with the focus on large sets of seismic data
2. To model this communication pattern mathematically such that it is possible to estimate execution time, and validate it by comparing results with empirical data from our implementations.
3. To develop, optimize and implement filtering algorithms and note the effects that compression has on the seismic process.

1.2 Contributions

The contributions of this thesis include:

- Optimized and modified implementations of several compression algorithms on both modern CPU and GPU including RLE (rRun length encoding), Huffman encoding, Transform encoding using DCT, AAN and LOT transforms.
- Optimized implementations of seismic algorithms, such as 3D convolution and Hough transform
- Introducing a method of compression where the GPU and CPU work as co-processors in order to achieve even better performance than could be achieved using either one on their own
- Methods to accelerate the I/O bandwidth for large seismic data using both CPU and GPU in combination with lossy and lossless compression algorithms
- A predictive model for the seismic process such that one can estimate execution time for different architectures
- Aqrawi and Elster Seismic Compression Library (The AESC library), a library of optimized compression algorithms for large seismic data sets

1.3 Thesis Outline

This thesis will be structured as follows:

Chapter 2: Relevant background material is highlighted and explained allowing the reader has to understand the work done in this thesis. Topics such as Parallel Computing ang GPU programming shall be addressed.

Chapter 3: In this chapter, an introduction of previous work within Data Compression is introduced and an explanation of the workings of the different compression algorithms to be implemented, in this thesis, are presented. We also have a closer look at seismic data and algorithm,s that have also been addressed in this thesis.

Chapter 4: Describes how the implementations in this thesis are performed, and why certain implementation decisions were made. Here one will also find the thoughts behind some optimizations and what the expectations are as to how they will ultimately perform.

Chapter 5: Describes a predictive model for execution time of given seismic algorithms with a focus on compression. The purpose of this chapter is to intorduce the reader to the concept of faster application I/O given compression of the data.

Chapter 6: Results regarding I/O tests are presented and discussed. The main focus here is on comparing the implementations and presenting speedup and computation results. An in depth analysis of the results and algorithmic traits are also presented, including a validation of the predictive model presented in the previous chapter. There will also be a discussion regarding error terms and visual effects given when different compression algorithms are employed.

Chapter 7: Here the conclusion of the work performed are presented. There will also be suggested future work within the field.

Appendix A: consists of a more indepth look at the definitions of the orthogonal transforms used in this thesis work. This has been moved to the appendix to avoid unnecessary reading for those familiar with this work. For those not familiar with orthogonal transforms such as the DCT (Discrete cosine transform) and LOT (Lapped orthogonal transform), we recommend you read this before our discussion and analysis chapter (Chapter 6).

Appendix B: To give more insight into some of the researched material, we have included an Annotated Bibliography

Appendix C: Here we have included Benchmark tables of our algorithms, the values here are then selectively presented in graphs in Chapter 6 when our results are discussed.

Appendix D: Here is some example source code from our CUDA and C implementations, since in Chapter 4 we speak mainly of optimizations and show pseudo code. This appendix functions as a lookup for those interested in the actual code.

Appendix E: Here we have an overview of the functions and compression methods available in our AESC compression library that was implemented as part of this thesis for further use in the field.

Appendix F: This an extended abstract/ short paper that lead to a talk at the PARA 2010 conference. It functions as a good overview for part of the reaserch done in this thesis.

Appendix G: Contains a poster created for HPC-LABs stand at ISC 2010 (International super computing) conference in Germany this year. It functions also as a good overview of some of the reaserch done in this thesis.

1.3. *THESIS OUTLINE*

CHAPTER 2

Parallel Computing

This chapter focuses on introducing some of the main concepts in parallel computing that the reader may need in order to understand our work. The following sections summarize the main references read. Section 2.1-2.4 gives a general overview of parallel programming, introducing main theories within parallel programming that will be used in the discussion of the thesis. Here one can also find introductions to OpenMP and the concepts of multithreading and explanations of the main aspects of modern CPU and GPU architectures, and NVIDIA CUDA, are also included

2.1 Parallel Computing Theory

In parallel computing, many calculations are performed simultaneously. This depends on dividing a larger problem into smaller ones such that each part can be calculated concurrently. There are several different forms of parallel computing: bit-level, instruction level, data parallelism, and task parallelism. Parallel computing has been around for quite a while now, but has lately been given more attention since it is no longer possible to continue to increase the clock speed of the processors at the same rate as before due to limiting the power and frequency walls. That is, much higher clock speeds would both consume too much power and become too hot to cool as the power needed grows exponentially with regards to frequency. This ultimately gave re-birth to the multi-core CPUs and created a paradigm shift in computer architecture.

2.1.1 Amdahl'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 predicted with the help of Amdahl's law [48]. 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 2.1 [48]. S is the speedup, and P is the portion of the code that is parallelizable and is a value lesser than 1 and greater or equal to zero.

$$S = \frac{1}{1 - P} \quad (2.1)$$

2.1.2 Gustafson's Law

Another law that is an extension of Amdahl's law in computer science is Gustafson's law 2.2 [48]. 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. Amdahl assumes that there is a fixed problem size, while Gustafson looks at fixed time. Here S is the speedup, P is the number of processors and α is the non-parallelizable part of the program.

$$S(P) = P - \alpha(P - 1) \quad (2.2)$$

2.1.3 Speed Up Limitations

Ideally one should be able to gain linear speedup, i.e. one would increase the speedup in direct proportion to the amount of processing units available. For example if one has two processors then it should half the overall execution time of a program since the workload is divided into two. However, this is rarely the case since there are several factors that limit speedup when programming in parallel. One issue is that not all problems can be parallelized, they might have some sequential parts. Here one processor is executing the sequential part while the other one would be 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 (such as thread and process creation) and this may result in sub linear speedup. However, one might get superlinear speedup due to access to more cache and RAM.

Other common 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. Often processes also 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. These effects become more evident when dealing with large datasets since a lot of time may be spent reading and writing data to disk once communicating between data processing units.

2.1.4 Data Dependencies

The issues described above are commonly referred to as data dependencies, and are a common issue in parallel programming and should be understood to perform good parallel implementations of algorithms. A formal description of dependencies in applications are given by the Bernstein conditions [48]. 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 . It can be said that P_i and P_j are independent if they satisfy the following conditions 2.3, 2.4 and 2.5 (from [48]).

$$I_j \cap O_i = \emptyset \tag{2.3}$$

$$I_i \cap O_j = \emptyset \tag{2.4}$$

$$O_i \cap O_j = \emptyset \tag{2.5}$$

The fact that no program can run faster than its longest chain of dependencies will prevent a lot of speedup, this is termed 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 fairly concurrently.

2.1.5 Types of Parallelism

As mentioned earlier there are several types of parallelism including instruction level, data and task.

Instruction-Level Parallelism

Computer programs are in essence a sequence 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 hardware 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. Such instructions can only be grouped 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.

Task Parallelism

In contrast to data parallelism, task parallelism is where one can perform different computations on the same or different data. a limitation of task parallelism is that it usually does not scale with the size of the problem.

2.1.6 Classes of Parallel Computers

There are many classes of parallel computers including: multi-core computing, symmetric multiprocessing, distributed memory computing, cluster computing, massive parallel processing. There are also specialized accelerators such as

FPGA (field programmable gate arrays), GPGPU (general purpose graphical processing unit), application specific integrated circuits and vector processors. This work will be focusing on multi-core CPUs and GPGPU because these two are representing the two most prevalent architectures and are also set as the scope of this thesis.

2.2 OpenMP and Multithreading

OpenMP [32] is a standard that hides the burden 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 2.1

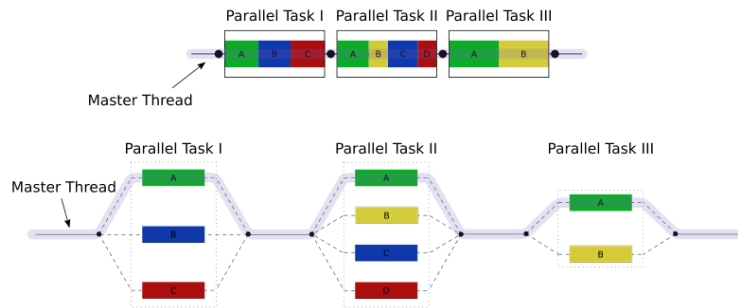


Figure 2.1: The Fork and Join of threads with OpenMP ¹

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.

¹http://en.wikipedia.org/wiki/File:Fork_join.svg, accessed 2009-09-11. Available to all under the license of creative commons

2.2.1 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 extensions see Figure 2.2 and [32].

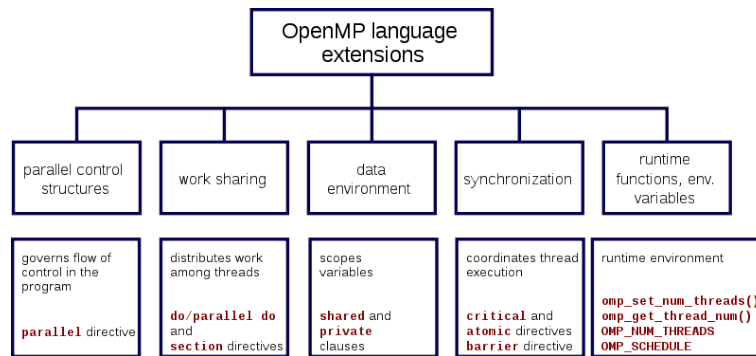


Figure 2.2: Language extensions of OpenMP ²

2.2.2 Pros and Cons

One of the greatest strengths of OpenMP is that it is simple to use, and that one does not have to deal with message passing. The shared memory architecture automatically handles 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 binary executables 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 the application runs. It only runs on shared memory multiprocessor platforms. It also requires a compiler that supports OpenMP.

²http://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

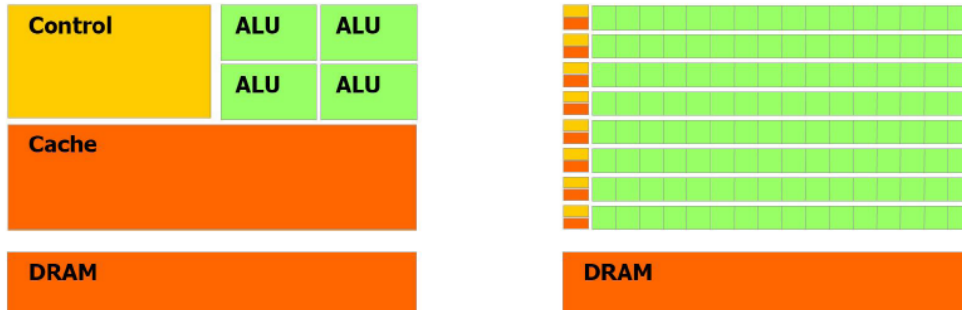


Figure 2.3: CPU and GPU transistor usage and layout from [23], with permission from NVIDIA

2.2.3 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 common 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.

2.3 GPU and CPU Architectural Features

The GPU 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) [44] [33]. 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 perform 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.

2.3.1 Main aspects of CPU architecture

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 thread.

The modern GPU should be viewed 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 sophisticated flow control that allows it to take advantage of the instruction level parallelism (See Figure 2.4). Another interesting aspect of the CPU is the use of many transistors to reduce memory latency with the use of caches, see Figure 2.3. 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. Optimize caching and memory access is very important when programming on the CPU in order to obtain maximum performance.

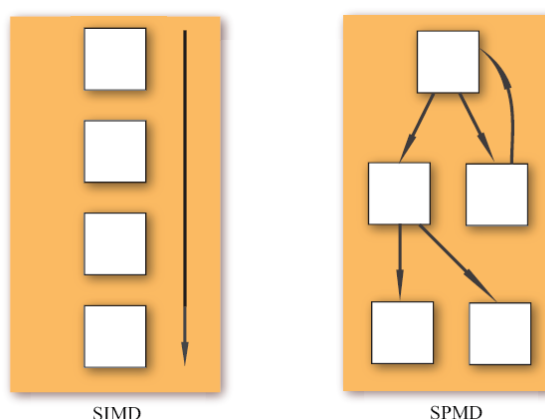


Figure 2.4: SIMD and SPMD from [23], with permission from NVIDIA

Recently CPUs have been expanding with parallelism and are now available with several cores. In this thesis, 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. This also introduces more advanced scheduling and flow control units, given that the different cores must co-operate and share data.

2.3.2 Main aspects of GPU architecture

The GPU is different from CPUs in that it focuses the use of the majority of its transistors for data processing and less on cache and flow control. Since it is mainly targeted for use in 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/neighbor 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 of 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 (e.g. tasks that have a lot of branching).

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. GPUs differentiate themselves from the CPU in that they focus on high throughput by using of many processors with low frequency. This also makes the GPU very efficient with regards to energy used (Performance per watt). Quite often in scientific and multimedia application one is to perform 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.

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

2.4. CUDA PROGRAMMING MODEL

as Cg, which introduced challenges since the programmer had to familiarize oneself 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 thesis, we will be focusing on the use of the Compute Unified Device Architecture (CUDA), a programming model for GPGPU developed by NVIDIA [23].

The newest GPU architecture recently released by NVIDIA for scientific programming is the Fermi architecture. The differences between this and the older architectures is outlined in Table 2.1. This new architecture is even used in the second fastest supercomputer in the world, where the Tesla c2050 model is used (we also use this in our tests), as shown on www.top500.org the official site for ranking.

Table 2.1: Architectural difference between NVIDIA CUDA architectures 1.3 and 2.0 (Fermi) as seen on www.nvidia.com

Specification	Tesla c1060	Tesla c2050 (Fermi)
CUDA Cores	240	512
DRAM	4GB DDR3	3GB DDR5
Memory Bandwidth	102 GB/s	144 GB/s
Data Cache	NO	YES
ECC	NO	YES
System Interface	PCI Express x16	PCI Express x16

2.4 CUDA Programming Model

CUDA (Compute Unified Device Architecture) is a parallel programming architecture and model, which includes a C compiler plus support for OpenCL, DirectCompute and others as shown in Figure 2.5. It is designed such that it naively supports multiple computational interfaces such as standard languages and APIs [23]. 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.

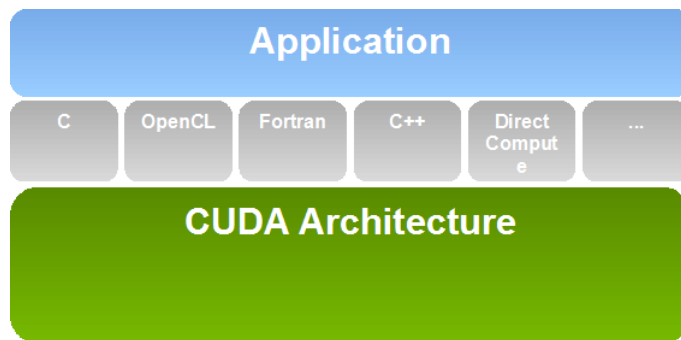


Figure 2.5: CUDA architecture from [23], with permission from NVIDIA

CUDA may be used as a C extension, so that those familiar with C programming can easily use it to accelerate their code using the power of the GPU. 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 coprocessors (GPUs). Each device is equipped with a large number of arithmetic execution units that has its own local memory, and runs multiple threads in parallel.

To invoke calculations on the GPU, one has to perform a kernel launch, which is a function deciding what each thread on the GPU is to perform. The GPU has a specific architecture of grids that are divided into blocks that contain the threads, see Figure 2.6. 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, where all blocks are the same size.

Which as mentioned, the GPU has 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 2.2 shows the relationship between these. Here the focus 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 [23]

2.4. CUDA PROGRAMMING MODEL

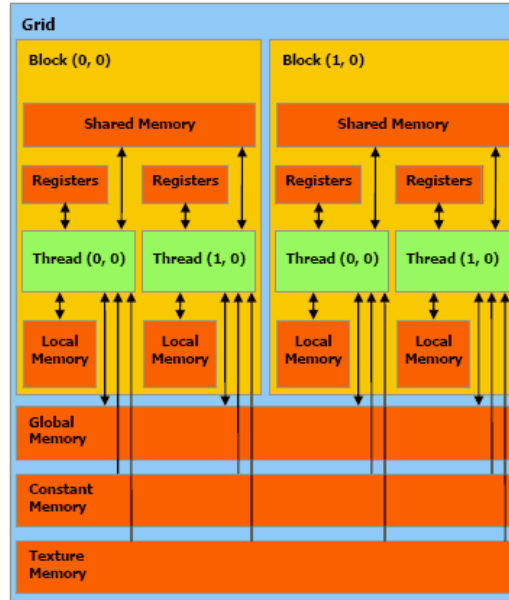


Figure 2.6: CUDA memory hierarchy and the GPU architecture from [23], with permission NVIDIA

Table 2.2: CUDA Memory Hierarchy, with data from [23]

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

Data Compression, Filtering and Seismic Data

In this thesis, we look at accelerating communication between the disk and computation units through the use of compression. Data compression algorithms are usually categorized into two, one being lossless compression and the other lossy compression. Lossless compression is the case where the compressed file does not lose any of the information of the original data. This is often used in compression standards such as *zip*, *rar*, *gzip* and others. Lossy on the other hand is compression where a loss of data is acceptable. This is often seen in image and sound compression where losing some data does not really effect the final outcome that much. Image compression standards such as JPEG and sound file formats such as MP3 use lossy compression.

In section 3.1 maps some important papers and findings in compression and also the literature that have been used to understand the fundamentals of data compression. Section 3.2, looks at several lossless algorithms. In Chapter 4, we adapt and optimize these algorithms specifically to seismic data. Section 3.3 discusses several lossy compression algorithms. Section 3.4, is an introduction to filtering algorithms used in the seismic process, mainly 3D convolution and the Hough transform. In Section 3.5, we introduce the structure of seismic data and the SEG-Y format is introduced.

3.1 Previous Work on Compression

Data compression is a well explored field, and many have contributed to it. Mainly there are two categories for data compression, lossless and lossy compression [24]. Lossless compression focuses on expressing larger word lengths with fewer bits. This can be done by simple methods that do not take into

3.1. PREVIOUS WORK ON COMPRESSION

account the type of data to be compressed, or more complex algorithms that analyze the data first. Fowler and Yaglet [18] show how lossless compression methods such as arithmetic encoding and Huffman encoding, that are part of the category of entropy encoding, can be used for volume data.

Entropy encoding is when one looks at the nature of the data, like how noisy the data is, before encoding, and try to find patterns that help in the process. An interesting method that has drawn some attention recently, is direct compression of scientific floating point data, as shown by Ratana-Worabhan *et al.* [37]. Here one uses adaptive algorithms that predict the next values given the nature of the previous values. These methods are good at giving good compression ratios, but are usually time consuming. This method has been tried on seismic data with good results by Xie and Qin [50]. They discovered that they are able to gain better compression rates with this method than with LZ compression methods, that are discussed by Salomon [40]. LZ adaptive methods are known for their good lossless compression rates are are per today used in general compression formats such as WinZip. The difference between the two is in the prediction algorithm for the next value and how they adapt.

Lossy methods gain greater compression rates by sacrificing some accuracy in the data. This is usually best done using transform coding, where one transforms the data to another domain such that it can be better compressed. These methods are used in many media formats that depend on compression of streams such as Mp3, JPEG and MPEG. Some of the first research done here is by Ahmed *et al.* [2], where they discuss using a discrete cosine transform (DCT) in signal processing. Watson [46], further shows that this transform can be used in compression, and introduced a method to do so. This method later on became really popular because of its efficiency and compact nature in the frequency domain, that faster DCT algorithms were developed.

One of the most popular algorithms is the AAN algorithm produces by Arai *et al.* [5]. This method is built upon the FFT and is therefore efficient. It is used in the JPEG format, as shown by Pennebaker [35]. Despite the efficient nature of the DCT, it has its limitations. The major known limitation is the blocking effect for well compressed data. Malvar *et al.* [30] [29], discuss using the lapped orthogonal transform (LOT) for compression to avoid the blocking effects in images. The LOT proved to give fewer errors for greater compression, but comes at a price of more computations when compared to the DCT. A fast LOT scheme is also introduced by Malvar *et al.* [29].

The LOT has gained some popularity in signal data compression, and its fundamentals are used to further develop algorithms like the GenLOT, introduced

by Queiroz *et al.* [12]. Here one uses the fundamentals of overlapping basis functions to overlap them not only with neighboring blocks, but with the n th neighboring blocks. This has proven to function well for compression, but lacks a fast implementation to make the process efficient enough to be used, and the algorithm requires a lot of computation when compared to the DCT and LOT. Nevertheless, Duval *et al.* [14] show how this can be optimized for seismic data. Generally all methods used to compress signal data are well applicable for seismic because it is also signal data.

In this thesis, the above mentioned mentioned lossless and lossy compression methods are explored for the purpose of accelerating I/O bandwidth and latency. We realize that some of these methods are time consuming to perform and expect them not to be suitable for I/O acceleration even if they result in good compression rates. Our attempt at accelerating these algorithms involves using the GPU and its computational capabilities, and exploring a scheme of CPU and GPU cooperation for efficient computations and compression. We are further developing these algorithms to specifically compress seismic data.

3.2 Lossless Compression Algorithms

Lossless compression is performed on a bit-wise level, where one is aiming at finding patterns in the input data that can be expressed in fewer bits than the original data. There are many categories and techniques to use for this task. The most common are RLE (run length encoding), Huffman encoding, dictionary lookup, LZ, and arithmetic encoding. Compression programs and standards such as *zip* and *gzip* usually use a combination. These methods are even present when using lossy compression.

In this section, we will give a description of common lossless compression method that are used in this thesis. Section 3.2.1 will introduce run length encoding and give some insight in its pros, cons, and implementation issues. In section 3.2.2, a description of the Huffman encoding algorithm is explained with emphasis of the different steps one must take and how they are performed. Section 3.2.3 gives a brief introduction to arithmetic encoding. While section 3.2.4 focuses on floating point data compression.

3.2.1 Run Length Encoding (RLE)

RLE is one of the simplest forms of compression, which works well on data with little noise [24][40]. And is often used not only in lossless compression, but also in lossy compression algorithms. Because of its simple nature it is very time efficient, but its major weakness is that its compression ratio is highly dependent on the input data.

RLE compresses data by iterating over all the elements and expressing the data as a collection of tuples of counters and values that expresses how many occurrences of the value are present in one sequence. For example, if we were to compress the string "AAABBAAACCCBBB" we would get the compressed string "3A2B3A4C3B". Here one can see that the string of 15 bytes can be expressed as a string of 10 bytes. This shows that RLE is a comparison of $n-1$ elements in an array of size n , which explains its efficient execution time.

However, as mentioned earlier RLE has a weakness of its reliability on the nature of the input data. In some cases, it can even give a larger compressed string than the original. For example if the input string is "ABDBAC", the compressed string would be "1A1B1D1B1A1C", which is twice as large as the original. This is of course an extreme case, but it shows the unpredictable nature of the algorithm, especially for noisy data. RLE is in the same way very efficient in an opposite scenario. If the input data is very similar then the compression rate is better than other algorithms. As an example if the input string is "AAAAAAAAAA", the compressed string would be "10A", which is one fifth the size of the original.

When it comes to implementing RLE, it is clear that it has a sequential nature. But, it can be easily parallelized with a little overhead in edge cases. The parallelization can be done such that each thread starts at a different position in the input data and start comparing values. The overhead is to compare the values where one threads ends and another begins to see if they are similar, and to merge the result arrays of each thread. If they are so, then they can be merged and the rest of the data can be shifted. Or if one is willing to sacrifice some compression in the edge cases to increase performance, one does not need to merge the values because it will be decompressed to the original data.

To decompress RLE encoded data one simply reads the tuples one by one and produces the value in the tuple the amount of times of the counter. For example, a tuple of "3A" is then decompressed into "AAA" actual three A characters.

Seismic data is generally regarded as noisy, and this might prove to be an

issue for RLE. However, since seismic data is expressed as a collection of floats and is noisy, but may look at fewer bits at a time, which would give more possibilities for compression.

3.2.2 Huffman Encoding

Both Huffman and arithmetic encoding are subsets of entropy encoding [24][40], which focuses on using fewer bits on frequent occurring values in the input data, and more on infrequent values. This way it is able to compress the input data by switching all the values with their new compressed values. The first part of Huffman encoding is to study the input data and create frequency tables to create a Huffman tree. The Huffman tree is denoting which bit value is given to a certain value given its frequency. This means that the second part is to actually build the tree. The final part of the algorithm is to use the tree to lookup and exchange values in the input data with corresponding values from the tree. Already here one can see that Huffman encoding can be computationally more demanding than RLE, but since the input data is analyzed before the compression it is always able to give a positive compression rate. See Figure 3.1 for illustrated flowgraph of the compression process.



Figure 3.1: Huffman compression flow graph

As mentioned earlier, the first step in Huffman encoding is to collect the frequencies of the given data set. The analysis can be performed on several data types given the same data to see if there is any statistical advantage for certain datatypes. This is similar to the scenario explained for RLE. The frequency of the values in a data set is directly related to the type of data one is looking at and the statistical variance of that data. This means that given similar data the frequency table produced should be somewhat similar. Since we are only looking at a specific type of data this step does not need to be performed every time the algorithm is executed, but rather the Huffman frequencies can be pre-calculated. This is done by studying several seismic data sets and comparing their results, which will be done in this thesis. This way we will get the compression efficiency of the Huffman algorithm and save time per execution.

The next step in the algorithm is to create a Huffman tree from the frequency data. This is such the most frequent value in the data set is the root node of

3.2. LOSSLESS COMPRESSION ALGORITHMS

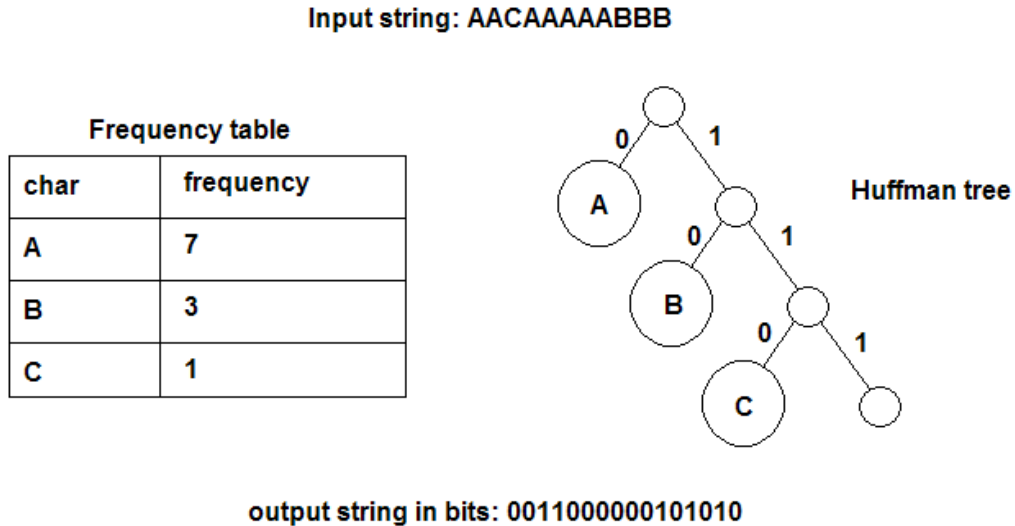


Figure 3.2: Huffman compression with illustrated steps

the tree and the least frequent value is the leaf node of the tree. See Figure 3.2 for tree generation example. The tree expresses how many bits should be used to express each value in the input data. This will result in variable bit lengths per value, but by being able to express the most common value with the least bits, one is able to compress the data. For example if one has an input string of "AACAAAA BBB", in the Huffman tree the char A will be replaced with a one bit value of 0. the char B will be expressed as a two bit value of 10, and the char C will be 110. Pragmatically this can be implemented such that the values are stored in an array and one can use direct lookup to avoid time spent on traversing the tree.

The final step of exchanging the values in the input stream with the values in the Huffman tree will produce the compressed output. In our previous example, the resulting output would be "001100010000101010", which corresponds to 2 bytes, thereby attaining one fifth of the original size of the data. The exchange can be tricky to implement efficiently because of the variable bit lengths. The biggest challenge here is the fact that there is no data type for bits. And therefore one has to use other data types and bit-wise operations such as shift, and, and or operations to process the exchange. This also means that there will be an edge case where some bits will spill over to the next value making it even more difficult to put together. This is also a problem when parallelizing because it requires that one shifts all the bits from one threads output to match the others when merging the them to one final output array.

This makes it hard to parallelize, but not impossible. There are also tricks to implement pre-calculated lookup tables to speedup the process, which we will be looking at in this thesis.

Huffman encoding gives generally good results, but does have its limitations and weaknesses. There is one major weakness in Huffman encoding and that is if the frequencies of the data are very similar, then the compressed data might end up being larger than the original file. This is because the values with lower frequencies are expressed with bits exceeding the datatype of the value. and if it has a quite high frequency it will result in compressed file larger than the original. This could be an issue for seismic data, but one can always change the granularity and get a new statistic. A further analysis of seismic data for Huffman can be found in the implementation chapter.

To decompress Huffman encoded data one needs the Huffman tree. It is used by looking up the original values given compressed bit patterns. this will have the same mapping as previously when the data was compressed. To do so one iterates every bit and everytime a 0 bit is detected, then by counting how many 1 bits were behind it one can find the character to replace the bit pattern. For example in the previous example if a 0 bit is detected and there were two 1 bits before it then the character to replace these is a C.

To summarize, Huffman compression is a good compression algorithm that takes into account the nature of its input data, and therefore is able to better produce compressed data. It has some weaknesses in that it is computationally demanding, because one has to iterate over the input several times. Once for analysis and another for compression, not to mention the time it takes to create the Huffman tree and the time it takes to perform bit-wise operations. Luckily there are ways to overcome these difficulties, which will be mentioned in more detail in the implementation chapter later on in this thesis. Another interesting aspect is the fact that Huffman is highly dependent on bit-wise operations that does not make it a good candidate for GPU acceleration.

3.2.3 Arithmetic Encoding

Arithmetic encoding is another form of entropy encoding, but differs from Huffman in that it accounts for more than one character at a time in the input string. It also has its roots in number theory, and aims at not wasting any bits in certain representations. Let's say one has three characters in a string in normal entropy encoding one would use 2 bits to represent the three, but one would waste a combination that will not be used. Since there are 4 possible

3.2. LOSSLESS COMPRESSION ALGORITHMS

combinations of 2 bits and we only need 3. A more efficient representation would be to represent the sequence as a rational number between 0 and 1 in the base of 3, where each digit represents a number. For example lets take the sequence "ABBCAB" its representation in the compressed form would be "0.011201". The next step would be to convert this to binary to have sufficient precision to recover it, which would be "0.0010110001". Then when decoding, knowing the length of the string is 6, by converting back to the base of 3 and rounding up to the 6 decimals one can reverse the process. See Figure 3.3 for flowgraph.

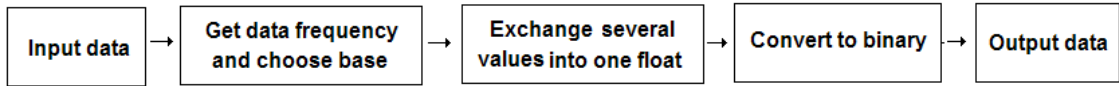


Figure 3.3: Arithmetic compression flow graph

3.2.4 Lossless Compression of Floating-Point Data

This is a method for compressing the data type of floating point by using the nature of the data to be compressed, which makes it fall into the category of entropy encoding. The algorithm is composed of three parts. First is the predictor, the second is bit-wise operations and last is compression. The predictor algorithm is to predict the next value in the input data given the N past values. Then the predicted value will be bit-wise XOR-ed with the actual value and compressed using any of the before mentioned algorithms to compress the amount of zeros present in new value. Since if the prediction is very close most of the values of the XOR-ed value will be zero. This means that the compression ratio is dependnt on having a good prediction algorithm, and on to have a good prediction one must use several previous values, which increases computations necessary to attain the predicted value. This is the major drawback of this method of compression. See Figure 3.4 for an illustration of the process.

The prediction algorithm works by taking looking at the differnce between previous data and the present value. How many values one wishes to look at will increase the complexity of the calculations but give better predictions and therefore better compressions. The prediction algorithm works like a hash function where its input is n previous original values and output a prediction value. it is important that it works in the same way in revers that given a predicted value and two actual values, the third actual value can be produced. The prediction function is very application and implementation dependent. In

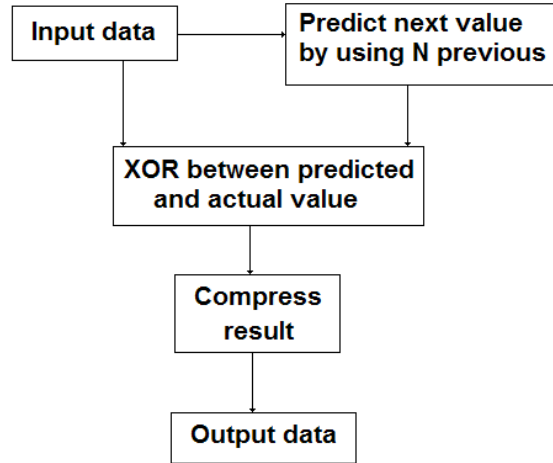


Figure 3.4: The process of compressing floats

the article by Ratanaworabhan *et al.* [37] they use the difference between the last 3 data points and perform shift and XOR bit wise operations on it to predict the next value. While in the paper by Xie and Qin [50] the prediction function is the third derivative of the last three values. Both had use arithmetic encoding to compress the resulting data set.

This algorithm is parallelizable in the sense that each thread can compress a set of the data and then they can all merge their results. But, border values of size N depending on how many values are used to predict must be exchanged between the threads. Here similar to Huffman since there are many bit-wise operation, the algorithm is not suited for the GPU, but is still parallelizable on the CPU.

3.3 Lossy Compression Algorithms

This section focuses on a lossy compression method that uses transform coding. Transform coding methods convert data into another representation that is more compressible. This means that after the data is transformed it is combined with other compression methods such as RLE and Huffman, which were discussed earlier. Transforms used in this thesis are the Discrete cosine transform (DCT), lapped orthogonal transform (LOT) and the generalized linear-phase lapped orthogonal transform (GenLOT). These algorithms are particularly interesting for images sound and other signal processing applica-

3.3. LOSSY COMPRESSION ALGORITHMS

tions because they give greater compression ratios than the lossless methods. However, this comes at a price of losing some data. The purpose of using such transforms is that we aim at using their properties and lose unimportant data, such as noise. In some cases losing a small amount of data can give great compression ratios, and this fits signal data well since it tolerates losing information without significant loss in quality. Seismic data is an example of signal data.

As mentioned earlier, the GPU functions as an accelerator that offloads computation from the CPU. To perform computations to convert to transform representations is very parallelizable and very adaptable to the GPU. There are three main reasons as to why the GPU is useful in this type of compression.

1. Signal data is often represented as floating point data and is not that sensitive to rounding errors. Modern GPUs have greater floating point capabilities than CPUs, but have limited support for integer operations, such as bit-wise shift. This makes the GPU a good candidate for transform encoding and a poor one for other encoding methods that are dependent on integer operations.
2. Transform encoding for separable transforms can be performed with coalesced memory access and the memory access is then sequential, but they require more computation. Since the GPU has more computation power than the CPU for very parallelizable problems makes it a good choice perform the computations. Other compression methods use auxiliary random access data structures. For example some compression algorithms use hash tables, such as floating point compression. The GPU has poor random access memory performance, and it is always recommended that one use coalesced and sequential memory access.
3. Transform algorithms mentioned in the transform coding are very data parallel where there are no dependencies between the blocks to be transformed. Thus the data parallel architecture of the GPU is well suited for this task.

3.3.1 Compression Using Transforms

There are 3 steps in the process of compression in transform coding. The steps 1) transforming data, 2) quantization of the data, and 3) encoding the data. This process works in reverse as well when one decodes the data by running the process in reverse. But, because of the quantization process, the data

decoded results in lossy data. We will now explain the three steps in more detail. Figure 3.5 shows an overview of the transform coding process.

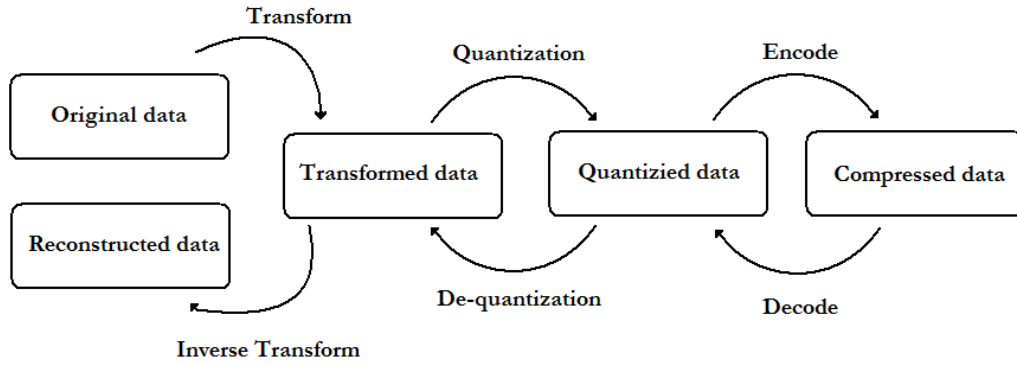


Figure 3.5: The transformcoding process, inspired by [27]

Transformation

The first step in the process is to transform the data from the spatial domain to the frequency domain. The purpose of this transformation is to represent the data in a more compression friendly state. One of the most popular transforms in image and signal processing is the Fourier transform, which transforms data to the frequency domain. There are other alternatives to this transform, which also use the same basis that is to transform to the frequency domain. The one that is most popular in image processing now is the discrete cosine transform. This is a transform that only looks at the real values of the Fourier transform and is used today in many DSPs and other encoding/decoding software and hardware such as the JPEG standard [35]. The advantage of the DCT is that it is faster than Fourier to compute and give similar, but not as accurate, results. This has resulted in the emergence of faster DCT algorithms such as the AAN algorithm that we will be looking at in this thesis, and later discuss in detail how it works. To accelerate the process it is common to filter parts of the image at a time and not the whole image at once. The common block size is 8 values. We have used this block size when testing all the seismic data in all three dimensions. It is also notable that the common block size is 8, when the optimizations for fast DCT and LOT are optimized for that block size.

Quantization

The next step in the process is to use the new domain to quantize the data such that it is even more compressible without losing too much of the important data. In image filtering it is not uncommon to use transforms to filter images using low/high pass filtering where all high or low frequencies in the frequency domain are removed. Low pass filtering is used in the JPEG standard and is what we use in the quantization process in our work. This results in removing noise/blurring the image and this results in larger compression with little loss of important data, since one is removing noise from the image. This method of filtering is common in all orthogonal transforms such as Fourier, LOT, GenLOT and DCT. There are other ways of quantifying data, but the main purpose is to increase compression with little loss of the data. One can even try to even out the number by rounding them off to the nearest neighbor to increase efficiency in encoding when using RLE.

Encoding

The final step is to use compression methods such as Huffman- and RL- encoding to compress the transformed data. This implies that the optimizations done to such encoding algorithms must match that of the quantization process. Such lossless compression methods are discussed in earlier in this chapter and are used in the JPEG and other standards. In particular, one has arithmetic encoding is used on floating point seismic data.

3.3.2 DCT (Discrete Cosine Transform)

The DCT was first introduced by Ahmed [2] in 1974, is a variant of the Fourier transform, which is more well known. But, this transform is often used in compression because of its attributes of being fast and performs well in the frequency domain for filtering images and sound, and is used in standards such as MP3 audio format, MPEG video format and the JPEG image format. There are four basic variants of the DCT [27][16], namely DCT-I to DCT-IV. The one most commonly used and referred to as the DCT is DCT-II, and where DCT-III is actually the inverse. DCT-I is a DCT that does not account for all boundary conditions. While DCT-IV is actually an orthogonal, and thus symmetric, representation and is therefore its own inverse, which is adopted in the fast DCT algorithms. For more details regarding the DCT, see Appendix A.

3.3.3 Fast DCT: The AAN Algorithm

The direct approach to solving the DCT will be implementing the mathematical definition discussed in the previous section. This gives an asymptotic running time of $\Theta(N^2)$ in the one dimensional version. This is because there are N elements and each takes N time to compute. It is a known fact that the FFT has an asymptotic running time of $\Theta(N \log_2(N))$, which means that there is luckily a faster approach to the DCT as well. As we already discussed the DCT is similar to the DFT, but less complex since it accounts only for cosine functions and there are no complex numbers. In this section we will discuss an algorithm tailored to solve the DCT that is derived from the FFT, which results in a faster DCT/IDCT algorithm. This algorithm is known as the AAN algorithm, and is developed by Arai et. al [5], and is also described by Pennebaker [35]. For more details regarding the AAN, see Appendix A.

3.3.4 Lapped Orthogonal Transform

When discussing the DCT we mentioned certain limitations that the DCT has, where one of them is the blocking effect. This was a result of the non overlapping basis functions. Malvar and Staelin [30], and in Malvar's book [29], discuss using a Lapped orthogonal Transform or LOT to solve for this effect. The foundation of this work is built upon overlapping the basis function of a orthogonal transform such as the DCT. Malvar [29] also produced an efficient algorithm/scheme to solve for the LOT. In this thesis we will not attempt to explain in detail the mathematics behind the LOT, which is done quite well in [30], but rather we will explain the definition and the scheme presented in [29] and how it is implemented. Appendix A includes the definition and flow graphs for LOT implementations.

3.4 Filtering Algorithms

In this section, an introduction to two widely used seismic algorithms will also be presented, convolution and the Hough transform. The algorithms are not specifically developed for seismic processing, on the contrary they are quite common in image processing, but we call them seismic algorithms because they are used in seismic data pre-processing and we would like to put them into contexts when we later analyze performance.

3.4.1 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 3.1 [36].

$$(f \star g)(t) = \int_{-\infty}^{\infty} f(a)g(t - a) da \quad (3.1)$$

$$w(x, y) \star f(x, y) = \sum_{s=-a}^a \sum_{t=-b}^b w(s, t)f(x - s, y - t) \quad (3.2)$$

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 into 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 3.2 [36], which is a discrete summation equivalent to the continuous mathematical definition of convolution in two dimensions.

When filtering using convolution we mentioned involving two functions. In practice, this corresponds to using one function that expresses the filtering mask to be used and the other being the original data to be filtered. The filtering 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. In this thesis, we will be using a Gaussian filter, also known as Gaussian smoothing. 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.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (3.3)$$

The Gaussian distribution in the 2D case is shown in Equation 3.3 (from [36]), where σ is the standard deviation of the distribution. The function is illustrated in Figure 3.6. this can be used to give a better idea of what a Gaussian distribution is and looks like. The idea of Gaussian smoothing is to use this distribution 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 3.7 to see an example of a discrete approximation of a gaussian filter in two dimensions.

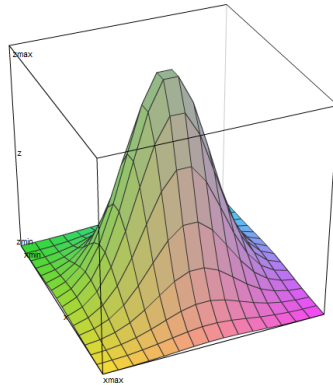


Figure 3.6: 2D Gaussian distribution graph drawn in online 3D grapher ¹

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

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

3.4. FILTERING ALGORITHMS

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	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 3.7: Example of two dimensional Gaussian Filter Mask with discrete values

3.4.2 Hough Transform

The Hough transform is used in image processing to detect to which extent a set of points in an image are on a line, and can be used to detect lines and curves. This is seen in Duda and Harts paper [13], where they show the use of the transform to detect lines. This is used in seismic when looking for faults, where the points are selected by the user. The transform, transforms data from the spatial domain to a domain expressed by angles and length of a point from the origin, θ and r respectively.

$$r = x\cos(\theta) + y\sin(\theta) \quad (3.4)$$

Now consider a straight line in the spatial domain. The line can be represented as the equation $y = mx + b$, where m is the gradient and b is the y-intercept. For a given point on this line one can define several lines that this point lies upon by varying m and b parameters in the equation. Due to the difficulties of representing lines with an infinite slope, it is more suitable to represent them as functions of r and θ as in Equation 3.4.

Here x and y are the coordinates in the spatial domain for the point that will be evaluated. This way one would be varying r and θ rather than m and b .

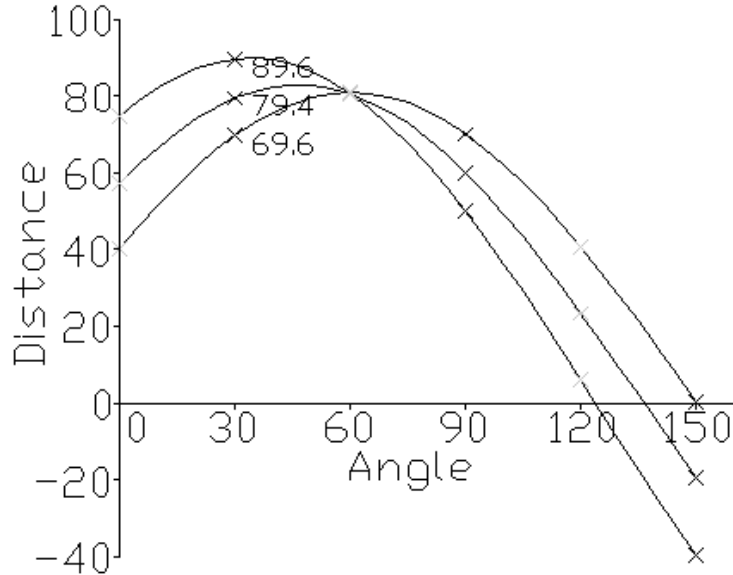


Figure 3.8: Three point represented in the (x,y) - space and (r,θ) -space, obtained from ²

Thus a single point (in the Cartesian space) on a line would have a whole range of r and θ values that map to a curve in the (r, θ) parameter space. As shown in Figure 3.8.

Now lets discuss how one is able to use the (r,θ) -space of the Hough transform to find lines in an image. First lets see what happens when we introduce another point to the domain, as shown in Figure 3.8. As one can see that the sinusoidal curves of the two points intersect each other. And at the point where the curves intersect is when the value of r and θ is similar for both points meaning that they are on the same line for these values. One can take this further and introduce more points and one would get more intersections representing several other lines between points.

By increasing the intensity in the image in (r,θ) -space one can see which lines incorporate most of the lines because they would be the most visible and one would have more lines present in the image that are represented in fewer intensities. This is best shown with an example such as in Figure ??.

²http://en.wikipedia.org/wiki/File:Hough_space_plot_example.png, accessed 2010-09-05. Available to all under the license of wikimedia commons, a freely licensed media repository

3.5 Seismic Data

In this section, the focus will be on presenting seismic data and pointing out the most important traits that effect computations and data processing. Seismic data is usually presented and formatted in the SEG-Y format.

SEG-Y, is a data exchange format for seismic that was developed in 1973, was first published in 1975. It has achieved a wide spread within the field of geophysics since then. The original structure of the format was developed for digital tape media. As the hardware used for seismic data acquisition has changed so has the format. The newest revision was released in May 2002 [41].

One of the main traits of the format is that it separates between traces of seismic data as they are acquired. A seismic trace can be viewed as a one dimensional vector into the ground. Looking at this in a three dimensional Cartesian coordinate system, a trace is all values in the z (depth) dimension given an x and y coordinate.

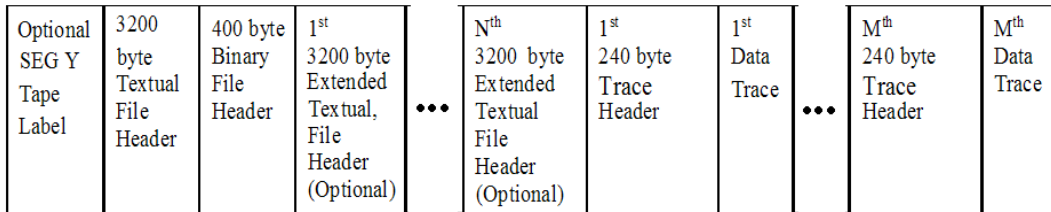


Figure 3.9: Byte stream structure of the SEG-Y Format with N textual headers and N traces and trace records. Image inspired from [41]

The format can be expressed as seen in Figure 3.9. Here one sees that the format consists of different header files and data traces, where each data trace has a header. The header files for each trace include information on details regarding the trace such as the length and position of the trace. While the trace data is a collection of floating point data. The file header includes information such as the cube dimensions, which company the seismic cube belongs, and other geophysical information. The important thing here is that when reading from disk the locality of the data is near and traces can be read sequentially giving a good balance between control and performance. Control comes from the header files and performance is reached by keeping the data locally close on the disk.

In the case of data compression, the header information is not that important since we are aiming at compressing the actual seismic traces. That is why

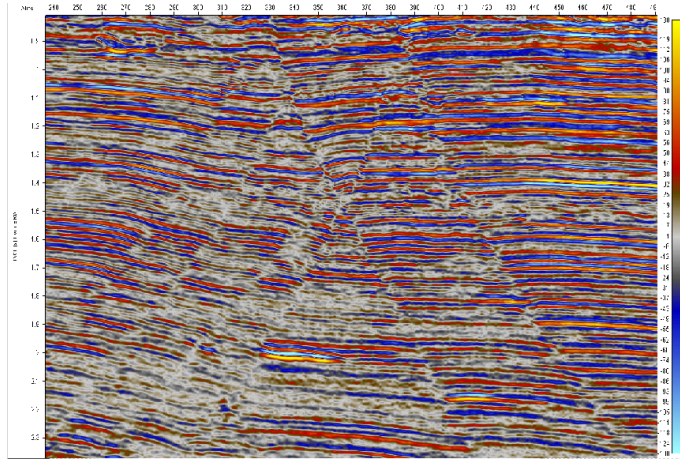


Figure 3.10: Seismic data example made available by Schlumberger from [1]

they can be ignored if we know the dimensions of the seismic cube. This will cause the reads from disk to be partitioned into traces, and one would have to jump over the trace header to read the next trace. One might think this would be an issue, but as mentioned before since the traces are so close to each other on the disk no delays will be experienced. For large datasets this will depend on the blocking method used to read from disk. This will be discussed in further detail in the implementation chapters and empirical tests will be run to determine the efficiency of different blocking. Previous work on blocking of seismic data is done in [4], which shows that blocking along the y or z axes of a cube (if each trace represents data in the x axis) are the most optimal because it gives the most sequential reads from disk. Figure 3.10 illustrates a 2D slice of seismic data, which gives an example of how seismic data looks like.

3.5. SEISMIC DATA

CHAPTER 4

Compression and Filtering Algorithm Optimizations and Implementations

In this chapter, describes how we implemented and optimized several of the compression and filtering algorithms described in Chapter 3 and Appendix A. Our aim is to not only see how compression effects I/O, but also the entire process including filtering and processing the seismic cubes. A discussion of our implementations here analyzed and optimized using profilers directly effect the implementations. The profilers used in this thesis are the NVIDIA CUDA profiler for the GPU implementations that are developed on NVIDIA hardware, and Intel Vtune profiler for the CPU implementations that have been developed on Intel hardware. Other than these algorithms a framework has been developed in this thesis to produce seismic images and to expand on given data to make it larger because of the need for larger datasets than those made available. The framework has been used in both debugging and verifying results qualitatively.

In the first section, Section 4.1, we will be explains the file compression format developed. Section 4.2 will be discussing the frameworks that we use to produce images and to test our work. In Section 4.3, the implementation and optimization of the lossless algorithms are discussed. While Section 4.4 discusses the lossy compression algorithms and how they have been implemented. In Section 4.5, we will be discussing implementations of the image processing algorithms, which in our case are 3D convolution and the Hough transform. finally, Section 4.6, briefly discusses the AESC library developed.

4.1 The File Compression Format

SEGY is a standard format for seismic data, but it is a hard format to compress and is slow in the sense that there are many headers that need to be read and a lot of jumps are performed on disk thus making file access slow. Our implementation considers normal I/O as just reading data in a sequence without these jumps. That is why we extract the seismic traces and created a file containing only the seismic data. We then developed a custom file compression format.

The compressed file format is shown in Figure 4.1. It starts with an unsigned integer that shows the size of the dictionary of the optimized RLE is. After the dictionary we will have a unsigned integer to show how large the next block of data is knowing that the decompressed data that was compressed is compressed 2GB at a time. This integer shows how large the compressed block is, and this is needed because the compressed length is unpredictable. This is the same format that we used for compressed lossy algorithms since they are combined with RLE.

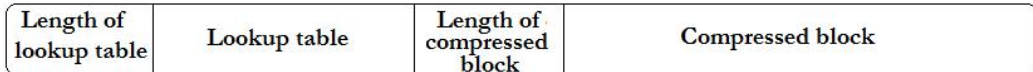


Figure 4.1: Illustration of our compression format of the RLE algorithm

The size of the pre-compressed block can be varied. To being with, we used a size of 2GB for the compressed block. But, when we have a filtering algorithm with overlapping sub-problems that will use this compressed data, it would have to decompress all the neighboring blocks to get the data. This is an issue because one does not have enough memory for all neighboring data, and one is decompressing large amounts of data that is not used. This issue can clearly be solved by using smaller blocks, and with asynchronous reads this is also an advantage. The disadvantage though is that too many small blocks will give lesser compression. Since data strings of zeroes that can be compressed to 2 bytes are cut in two then they will be expressed with 4 bytes, and so on. This means that one can miss-use the size of the compression block to the detriment of the compression rate.

Another compression format implemented is one that includes neighboring data. This of course gives an overhead for filtering algorithms that do not have overlapping sub-problems in that one decompresses much more data than

needed. But, this gives overall less overhead than having to decompress many neighboring data blocks. We will test all these options and discuss them in detail in Chapter 6.

4.2 Our Testing Frameworks

There are two Frameworks developed as part of this thesis. The first is a framework that is mainly used for black box test, and has the main task of producing images by reading from the a result file from the last run application. This framework is also used to produce seismic cubes of varying sizes for testing. The other framework mentioned, is one that is used mainly to test the code and algorithms written and to benchmark them. In this section we aim at giving some insight at what these frameworks do and how they are used.

4.2.1 Producing Images

At the start of the thesis 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 4.2 for an illustration.)

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 correct. Instead modifications were 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 not a necessity because usually one only needs one image in each dimension to see if the algorithms are performed successfully, and the results are correct. The original image producing algorithm has poor 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 thesis, the only modification made is the selection of single images and no optimization was done to the image production algorithms.

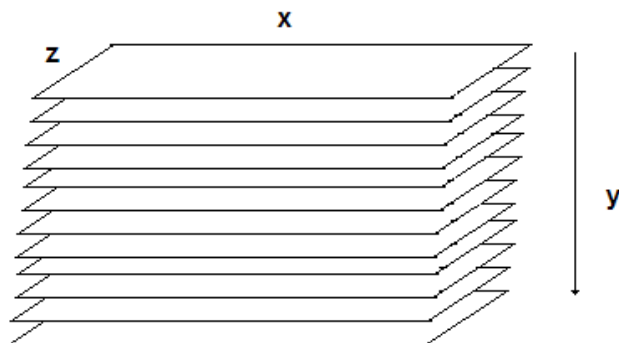


Figure 4.2: Image illustrating slices of x,z dimensions along the y axis

4.2.2 Producing Seismic Cubes

However, the framework is not only used to produce images, but also produce different sizes of seismic cubes for testing. The data received for testing in this thesis was of sizes between 256MB to about 1GB, but this is not enough to test for large data sets. Our definition of a large data set, is a set of data that does not fit into memory, and results in forcing the computation to access the disk several times. This is important such that the effect of I/O is present in the benchmarked results. The largest set we are testing with is a 32GB seismic cube that we created by combining and rotating the cubes we obtained. We also tested with smaller sets ranging from 256MB to 32GB to see if there is a pattern, and this would help when testing our prediction model.

The main aspects worth noting when speaking about the data sets is that it is important that they be actual seismic data. This is because when compressing data one is looking at patterns that can be expressed in a more efficient way. This gave us a chance to look at seismic as a certain type of data, which can be categorized as noisy signal data, and revile some its patterns and work on compressing them. When constructing the data sets we first took the original data and started rotating and combing it to obtain larger data. What is important to note is that we in this way are able to sustain the variance of this type of data and in this way obtain legitimate results. Another point worth mentioning is that this data is then first compressed and converted to our compression format before testing. To see the source code for the cube producing algorithms see Appendix D

4.2.3 Synchronous and Asynchronous I/O

Other than the framework for producing images and seismic cubes, we developed a separate framework such that we can test our code. This framework has two parts one being the synchronous main method that uses the algorithms from our compression library and test them, and the other is the same but now performed asynchronously with two threads.

The synchronous method is a single threaded main method that performs task in a sequential nature in that it runs first the I/O process of actually reading the compressed data and then decoding it, which then is used to filter, only to be encoded and written back to disk. When testing we time this process and compare it to doing the same with the non compressed data in the sense that it is read, filtered and written back to disk.

The asynchronous method is where we use two threads that run in parallel, where one thread reads while the other decodes at the same time. And to insure that there is no conflict we have a barrier between each step such that one does not start to decode until the data of the next block is read and vice versa. We also do the same when it comes to encoding and writing the data back to disk.

4.2.4 Benchmarking Framework

When benchmarking results it is always important to note, which method is used to time ones process and the accuracy of that method. Since our process take a long time to perform, they do not require millisecond accuracy, rather a time taking method that is able to be accurate for longer tests. When one tests code that lasts for hours a millisecond or two is not an issue. This is why we have used wall time clock because this shows how long time one has actually used and not calculated from how many cycles one has performed on the CPU or other methods like that. This will of course give some variance, but nevertheless the most accurate time as to when it comes to how much time one has spent if one is looking at ones wall clock.

We have also made a point of measuring time for separate processes separately rather than just noting the total, we do note the total time as well. The records of all tested material is found in Appendix C. We have also included the most interesting results in the chapters to come where we actually discuss these results and their influence on the seismic process.

4.3 Optimizing Lossless Compression Algorithms

In this section, we will discuss optimizations of the two lossless algorithms implemented in this thesis. First we will have a look at run length encoding and the second is Huffman encoding. We have discussed arithmetic encoding as a method, but chose not to implement it because this is already done lately by Xie and Qin [50], and because of the limited time for this thesis we will use their results to compare in the results chapter rather than presenting our own. Another reason is that after comparing their results to our own we see no reason to redo that work.

4.3.1 Optimizing RLE w/ Dictionary lookup

The seismic data read is a stream of floats, and floats can express a larger range of numbers than integers, short and char data-types. The interesting fact is that a float also uses 32 bits just like integers. And when using RLE encoding it is smart to limit the number of values one has to count. Because as we explained before RLE relies on counting the occurrences of a number in a row. Knowing that seismic data is noisy data, one can assume that the values of the floats changes rapidly and little recurring values are there. This is why we choose to read the data into unsigned char data type to limit the values to 256 values. After studying the data, which we show later on when discussing the Huffman encoding, it shows that when looking at the data on a byte wise level there are a lot of recurring zeros. That is why after testing the compression rate, we discovered and decided to compress on a byte level. Where we used one byte to count the recurrence and the other to represent the value. A pseudo code version of the algorithm implemented is as follows, for detailed implementation see Appendix D.

```
1 Input data X
2 Result table R
3
4 for each bytes in X
5     if current_value == next_value
6         counter++
7     else
8         add counter to R
9         add value to R
10        counter = 1
11        current_value = next_value
```

Pseudo code for naive RLE algorithm

The best results for compression for the basic RLE algorithm is a compressed file larger than the original. This means that it did not succeed to compress. This comes from the fact that seismic data is too noisy and there are a lot of changes. But we made an optimization by using a dictionary lookup technique and combined this with RLE. To avoid using a byte for the value and a byte for the counter when we had many values that occurred once and therefore we are exchanging a byte of data with two for those values. We solved this by targeting all the zero values because they are the values that are recurring. This way we have a dictionary that is 1/8 of the file and here we traverse this on a bit-wise level. If the value is true or "1", then the value in the corresponding byte-wise stream is a counter for the amount of zeros. If the value is false or "0", then the value in the corresponding byte-wise stream is just another value than zero. This way we were able to compress the data to 80% of its original size including the library being part of the file. This is a big change given that the data is noisy.

As a first step we expanded our implementation to a multi threaded implementation that uses OpenMP such to utilize the quad-core CPU computation power. It is here we realized the sequential nature of compression. The point here is if we are to divide the original data into several threads and compress them individually we would have to shift all the data in the compressed data to combine it with the rest of the compressed data. This means that if we use many threads on a platform such as the GPU, then we would have to have threads that wait for their turn to combine the sub problems. This might not be a big problem for platforms that do well with bit-wise operations like the CPU. But, platforms like the GPU, that are not that efficient on bit-wise operations will suffer and are limited by the sequential part that demands a shift of all the data that comes after the first block.

Table 4.1: Intel Vtune results for RLE

L2 cache Miss	L1 cache Miss	Branch Mispred.	Prefetch Miss
0.3%	0.1%	2.6%	3.4%

For the Quad-core implementation we split up the data into four sub problems, and compress each for its self. Then at the end a single thread is responsible to combine the compressed sub-problems. When testing for optimizations for things like memory efficiency, caching and cache misses. We observed that the algorithms have no problem with this when using Vtunes to test for this. A table showing these results is shown in, Table 4.1.

4.3.2 Optimizing Huffman Encoding

When implementing Huffman encoding one usually needs to read the whole data file twice. Once, to analyze the input and create a Huffman tree for the substitutions, and another to actually perform the substitutions. In our case since we are only considering seismic data, we can pre-process the tree and use the same one for each calculations hence saving computation time and reducing I/O time by two. To perform this approach we would need to analyze the recurrence of the data in our seismic examples.

When analyzing the data we looked at several aspects. One of which is the length of the word we are looking at. As we have mentioned earlier about the Huffman encoding, we would like to minimize the variable lengths to be able to compress well. In other words if we only look a word length of 2 bits then we only have 4 combination to account for, and thereby is one of the 4 options is more dominant than the others we would gain greater compression. When studying the data we analyzed word lengths of 16-, 8-, 4-, and 2-bits. The graphs below in Figure 4.3 show the distribution between the values that occur in the seismic data.

The data analyzed is an average from all the different sets given by Schlumberger and Statoil, which adds up to 4 sets. We noticed that there was little variation between them and an average of the data would be representative for seismic as a whole. Knowing these numbers one can pre-calculate the compression possibilities for each combination and figure out which is most effective. This is done by replacing the most frequently occurring value with 1 bit and the second with two and so on. Then by summing this and comparing it to the original size of the data we get a compression ration that represents a good estimate of what to expect when using one combination or another. The results show that the most efficient is a 2-bit word length. Which is able to compress up to 72% of the original size. These results are represented in Table 4.2.

Table 4.2: calculated compression rates from study of data of 2GB

Bit length of analysis	Compressed size in GB	Compressed zise %
16-Bit	3.74	183
8-Bit	1.65	81
4-Bit	1.73	85
2-Bit	1.46	72

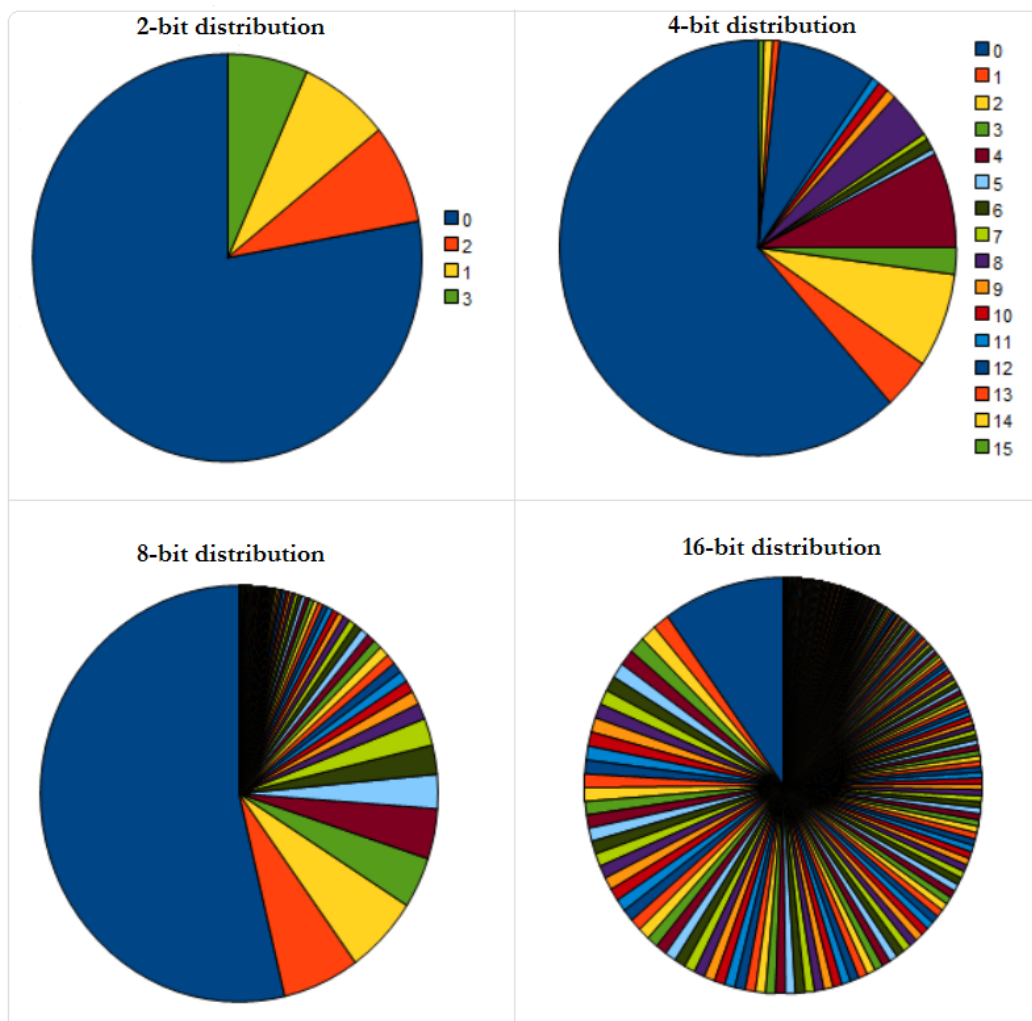


Figure 4.3: Graphs of data distribution of seismic data given values of length 2-, 4-, 8-, and 16-bit

4.3. OPTIMIZING LOSSLESS COMPRESSION ALGORITHMS

Now that one knows the distribution and since the tree is going to be so small one can use a direct look-up table (array) instead of generating a tree. This is good because it will give a great speedup in the computations not having to traverse the tree for every look-up when substituting the values. To avoid having to traverse the input data 2-bits at a time, one can pre calculate for permutations of the 2-bit substitution for larger words such as 8-bit or 16-bit. This will not change the compression results, but rather sacrifice a little pre-calculation time to accelerate the total traversal time of the input data. Test results show that performing the traversal for each 8-bits with the overhead of pre-calculating the Huffman table is the most efficient for smaller cubes. While for larger cubes it is a 16-bit traversal of data-type short is the most effective for larger cubes.

```
1 Input data X
2 Result table R
3 Lookup table for values T
4 Lookup table for lengths TL
5
6 for each INT in R
7   value = data from X
8   Get replacement value from T
9   Get length from TL
10  Shift bits to match length using TL
11  OR with value from R
12  Counter for amount of bits += value from TL
13  if(Counter > 32)
14    Get rest code
15    Get rest length
16    Add result INT to R
17    shift rest code
18    Or rest code with result INT
19    update Counter with rest length
```

Pseudo code for naive RLE algorithm

Pseudo code for the implementation is shown above. Here we aim at showing how we create the tree and look-up table for the substitutions, and how the table is actually used. Here as well as in the RLE implementation we have a sequential dependence. This dependence is the fact that we do not know how large a block will be when encoded and therefore when putting together parallelized encodings we need to do so sequentially. When one uses a limited amount of threads such as on the CPU it is easy to control that all do the sequential bit and one thread will not be waiting too long, but for the GPU implementation it might be more consuming since there are more checks and more pieces to put together. One of the reasons why the GPU implementation

was outperformed by the Quad-core implementation. Again another reason as to why this algorithm might not be as efficient on the GPU is because of the use of bit-wise operations.

Our OpenMP implementation of the Huffman encoding follows in the same fashion as in the RLE implementation. The input stream into 4 and start compressing the 4 arrays separately to then use one thread to put these together. The problem with this implementation is the sequential nature of having to shift all the bits when putting the array together to one compressed data set. The main issue is that the length of the compressed sub arrays is unpredictable, and thereby one cannot estimate where to start writing the next array in the sequence. Therefore one just has to put these together after having compressed since then one can find the length and share it with the other threads. One can argue for using a tree structure such that puts together several parts of the original array together at a time, but the amount of bit operations or shift operations performed will be the same. Since even if one has put together another part, there are still the same amount of elements that need to be shifted corresponding to the length of the first block that they are supposed to be put together with. That is why it does not matter if one uses a tree structure or not. On these basis we did not use a tree structure and rather had one thread perform a sequential combination of the sub domains, which we discussed is just as efficient.

When optimizing one has to look at memory access efficiency and memory bandwidth use. This is assure that the implementation is optimal and that the comparison further on with GPU would be representative. Our implementation showed to be memory optimized, and by testing it with the Intel Vtune profiler we obtained the results shown in Table 4.3. As one can see there are close to zero cache misses and there are close to no prediction misses.

Table 4.3: Intel Vtune results for Huffman encoding

L2 cache Miss	L1 cache Miss	Branch Mispred.	Prefetch Miss
0.5%	0.3%	2.7%	3.2%

We adapted the implementation of Haugen [20] for the GPU Huffman implementation, and is why we achieved similar results. We did not find any major weaknesses in that implementation such as wrong use of memory banks in the GPU architecture. The only advantage we had over Haugen is the hardware used. Since we are able to test on the new Fermi architecture, we have more cores to use for computations. However, this does not scale well for Huffman

due to the sequential dependency between the threads.

4.4 Optimizing Lossy Compression Algorithms

When implementing lossy compression we have focused on transform encoding, which is typically used for signal processing and image processing. Seismic data is categorized as signal data in the same way ultra sound is. There has been work done with similar methods with the aim at maximizing compression of seismic data, such as the work of Duval *et al.* [14]. Here they showed how well transform coding with lapped transforms is for compressing seismic data. Our case differs in the sense that we aim at not only compressing more, but we need to do so quick enough to beat I/O time, which sets different conditions for our thesis. Whats interesting is all lapped orthogonal transforms build upon the DCT, which is an orthogonal transform with no overlap. That is why we had to have a DCT implementation. We further on implemented a faster DCT algorithm which is based on the FFT algorithm. Finally we tried to expand to the LOT algorithm, our implementation was fast, but did not give the desired effect on seismic data and was not better than the DCT, which is why we stopped expanding on it. Because for our purpose the fast DCT AAN algorithm was the most optimal.

All the implementations referred to above are performed on both CPU and GPU for comparison and speedup analysis. Since the transforms are typical tasks to perform on the GPU and are often parallelizable, we believe expanding these to the GPU will give great speedup. When perform transform encoding we need to use one of the lossless compression algorithms to compress the transformed data. As we discovered in the previous sections, a pure GPU implementation of the RL or Huffman encoding are not as efficient as on the CPU. That is why we have set up a division of the tasks where the GPU and CPU co-operate. The GPU is then responsible of transforming the data and the CPU is responsible of compressing them using RLE. We chose to use RLE because the results show that it is the most effective both in compression rate and execution time on transformed data. The same does not apply on non-transformed data. In other words our scheme implements transform encoding by using both the GPU and CPU and have them cooperate. When this is performed on smaller blocks of data it is run asynchronously.

In the next sections we will show in more detail the three transform implementations we have performed to perform the encoding, but will only refer to

pseudo code to illustrate our point. For the actual source code see Appendix D.

4.4.1 Optimizing the Naive DCT Algorithm

The naive implementation of the DCT is basically implementing the formulas presented in Appendix A. This implementation was not too challenging because it requires only a double for loop. What is nice with the naive DCT is that one can decide the block size of each DCT block. In contrast to fast implementations which have been optimized or given block sizes, like the AAN algorithm is optimized for a block size of 8. When testing we set the block size to 8 here as well for a legitimate comparison. A pseudo code implementation is shown below to give an idea of how this is performed.

Another aspect that is important to mention is that the DCT is separable and therefore one can filter one dimension at a time. We implemented the naive DCT up to three dimensions. An interesting finding is that because of the amount of computations that have to be performed the three dimensional implementation could not beat I/O time even though it gave good compression rates. Which is part of the motivation for implementing a faster DCT algorithm. This is because the compression ratio is directly linked to maximum I/O speedup achievable.

What is really nice about the DCT algorithm is that it is very parallelizable since each 8 pixels are transformed from one domain to another independently from the others. This made parallelizing the DCT algorithm on the CPU and GPU easier. The way it was done is that each DCT block, a collection of 8 pixels, is transformed by a thread at a time. And this was done one dimension at a time. The naive implementation has little variables and therefore uses few registers. We checked for memory optimization and prefetching optimizations for the algorithm to avoid the memory bandwidth bottleneck. When testing our naive algorithm with the Intel Vtune profiler, we discovered that there was no need to optimize for this because it was already optimal the way it was performed earlier. This comes from the memory access pattern of the algorithm. In that it analyzes 8 floats at a given time, and this makes the prefetcher be able to do the job well. Another thing is that the algorithm performs in one dimension at a time which gives room for less jumping in memory, but when running across the crossline or depth dimension there are jumps in memory. These jumps are luckily of constant size and the prefetcher is able to fetch the data with little misses. The results of the profiler for the two

4.4. OPTIMIZING LOSSY COMPRESSION ALGORITHMS

dimensional quad-core implementation are displayed in Table 4.4.

```
1 Input data X
2 Result table R
3
4 for each 8 floats in X
5   for each of the 8 floats
6     Temp value = 0
7     for each transform value of the block of size 8
8       multiply corresponding value form X with cos() value
9       Temp value += calculated value
10    if(first block)
11      scale by square root of 0.25
12    else
13      scale by square root of 0.25 divided by square root of 2
14 }
```

Pseudo code for naive 1D DCT algorithm

Table 4.4: Intel Vtune results for Naive DCT transform encoding

L2 cache Miss	L1 cache Miss	Branch Mispred.	Prefetch Miss
0.1%	0.7%	1.3%	2.4%

The GPU implementation as with the CPU, each thread runs a block of size 8 and transforms it. The memory mapping between the input data and the resulting transformed data is one to one unlike when compressing. This means that there are no collisions in memory access neither when reading or writing. This way each thread is very parallelizable and independent from the others. The most important part of the GPU implementation is the amounts of thread and blocks used. These decide one has good occupancy on the card or not, and if one is running optimized code. We are using a thread count of 128 threads per block and block count that is based on the y and z dimensions of the cube. But, given our thread count and register use we achieve 100% occupancy on the graphic cards pre-Fermi. On the Fermi architecture cards we run 256 threads to achieve 100% occupancy because there are twice as many cores. These numbers were calculated using the NVIDIA occupancy calculator [9], and later confirmed using the NVIDIA CUDA profiler.

4.4.2 Optimizing Fast DCT: The AAN Algorithm

After realizing that the three dimensional DCT implementation gave such good compression, about 17% of the original file, we had to find a way to accelerate

it such that it can be used to accelerate the I/O process. One of the most known fast DCT algorithm is the AAN algorithm that we discussed earlier.

The algorithm is best explained by a flow graph (see Appendix A), and to follow the graph for a 8 float block size one needs 16 variables. Eight for the current values that are to be calculated and eight for the previous values used in the calculations. By setting these to variables they are then used in the registry, which makes access both writing and reading take one clock cycle. This is the case for both the CPU and GPU implementation. The AAN algorithm needs only one read and write to system memory per float calculated. All the other calculations performed on that float are done in the registry. This is part of what makes this algorithm efficient. We again performed tests on the algorithm both for the GPU and the CPU with the corresponding profilers. The results for the Vtune profiler showed that the algorithm is memory optimal, and there are close to zero percent pre-fetcher and cache misses on all levels (L1 and L2). The results are shown in Table 4.5.

Table 4.5: Intel Vtune results for AAN DCT transform encoding

L2 cache Miss	L1 cache Miss	Branch Mispred.	Prefetch Miss
0.3%	0.2%	0.7%	1.4%

When this algorithm is adapted to execute on the GPU, we realized that since it is very parallelizable more time was used on copying the data to the GPU than actually performing the calculations. More accurately 60% of the time is used on copying memory while 40% on computations. This was the case for the one dimensional DCT. We then started to add more calculations by expanding the dimensions one is to calculate on. By expanding to three dimensions the distribution between communication and computation on the GPU is 20% to 80% respectively. Other issues regarding the GPU implementation are threads- and blocks- division and the work performed per thread. To avoid conflicts in memory and communication between threads, which create dependencies and slow computation time, we aimed at that each thread solves for a DCT block. Then to optimize for occupancy on the graphic card we used similar values as for the naive DCT, which we explained earlier. And of course the partitioning of threads and blocks differs for the Fermi architecture since it has twice as many cores.

When it comes to implementing the algorithm for different dimensions, the same one dimensional algorithm is used but in a different dimension. This means that we read the 8 consecutive in the height dimension or the depth.

This however changes the memory access pattern and makes it non coalesced. Since now we have to jump in memory. But since there is only one read and write to the global memory there is no way we can improve on it, which also means that it is optimal given the situation.

4.4.3 Optimizing Fast LOT

The LOT is built upon the DCT. As we showed before in Section 3.3.4 and Appendix A, the fast DCT requires that the data is transformed to the frequency domain using the DCT. That is why in our implementation we first use the AAN algorithm to transform the data, and then run our LOT algorithm to transform the data. But, there are some changes that need to be made to successfully implement the transform. One is that padding needs to be added to the DCT transformed data. The padding is added such that each line in each dimension has two empty blocks of size 8 are added, one at the beginning and the other at the end. This way for the one dimensional case we will see that the x dimension will be bigger by 16 elements. This can be an issue because this means that before compressing the data we are increasing its size to transform the data. And as one increases the dimensions for the transform the bigger the transformed cube will be. Another interesting aspect is that one could perform the transform with out the padding, which is often done for images [30], but in our case it effects the accuracy of our transform and this is something we want to decrease with the use of the LOT. That is why it is necessary to decrease the compression rate (increase in blocksize) to increase accuracy. One can also view the increase in accuracy as removing the blocking effect.

The LOT implementation was tested as we did with all other implementations, and the tests for the first implemented one dimensional case showed that the accuracy did increase. But, the accuracy did not increase significantly enough and the compression rate was lower. Meaning that the compressed data was larger than with the DCT and that the error was not significantly lower to give more room for further compression. This means that for the common case of compression and image enhancement (removing the blocking effect) the LOT is useful, but in our case where we would like to beat I/O time the extra time spent on computation gave little compression advantage, which resulted in lower I/O speedup. That is main reason we did not implement any further LOT algorithms such as GenLOT or GULLOT. The fact that they work well for compression in general [14] does not mean that they work well for compression when trying to beat I/O time. And this was even the case

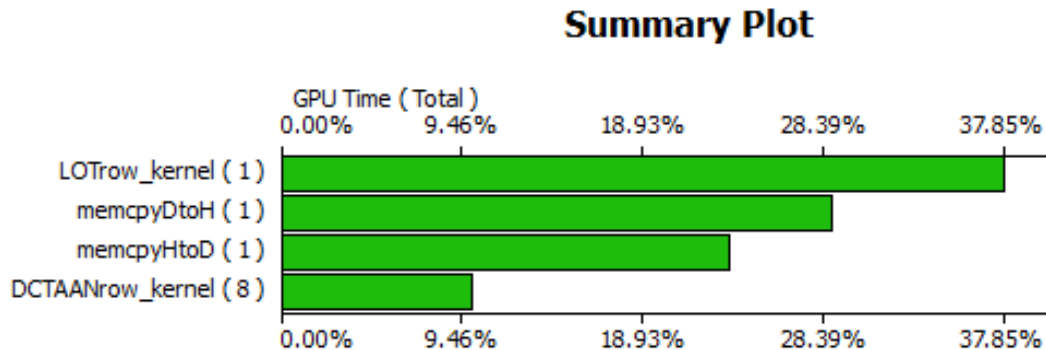


Figure 4.4: CUDA profiler snapshot of the LOT execution

with the computation advantage of the GPU. Even though the LOT did not give more speedup it could be an option for the case where one wants more accuracy.

When it comes to memory optimizations and memory access patterns the fast LOT algorithm is quite similar to the AAN algorithm. They both depend on 16 variable for the current and the previous state. Where the current state of the transform is calculated based upon the previous one. They both have one read and write to system memory to calculate a block of data. And even when parallelized for the CPU and GPU the same optimization strategies are used. Such as running that each thread calculates a block of data of size 8 elements. And the profilers show the similar results as they did for the AAN algorithm in that the pre-fetcher had little misses, and that there are limited cache misses. Another aspect that is similar here is that in both algorithms, since they are separable, one can run the transform in one dimension at a time. This does however inflict some minor faults in memory access in that it is no longer coalesced. This is because reading in any dimension other than the width of the cube will result in jumps in memory. However these jumps are done in a structured and recognizable pattern for the pre-fetcher and hence do not effect the ability to attain the data before it needs to be calculates. Another issue is that since it is only read and written once, one cannot try to reorganize the data because that will also require a read and write making that optimization useless and more time consuming. In Figure 4.4, we show our profiler reslts for the LOT algorithm.

The GPU implementation is performed in a manner such that we have only one transfer of the data to the GPU and back, while the data is on the GPU we run two kernels. The first is the AAN kernel and the other is the LOT kernel.

In the case of several dimensions, we will perform these two kernel in the same manner but once for each dimension. This is a good thing for performance in that the communication time (memory transfer) becomes less evident in the execution time, which is a penalty of using the GPU. The computation time will however rise, and this is positive because then the computation advantage will be more evident compared to the CPU. And the penalty will then be insignificant.

4.5 Optimizing Image Processing Algorithms

As part of this thesis, we not only look at how we are able to accelerate the I/O process with compression, but we are to look at how this effects the whole seismic process including filtering. In our previous work we have looked how the GPU can be used to accelerate the filtering process with the use of convolution, Aqrabi and Elster [4]. In this section, we will be explaining how we implemented two filtering algorithms that have significant characteristics for the compression in the process. The Hough transform is an algorithm that has no dependencies between the sub-problems and no overlap, which makes it ideal if one uses compression. In contrast, the convolution algorithm has overlapping sub-problems that create dependencies and makes the use of compression more complicated. We will address these issues and in the sections to come and how we solve them. The filtering algorithms are both implemented on both the CPU and GPU.

4.5.1 Optimizing 3D Convolution

The implementation of convolution is of a non separable and three dimensional convolution algorithms. This is part of our work from [4]. The major difference here is that we have compressed data being read in to memory before running the filter on it. Since convolution has an overlapping sub-problem, being that borders between sub problems have to be exchanged, creates a dependency between the data read. Now since the compressed data is a 2 GB data set that is compressed. It is hard to read the neighboring border values of a compressed block and decompress them. This leaves three options, one being that we have to decompress three 2 GB blocks read the neighboring values and then move on to filtering. The second option, is that we have smaller blocks of data compressed at a time such that we do not read too much (up to 6GB

like the previous option) at a time. The third option is that we incorporate the neighboring values of the sub cube in the data compressed to being with.

The first option has no chance of giving speedup since one has to essentially read in 6GB to filter 2GB of data. This does not sound like a viable option. The second is limited by two factors, one the fact that the compression rate will become less as we limit the ability to see patterns over the a bigger perspective, and the other is that again one would have to read more than necessary because when expanding to several dimension we would probably have to read in data that we do not need. The third option does not come without its drawbacks, but is the most controlled and least wasteful of the three. Her one should get all the data one needs and gets to perform the filter on the data one needs. There are two major drawbacks here, one being that we will be duplicating data because the borders of one block are the data of another, and the other will be that we will increase the size of the compressed file. It is also wasteful for filtering algorithms that do not have overlapping sub-problems. But, this is the major drawback of working with compressed data is that one can no longer do selective reads and writes without reading whole blocks. For convolution the third option would be the most efficient, but in general the second option is the most practical. In our case we have implemented both, and in we will discuss their effects in the results and discussion chapter to come in Chapter 6.

Now lets discuss the implementations of the actual algorithm. Since it is a three dimensional non separable algorithm it will create issues in memory access and prefetching if not performed correctly. This is because it will have to jump a lot in memory when convoluting in all three dimensions. Another aspect here is that the complexity of the convolution process is not only lead by the amount of data in the to be filtered, but also by the size of the filter as one can see form the formula the size of the filter is equivalent to the amount of computations per pixel. This is however a very parallelizable problem in that calculations form one pixel is independent of the neighboring pixels and therefore they can be parallelized, but the dependency here is between the neighboring sub problems in that we need a padding with the neighboring values. Below we have pseudo code showing the code to generate the filter, which in our case is a gaussian filter, and the convolution process. For more details regarding the convolution process see Gonzales *et al.* [36].

4.5. OPTIMIZING IMAGE PROCESSING ALGORITHMS

```

1 Input data X
2 Result table R
3
4 for each point x dimension
5   for each point y dimension
6     for each point z dimension
7       for each point x dimension of filter
8         for each point y dimension of filter
9           for each point z dimension of filter
10            image value in point += value of filter * value of
11              corresponding token in X
12            add image value to R in coordinate (x,y,z)
13          }
14        }
15      }
16    }

```

Pseudo code for 3D convolution algorithm

On the CPU we have both a single processor and a quad processor version of the code. We use OpenMP to multi-thread and use the computations capabilities of the CPU. Here each thread solves for one pixel by iterating over the size of the filter, which in our case is of size 13^3 . The values for the filter are placed in constant memory which helps because then after these values are buffered they only use one cycle to be read and used and they are not removed from the L1 cache since they are used all the time in each calculation of a pixel. The other values however must be interchanged, and even though the computations are for three dimensions and there are jumps in memory, the profiler results show that there are close to zero percent cache and prefetcher misses. Which means that we were able to memory optimize the algorithm. For the profiler results see Table 4.6.

Table 4.6: Intel Vtune results for 3D Convolution

L2 cache Miss	L1 cache Miss	Branch Mispred.	Prefetch Miss
0.6%	0.7%	0.4%	2.5%

when calculated on the GPU we also focused on utilizing the memory available to best perform the computations. That is why we used constant memory in this scenario as well, which works similarly in that it uses only one cycle to be fetched when need and since it is used all the time it is important that it is available. Here as well as in the other algorithms we have a problem in coalesced memory access, in that when accessing the data in any other dimension other than the width of the cube we will be forced to jump in the memory causing it to be non coalesced, but since each value is accessed once and written once we do not have the use for shared memory, and that

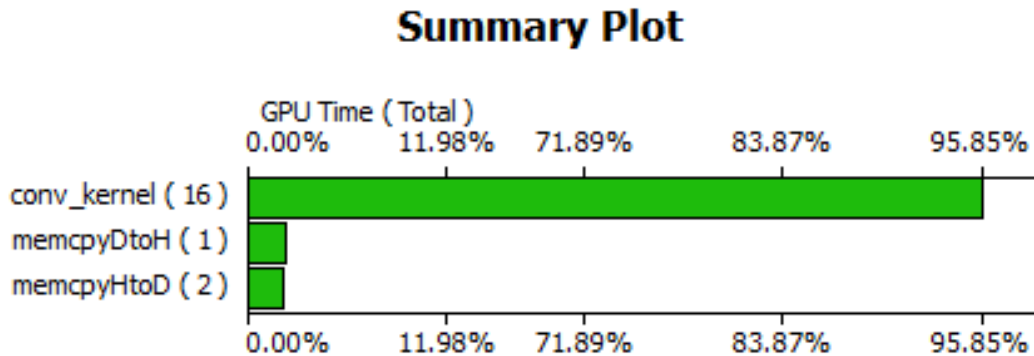


Figure 4.5: NVIDIA CUDA profiler results for 3D convolution

means that we cannot optimize against this and accept that this is part of the nature of this problem and that it is what makes interesting to benchmark and compare to the CPU version since the global memory on the GPU is a limiting factor. We also tuned this algorithm with respect to threads and blocks to adjust occupancy for the Fermi architecture. In contrast to the old CUDA architecture where we were not able to achieve 100% occupancy because of the register use. In the Fermi architecture this was possible because there is support for more register use per multiprocessor. We have mentioned the most crucial elements considered when implementing the algorithm for a more detailed look at the access patterns, see [4]. For Profiler results of the CUDA code see Figure 4.5.

4.5.2 Optimizing Hough Transform

In contrast to convolution, the Hough transform has no overlapping sub-problem and therefore much easier to handle when it comes to reading in compressed data form disk. Considering the options we discussed earlier for the format of the compressed data. The option that will most benefit both convolution and the Hough transform would have to be smaller sized compressed blocks. This is because if we go for the option of adding the neighboring values in the compressed block we would be reading and compressing more data than needed for the filtering. We will be looking in more detail at this aspect in the result and discussion in Chapter 6, and the tests we ran on both options to see which results in most I/O speedup.

The Hough transforms computation complexity is based upon the number of

4.5. OPTIMIZING IMAGE PROCESSING ALGORITHMS

points selected to be computed. The transform starts off by selecting a set of points in the image and then by using the transform one can see which of the points are most aligned. The trouble with the algorithm is that it is prone to conflicted writes to memory. The algorithm reads in coordinates for the set of points in the cartesian and then uses these coordinates to calculate the r and θ values in the Hough domain. The problem is that separate points in the cartesian domain have similar values in the Hough domain, thereby the write conflict. The pseudo code of the algorithm is presented below.

```

1 Input data X \\tuples of points (x,y) coordinates
2 Result table R
3
4 for each point in X
5     for each angel in resulting image
6         calculate rvalue using x, y and angel
7         lookup (r,angel) coordinate in result image R, and add 1 to
           value
8 }
```

Pseudo code for Hough transform 2D algorithm

In our implementation, we are performing the calculations on two dimensional slices such that each thread actually performs all the calculations on the points for the two dimensional image. This is to avoid write conflicts between threads when parallelizing. This will make the algorithm separable and non dependent, resulting in it being very parallel. Looking at our profiler results for the CPU implementation we could see that the algorithm is memory optimized in the sense that there are few cache and prefetcher misses, but an unavoidable nature of this algorithm is the conflicted writes when parallelized. And our solution to limit its effects on the algorithm is to avoid the conflict between threads, which seem to work given our profiler results. The results of the profiler are show in Table 4.7.

Table 4.7: Intel Vtune results for Hough Transform

L2 cache Miss	L1 cache Miss	Branch Mispred.	Prefetch Miss
0.5%	0.3%	0.5%	1.3%

When it comes to the GPU implementation, we have followed much of the same optimization steps as in the CPU parallelized OpenMP version. This implies that each thread solves for a separate two dimensional image and this avoids write conflicts between threads, which results in good parallelity. However there is a timeout limit on the GPU set by the operating system, which means

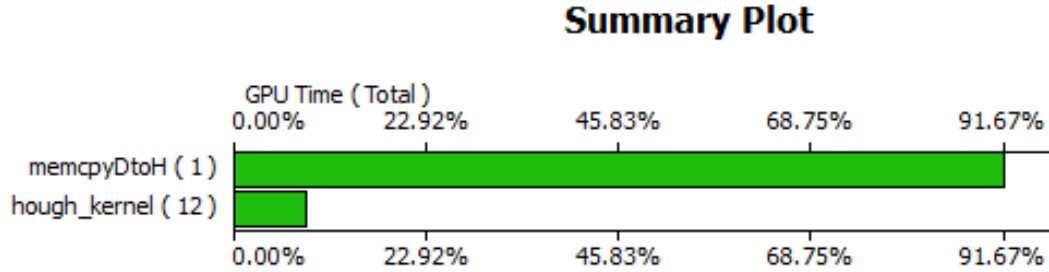


Figure 4.6: CUDA profiler snapshot of the Hough transform execution

that for larger dimensions it could be a problem. This was avoided by turning off this limitation and not going back on the method. This is because if we were to divide the two dimensional image then we would have conflicting writes to worry about. For the Fermi architecture here again the optimization made was to tweek th threads and block sizes to meet the requirements of the occupancy calculator [9]. We were able to achieve a 100% occupancy, which is visible in the results as one can see in later chapters.

4.6 Our AESC Library

Part of this thesis is to create a compression library that uses our findings and the use of the GPU with CUDA. The AESC is a compression library specifically designed for seismic data that use the GPU and is a collection of compression algorithms that we have been discussing in this chapter. The library functions demand a pointer to the input data and output data, the dimensions of the seismic cube, and the amount of threads and blocks that are to be used on the GPU. The amount of threads is important and should be set by the programmer using the library because it is dependent on the architecture one is running on, and we advise the use of the CUDA occupancy calculator [9] to gain most speedup of using the library. An overview of all the functions available in this library are presented in Appendix E, and we have also presented the parameters.

4.6. *OUR AESC LIBRARY*

CHAPTER 5

Predictive Model for Seismic Processing I/O

The aim of this predictive model is to estimate I/O execution time of a certain seismic process. The process is divided into two parts. One, being the I/O process of the seismic data, which we have focused on accelerating in this thesis. The other is the computation process that we looked at accelerating in our previous work [4]. In both cases the execution time will be dependent on the hardware as well as the algorithms used to execute. This implies that the models will reflect this by using variables that can be tested on a certain machine first, to then estimate the rest of the process to be run on that machine.

The normal I/O process on any machine can be expressed as Equation 5.1. Where n is the amount of data to be read/written measured in bytes, and r is the rate at which the disk is able to read/write the data measured in bytes per second. This functions returns the time spent to read or write a certain amount of data.

$$t(n)_{I/O} = \frac{n}{r_{disk}} \quad (5.1)$$

In our case, we are aiming at speeding up the normal I/O process by compressing the data such that the disk read/write rate r is no longer the main bottleneck in the system, but one can introduce compression and decompression algorithms performed on a computation unit to help accelerate the process. This means that the time spent will no longer be dependent on the I/O unit alone, but also the computation unit. This introduces the dependency between compression rates of the algorithms and their efficiency.

The new I/O process can then be modeled in two parts, and conceptually is modeled as in Equation 5.2. This expresses that the I/O time is now the time

5.1. SYNCHRONOUS MODEL

it takes to read or write the compressed amount of data, $t_{compressedIO}$, and the time it takes to compress or decompress the data. This concept reflects the synchronous model. One is also able to model this asynchronously, which we will look at later.

$$t(n)_{SyncIO} = t(n)_{compressedIO} + t(n)_{compress/decompress} \quad (5.2)$$

5.1 Synchronous Model

When using the conceptual model and is expressing it as a function of the original data size n , it will be as in Equation 5.3 for the case of reading from disk, and Equation 5.4 for when writing. Here $n_{compressed}$ is the amount of compressed data in bytes, which is dependent of the algorithm one uses to compress and can also be expressed as $n_{compressed} = n * C$, where C is the compression factor and n is the original data size in bytes. r_{disk} is, as earlier, the rate at which the disk reads or writes seismic data depending on the formula and is expressed in bytes per second. Whereas $r(n)_{decompress}$ is the rate, in bytes per second, n bytes are produced by the decompressing algorithm, and $r(n)_{compress}$ is the the rate at which n bytes are compressed. It is important to note the difference to avoid erroneous estimates.

$$t(n)_{read} = \frac{n_{compressed}}{r_{disk}} + \frac{n}{r_{decompress}} \quad (5.3)$$

$$t(n)_{write} = \frac{n_{compressed}}{r_{disk}} + \frac{n}{r_{compress}} \quad (5.4)$$

The main difference between Equation 5.3 and 5.4 is that when compressing, the original data size is compressed into a compressed file size. While when decompressing the compressed file size is used to reproduce the original file size. The difference is then the amount of reads and writes to memory. When compressing one reads the original file size n than writes $n_{compressed} = n * C$ amount of compressed data, where C is the compression factor dependent on the algorithm used. When decompressing it is the opposite situation, where one writes to memory more than reading. The total time used to compress or decompress is about the same because one performs the same operations in reverse, but usually decompression is considered slower because one has to write more. In our case, we only decompress to memory to perform calculations

and then compress the data again when storing it back to disk making the compression and decompression times quite similar. This is because now that the disk reads and writes the same amount of data, in contrary to standard compression, the major factor is the memory bandwidth, which is usually pretty equal for reads and writes. This is reflected in our results

5.2 Asynchronous Model

When we start using more than one processor and parallelize code, one can do so synchronously by first reading then performing computations on several computation units. Another alternative is to split the reading and computations into different threads and perform them in parallel. This way one would hide a lot of the computation time with I/O time or the other way around depending on which step is the most time consuming. By reading the data asynchronously, one would have to divide it into blocks this way one could read a small part and start performing computations on that, only to read the next part as these computations are being performed in parallel. This way one would hide these computations. In Figure 5.1, we illustrate the scenario of asynchronous reads by splitting the data into 5 blocks.

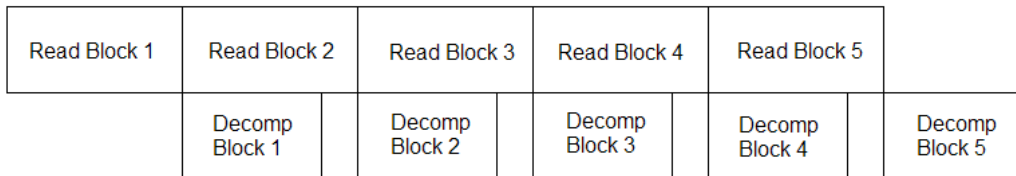


Figure 5.1: Asynchronous I/O Pipeline

Conceptually this would result in the model presented in Equation 5.5, where b is the number of blocks and $\text{Max}(x,y)$ is a function that returns the greater execution time between processes x and y . the other variables have been explained previously in the synchronous model.

$$\begin{aligned}
 t(b, n_b)_{\text{AsyncI/O}} &= t(n_b)_{\text{compI/O}} + \\
 &\quad (b - 1) \text{MAX}(t(n_b)_{\text{compI/O}}, t(n_b)_{\text{comp/decomp}}) + \\
 &\quad t(n_b)_{\text{comp/decomp}}
 \end{aligned} \tag{5.5}$$

A clear advantage here would be to increase the number of blocks to decrease the overhead from the first and last block of the asynchronous process. But, one has to keep in mind that all reads and writes have an overhead when invoked, which means that if one reads small amount of values per read than the read process will take much longer than reading a larger amount of sequential data. This is why we try to read a large amount of sequential data per read and use specific blocking techniques to do so. This was explored in our earlier work with seismic data [4]. Therefore there is a tradeoff between increasing block size and execution speeds.

5.3 Compression Computation and I/O Tradeoffs

Generally the tradeoff in this case is the time taken to compress and the resulting compression rate. Optimally one would like to have a compression algorithm that takes little time to execute and compresses the data well. This way one would reach the best speedup compared to the normal I/O process. But, in reality there is a relation between the time one uses on compressing data contra the execution time, which is that the more time one spends the more one is able to compress. This works ofcourse only to a certain extent where the data isno longer compressable. But, the more time demanding methods compress better as well.

Generally when one is aiming at using compression, the main goal is to compress as much as possible while there is no time limiting factor i.e. one can use a lot of time to achieve the compression. In our case on the other hand, we are limited by the normal I/O time. If the execution time of a compression algorithm exceeds that of normal I/O than it will never be able to achieve speedup, and therefore should use regular I/O. That is why we are aiming at using the computation power of the GPU to run heavier compression algorithms fast enough to gain I/O speedup. This also means that when comparing to faster I/O units such as the SSD disk one has even more limited time and must therefore depend even more on fast compression algorithms, which in return gives limiting compression options.

In the case of performing the compression and I/O operations synchronously one would be limited by the execution time of both since they are performed after one another. This is always the case when performing on a single CPU, everything is performed sequentially. While in the case of parallelism and

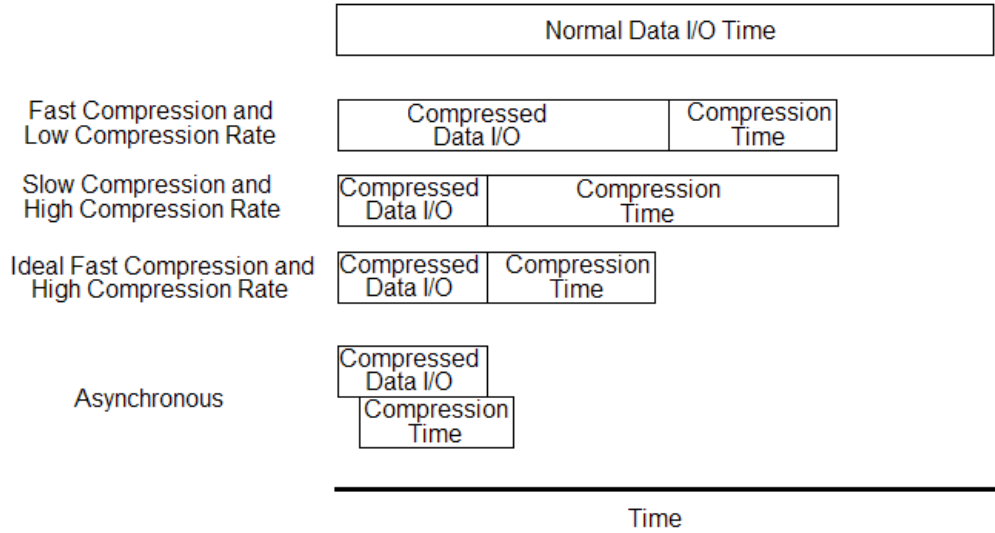


Figure 5.2: Showing advantages in execution time for combinations of fast/slow compression, high and low compression rates and asynchronous compression

using several threads on the CPU, one can perform the computations and I/O asynchronous, and then one would be limited by the execution time of the part that is most time consuming. This way one can hide either the I/O or the computations in by overlapping the two and the one with the highest execution time will be overshadowing the other. This gives more dimensions and possibilities to use more comprehensive compression algorithms even on fast I/O units, up to a certain point of course. The same applies as earlier that if the compression algorithm is more time consuming than the original I/O time then it will not give any speedup. See Figure 5.2 for details.

5.3. COMPRESSION COMPUTATION AND I/O TRADEOFFS

CHAPTER 6

Results, Discussion and Analysis of Benchmarks

In this chapter, we will be presenting and discussing the benchmarked results of our implemented algorithms. We will study the results of individual algorithms and analyze the attempts at using the GPU to accelerate the computations. We will also discuss the results of executing on different hardware and interpret the behavior of the algorithms across different platforms. More importantly we will discuss the results of I/O acceleration efforts.

We will also look at the effects of different choices made and the outcomes they give to the I/O process. Choices such as the devised compression format or choosing between lossless or lossy algorithms. We will analyze the use of synchronous and asynchronous I/O. Finally we will see how all this effects the overall process including filtering the compressed data. This will allow us to evaluate of how effective the use of the GPU can be to the seismic filtering process.

This chapter is divided into 4 sections. In Section 6.1, we will discuss the hardware used to benchmark our results. We will have a look at the data sets used to test and what significance this has for our results in Section 6.2. The final two sections will be benchmarks for the individual algorithms, Section 6.3, and the effect the algorithms have on the seismic filtering process, Section 6.4. Here we will discuss and analyze in detail the behavior of the algorithms and the their influence on the seismic process. We will look at how the process will probably change depending on the platforms and computational hardware the algorithms may potentially run on.

6.1 Hardware & Platforms Used for Testing

Before introducing the benchmarking results it is important to point out the hardware and platforms we have been testing on. We have been testing on two machines that have different hardware specifications. The first machine is one that is put together by us and runs on a windows 7 operating system. A list of the hardware in the first machine is described in Table 6.1. It is worth noting that we have mentioned alternative options on the hardware such as the disk and GPU. This is mainly to underline that we have used the same machine, but have changed to the alternative option such as when we tested the new Fermi architecture of NVIDIA. When it comes to the alternative disk we added another disk to add more variety to our tests.

Table 6.1: Table showing the system components used in machine 1

CPU	Intel Q9958 2.81GHz
RAM	8 GB DDR3 memory
Disk: Alternative 1	Samsung 750 GB 7200 rpm Disk
Disk: Alternative 2	Samsung 500 GB 10000 rpm Disk
GPU1 (used for display)	NVIDIA Geforce 8600
GPU2: Alternative 1	NVIDIA Tesla c1060
GPU2: Alternative 2	NVIDIA Tesla c2050

The second machine is put together by the personnel of the group, which a powerful machine that has some of the newest hardware one can find, but one cannot change the hardware on this machine. This is why we could not have the combination of SSD and Fermi. This machine uses a Linux operating system and is accessed remotely. It is worth noting that the change in operating system can be a source of different results in that the operating system schedule tasks differently. An overview of the hardware of the second machine is presented in Table 6.2.

Table 6.2: Table showing the system components used in machine 2

CPU	Intel i7 2.81GHz
RAM	12 GB DDR3 memory
Disk	Cosair SSD disk 256 GB
GPU	NVIDIA Tesla s1070

6.2 Data Sets for Tests

During our thesis we got permission to use two seismic data sets. One given to us by Schlumberger, which is part of the West-cam oil filed. The other given by Statoil, which is part of the Gullfaks oil filed. The problem with these data sets that they are not that large, and newer sets that are actually larger are confidential. That is why we have combined these sets, and rotate the sets to get a larger cube that has the same statistical distribution such that it is representative of seismic data. We have created several sizes of the sets ranging from 256MB to 32GB. The sets are randomly put together such that we get distributions that vary a little such that we get a representative view of compressing data. The data sets were verified as representable by engineers in Schlumberger [1] [26]. The reason to have different sizes is to see how the algorithms scale. Actual seismic data from test sets are presented Figure 6.1

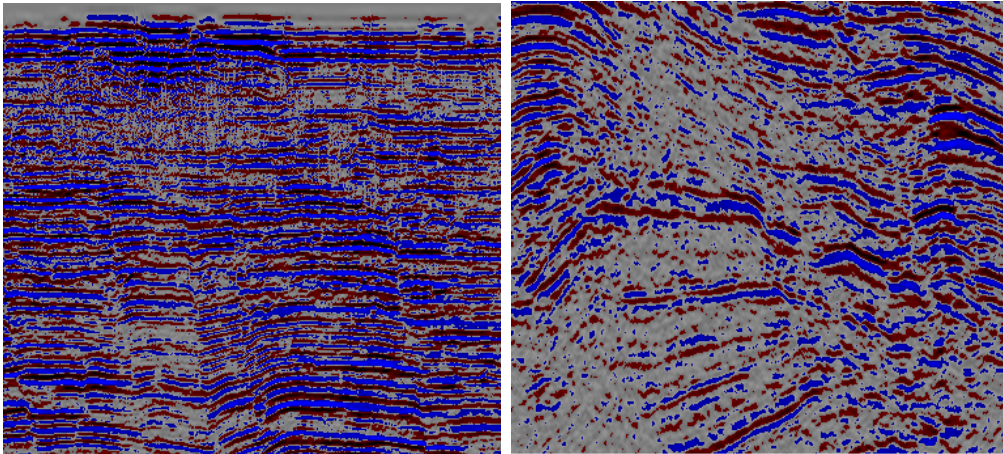


Figure 6.1: Seismic BMP images generated by our framework, based on the Westcam raw data set [1]

In this work we define a larger data sets as a set that does not fit into memory and therefore one is forced to access the disk several times. This is such that the I/O time is representative for the algorithms. The largest memory we have is on the second machine we are testing on, which has 12 GB of memory. But, given the need to duplicate some arrays for result arrays and some extra memory for aspects such as the RLE dictionary and the Gaussian filter for the convolution. That is why even for that machine with larger memory, it will access the disk up to 8 times for the largest data set.

The dimensions of the data sets vary from 400^3 to 2000^3 floats, and the non-

binary dimensions force us to take into account boundary conditions. And makes optimizations more difficult, which makes the tests and scenarios more realistic.

6.3 Compression Algorithms Performance and Visual Results

In this section, we will be presenting, discussing and analyzing our results of benchmarks. The focus here will be to look at the results of individual compression algorithms run on different platforms. This way we can analyze performance on a smaller scale. In contrast to later sections where we discuss the performance and effects the algorithms have on larger processes such as I/O or the seismic filtering process. The layout of section is divided by the algorithms we have implemented, and within those subsections we comment implementations on the platforms of CPU, Quad-CPU and GPU.

6.3.1 Modified RLE Benchmarks

In Figure 6.2, we present the various execution times of the RLE algorithm for the various platforms given the size of the data. Our results show that the quad-core CPU is significantly faster than the GPU at compressing. The results also show that we are able to achieve a 3 times speedup, compared to a single core CPU implementation, for larger data sets. While a 3.5 times speedup for smaller sets.

It is important to note that the compression rate for varying data sets and sizes was significantly similar, at a compression ratio of 1.31. This is significant because it indicates that our analysis of the data and our approach to compressing seismic data with selective compression was successful and robust. The compression rates are similar for all platforms.

Visually the results of this algorithm are no different before or after construction. This is because the algorithm is of the lossless nature. Meaning that no data is lost, which results in no error.

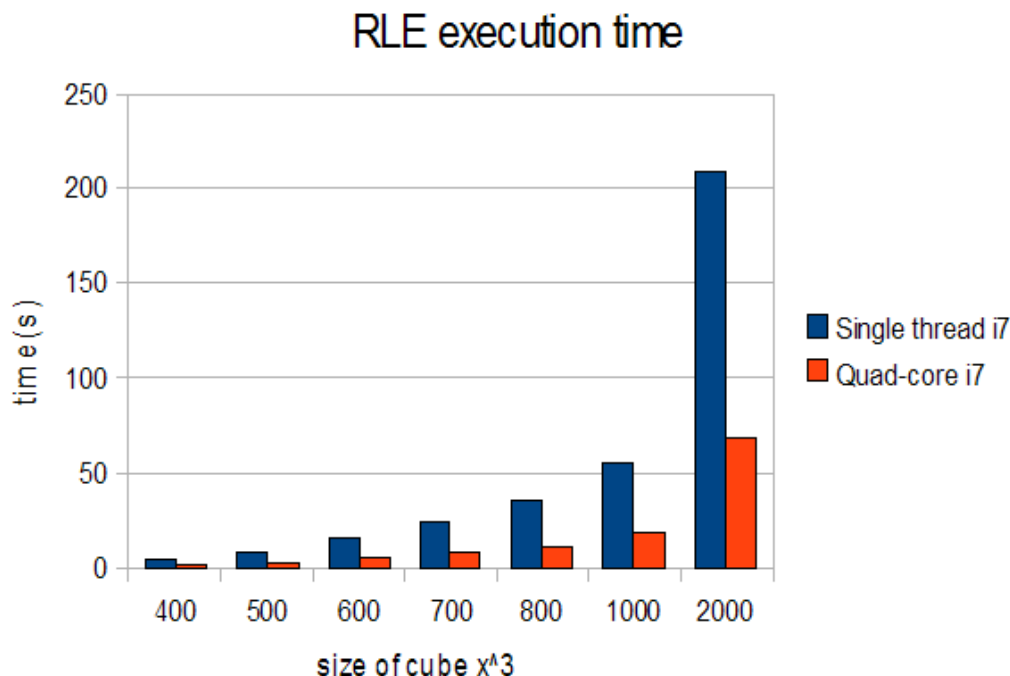


Figure 6.2: Execution time results for RLE algorithm

Why is the CPU faster than the GPU when performing RLE?

There are two major reasons as to why RLE would run faster on a quad core CPU than the GPU. One, is the sequential nature of the algorithm. The other, is the efficiency of the GPU on bit-wise operations. It is documented [23] that the GPU is about four times slower at processing bit-wise operations than the CPU. That is why even with the massively parallel nature of the GPU, a linear speedup will not be achieved compared to the CPU. Another aspect is that each thread would have to wait for other threads to finish before one thread could be responsible to attach the compressed bit together with the other. This will give a dependency in that most threads would be idle before being able to attach their part. One might suggest that using a tree structure to perform the attachment of the sub-problems, but this would not give any speedup. The reason is that as a sub-problem is to be attached all its bits have to be shifted, which is the overhead of attaching. By attaching larger subproblems together, as one does in a tree structure, it would result in the same amount of bit-wise shift operations as the linear method. That is why it would not make a difference.

Why does normal RLE not give the desired I/O speedup results?

As we have mentioned earlier, seismic data is quite noisy. This means that there are many spikes in the data and little consistent values in a row. Run length depends on the data being of the evenly distributed and the same values occur after each other. For example one can replace 50 floats of the same colour in a row with 2 floats expressing the colour and the count, which would give a compression ratio of 25. The fact the data is noisy can actually give a negative compression, which means that the compressed file is actually larger than the original. This was actually the case to begin with, and we optimized by aiming at a selective repeated value to achieve compression.

What effects did the zero optimization have on the compression ratio and computation time?

When optimizing the RLE algorithm we aimed at only compressing for zeroes in a row because after studying the data we found that only zeros actually come in a row. This was done by combining RLE with a dictionary look-up, where each address in the dictionary gave the address of the next zeroes in a row and the compressed data stated the number/count in the same array index. This was very successful in the sense that normal RLE gave a compression ratio of 0.71, which means that the compressed file was 1.42 times larger than the original. While the optimized RLE gave a compression ratio of 1.31, which is about 77% smaller than the original data. This optimization had also a positive effect on the execution time, in that it is faster since it does not have to account for similar number other than zero. This gives an advantage in that execution time is critical for the I/O speedup

Is RLE encoding a viable option for the compression process?

The I/O speedup of the process is at about 1.05 for the larger sets. This means that by using RLE for compression on a quad core CPU then one is able to accelerate the I/O process by 5%. This is not a significant change. The limiting factors here are both the compression ratio and the execution time. The compression ratio is an indicator of the maximum attainable speedup for the I/O process given a compression algorithm. This is because it is an indicator of how much data will be read by the disk, given that one is able to hide most of the computation in an asynchronous fashion, this will be the major portion of the execution time and hence the best indicator for maximum

I/O speedup. Another interesting result is when RLE is run on the SSD disk we get a speed down, which in turn shows that the speedup gained for the HDD disk is limited. That is why one can conclude that it is not a viable option and one might as well use standard disk access.

Are there other lossless algorithms that can give better results?

In the domain of compression, there can be many algorithms found. Most of them are in the category of lossless. But, they do have one thing in common and that is that they are time consuming. The most effective algorithms such as the algorithms that predict and adapt to their data are time consuming in that they need to study the data and then compress it. Algorithms such as LZMA that are used in common compression tools such as the winzip and rar format, have been tested in our thesis. We used the tools to compress our data, but the compression takes up to minutes to perform, and the result is not that much better. The best compression ratio we achieved was with the Winrar tool, which gave a compression ratio of 1.66. Other lossless methods such as the one used by Xie and Qin [50] also resulted in a compression ratio of 1.6. Here they used the method of arithmetic encoding in combination with floating point compression, which we discussed in the data compression chapter (Chapter 3). In other words there are methods that can compress slightly more, but need more computation time that they are not useful in our scenario of beating I/O time.

Why is there a difference in speedup between the smaller and larger data sets?

For the smaller data sets there are fewer shift operations and smaller tables that are to be put together when parallelizing. This results in that for smaller data sizes one will get faster executions when parallelizing, while for larger sets one gets a more stable execution time that reflects better the operations performed. The smaller sets showed a speedup of about 3.5 when run on 4 threads, while the largest dataset resulted in a 3.1 speedup.

6.3.2 Huffman Encoding Benchmarks

The execution times for our Huffman encoding are presented in Figure 6.3 in a graph. Here data sets of various sizes are tested and their execution time

6.3. COMPRESSION ALGORITHMS PERFORMANCE AND VISUAL RESULTS

is recorded. A notable result here is that all the compressed blcks showed a similar tendency in compression rate, with a compression ratio of 1.4. this indicates that the maximum achievable speedup is 1.4, which already looks like is not a viable option in the same way as RLE.

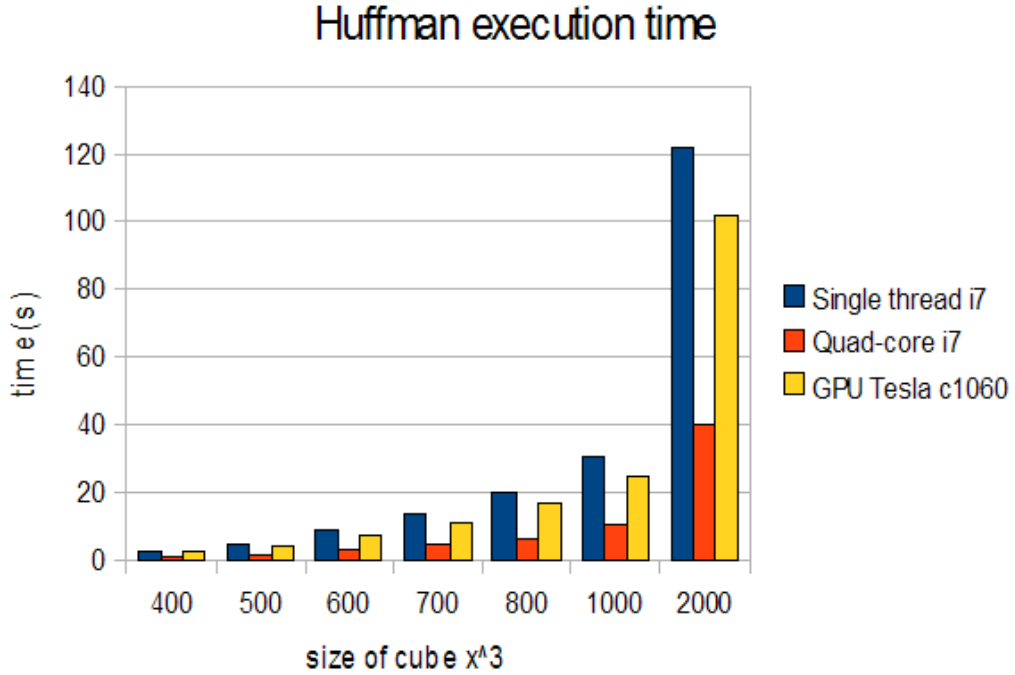


Figure 6.3: Execution time results for Huffman encoding algorithm

How does our CUDA implementation relate to other Huffman implementations?

We achieved similar speedups as others who have performed Huffman encoding on the GPU (such as [20]), but we differ in that we have an approach that better utilizes the memory hierarchy of the CUDA architecture. We use constant memory to utilize memory access to the lookup table because it is buffered, which means that our computations are more optimized. This shows that the issue with the huffman algorithm is not the computational complexity, but rather its sequential nature. We were able to achieve a 1.2 speedup for the huffman algorithm on the GPU in contrast to the CPU, but in this case the GPU did not give an advantage good enough to beat general I/O time. This makes this algorithm not a viable option for I/O speedup, but has the advantage

of being better at lossless compression than RLE for non transformed seismic data.

What effects do our optimizations have on the algorithm?

The most effective optimization we implemented is the predefined huffman tree, which is generated at the beginning of the program. This optimization resulted in dropping to have to read the entire data set twice from disk. Because in normal huffman algorithm the data set is read once to generate a tree and then again to compress. This optimization was able to cut the execution time in half by simply predefining the data to be read in and predefining a huffman tree. This optimization is only possible if one knows the data to be compressed beforehand, which we do in this case. When testing on the several datasets we have all returned a similar compression rate, which means that our assumptions about the nature of seismic data is correct.

Another optimization that influenced the execution time drastically is instead of building a tree we have created a lookup table for all combinations of the 256 values a byte can have, and this is done with a base of replacing to bits at a time. This will add a bit of a delay to begin with to generate the table, but accelerates the execution in a way that we skip many bit-wise operations to access the data to bits at a time. This was the optimization made it possible to beat I/O time at all, because we now traverse the data a byte at a time instead of to bits at a time, resulting in a quarter of the operations. In other words the algorithm is 4 times faster because of this.

What makes the algorithm less efficient on the GPU?

The GPU is well known for its computational advantage for algorithms that can be parallelized. But, it does come with its limitations. The GPU is made to operate on floating point data and therefore when operating with bit-wise operations, which are performed on integers, it is not effective enough as the CPU. This is of course a problem for compression in that we try to replace existing bits to express the same information with fewer bits, therefore we must use bit-wise operations. Having said that, one would probably get a speedup on the GPU even when performing only bit-wise operations, if the problem is separable and parallelizable.

In compression one has to always put together the separate compressed blocks to a compressed array, and the more pieces one has to put together the more

troublesome it will get. To utilize the computational advantage of the GPU we would have to use many threads to hide computations, but having each thread wait for the other to put together the data is too time consuming, and as we have discussed earlier there is no way around it. This is even a trend that we see when scaling on the CPU. That we do get a speedup up, but far from linear speedup. This is because there is a sequential overhead in putting the data together. Proceeding to use the GPU might cause that overhead to grow because then there are several more communications that need to take place between threads causing an even greater overhead. This sequential nature of the algorithm is the biggest cause of lacking in computational advantage on the GPU.

How does the algorithm scale?

The results clearly show that the algorithm scales well in computational throughput and that the compression ratio is similar for the various test sets. This is a good sign that shows that our algorithm is stable in that it is able to compress the same amount of data every time. It also verifies our assumptions about the seismic data distributions. It also shows that the time spent on creating the look-up table for the algorithm is insignificant and gives such a great speedup for the algorithm as a whole. Other aspects of its scaling is how it scale cross platform. The results indicate that the algorithm functions better on the CPU than the GPU, which reasons are discussed earlier.

6.3.3 Naive DCT Benchmarks

The naive DCT in our work is implemented in two dimensions, this is mainly because the results after the two dimensional implementation revealed the need for a faster algorithm, but that the compression rates were very useful. That is why we approached the AAN algorithm and implemented in in three dimensions. The DCT results however showed how effective transform encoding is compared to lossless encoding without having to lose much data. To measure the loss of the data we used a common formula in image processing that is the root mean square error. This is a good indicator if there are spikes in the resulting transformation because by squaring the difference a spike would gain more error. One could however argue that one could simply use the mean error in this case since the error should not be more than two decimal places and is the average error is less than that we have a quite reliable error margin. The formula we used to calculate the error is as follows:

$$rMSE = \sqrt{\frac{\sum_{x=0}^n (I(x) - O(x))^2}{n}} \quad (6.1)$$

The one dimensional DCT gave a mean error of 0.0024 while the two dimensional DCT gave a mean error of 0.0032. This error is however emphasized in the case of using the rMSE, which resulted in an error of 43 and 81 respectively for the one and two dimensional case. What this means is that there are few errors since the average error is almost similar, but the few errors are spikes of change that are even visible on the visual results. However for the visual results we can see that the traits of the data have not changes significantly, and the image is quite similar. After consulting engineers at Schlumberger [1][26], we confirmed that these results are acceptable, and that given this loss of data close to no influence is made on further analysis. This shows that we are within acceptable error rates. The before and after images of the seismic data with 10x zoom shows the effects that we mentioned of spike data and few errors in Figure 6.4.

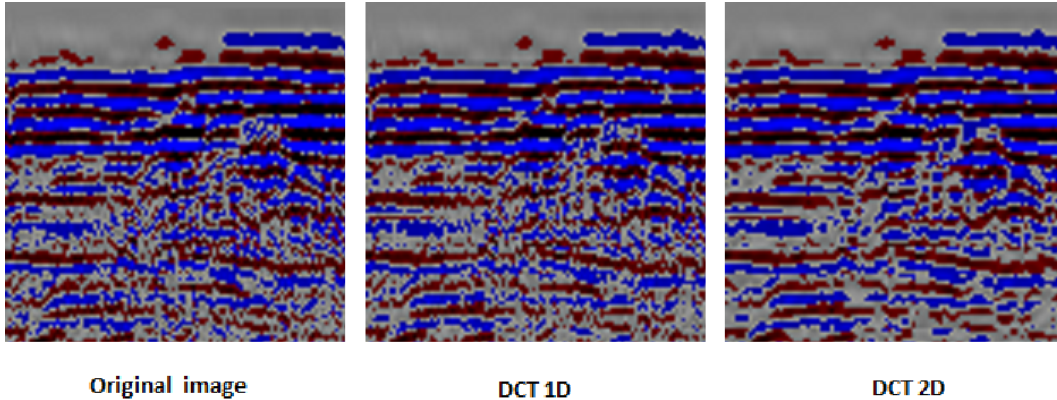


Figure 6.4: Before and after images of the transform encoding process using the naive DCT algorithm, generated by our framework

The execution time results of the naive DCT for both one and two dimensions are shown in Figures 6.5 and 6.6. Here one can clearly see that the GPU is a great asset in the transform encoding process and the cooperation between it and the CPU have given great results. The compression rates of the DCT are 1.55 and 3.01 respectively for the one and two dimensional case. Which shows that the maximum speedup one is able to achieve is much larger than that of the lossless algorithms, which opens up a potential to actually beat I/O time significantly even on newer disk platforms.

6.3. COMPRESSION ALGORITHMS PERFORMANCE AND VISUAL RESULTS

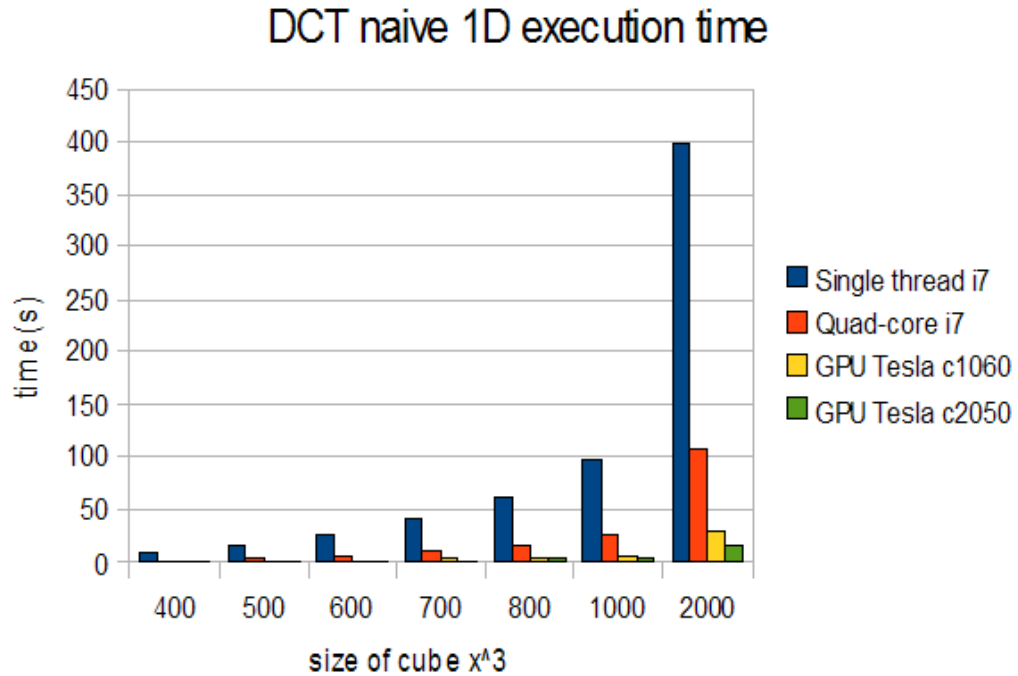


Figure 6.5: Execution time results for naive DCT 1D algorithm

How does the quantization step help in compression?

One of the significant aspects of the DCT is that it is able to compress the energy in the frequency domain better than the fourier transform. In the quantization step of the process a low pass filter is used to remove noise from the seismic images. By doing so one gets better compression because only some of the data is valuable now. In our quantization step we set values that are close to zero to actual zero. This is usually the last 4 of the 8 elements evaluated in the transform. This can be used as an advantage in the transforming algorithm in that not all 8 values need to be evaluated one could only evaluate 4 of them since the last 4 would be set to zero anyways. However by setting the value to zero of values that are not even close will result in a large loss of data. The reason as to why this gives such good compression is because when setting values that come directly after each other to zero and we run our modified RLE on them. We get to compress a four 32 bit float as a 1 bit tag and a one floating point counter, which gives greater compression.

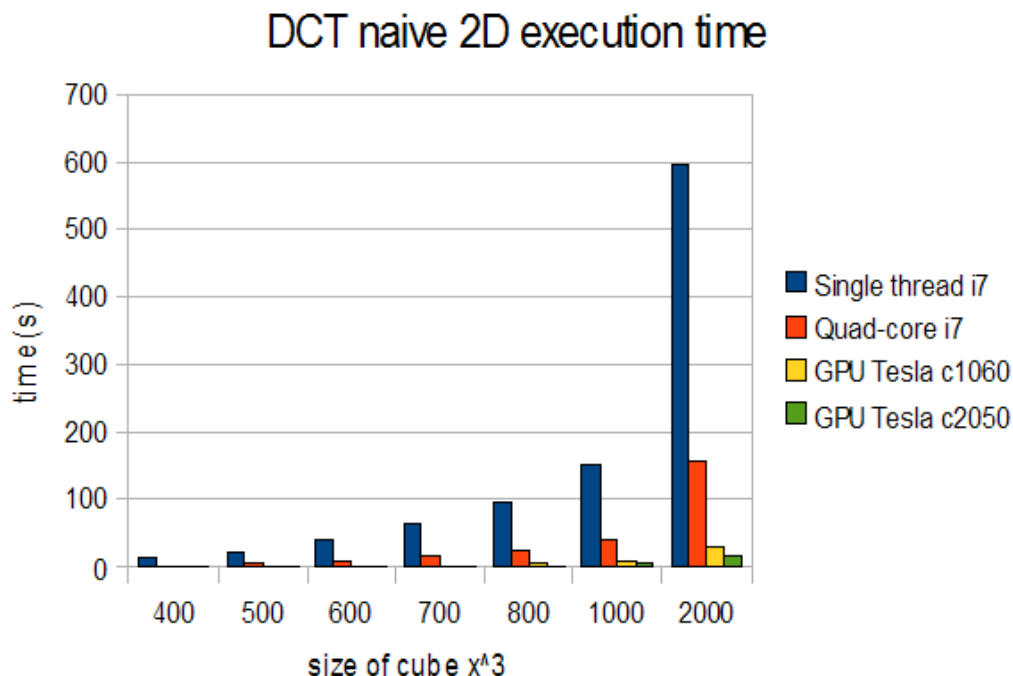


Figure 6.6: Execution time results for naive DCT 2D algorithm

Why does the compression ratio vary for compression in different dimensions?

By transforming the data in several dimensions more of the transform would be zeroed out and this gives more room for the RLE algorithm to compress the data. In the one dimensional case if all the data was non zero we would be able to express 8 floats with 5 floats and 8 bits. This results in a compression ratio of 1.52, which is almost the case in our data as well. For the 2 dimensional case however we would have for each 8×8 (64 floats) block only a 4×4 (16 floats) of significant data, the rest 48 floats are then zeros that can be easily compressed. The placing of this data would give the RLE algorithm the chance to compress twice as much data. This is because the first 4 rows of transformed data would be similar to the one dimensional scenario, which comes from the fact that a new block starts every 8 elements. But, the next 4 rows would only be zeroes causing them to be expressed with one float. This means that one can express double as much data with same amount of compression giving room for doubling the compression rate, which again agrees with our results. And this is how expanding to new dimensions gives the advantage of compression, because several rows or even 2D slices (in the 3D case) are then zeroed out

creating more room for compression.

How does the algorithm scale on the different platforms?

The naive DCT gives good compression ratios, which give room for more I/O speedup, but is very computationally expensive. This is why we see the great scaling of using the GPU for the transformation. In this thesis, we do not measure the speedup as a speedup of performing only the transform from the CPU to the GPU, which works very well, but rather the whole compression process when performed on the CPU in contrast to being performed by cooperation between the CPU and GPU. This cooperation gives a great advantage and a computational speedup of about 13 for the one dimensional case and a computational speedup of 24 for the two dimensional case. This is primarily because the GPU is so efficient in performing the transforms, and when it becomes more computationally expensive the GPU excels, as we see in the two dimensional case. But, the fact that the computation time is accelerated for the compression process does not mean that I/O speedup, which is the goal has gained any significant advantage.

Why not perform the naive DCT in three dimensions?

The reason for this is that even though the computations can be performed faster using the GPU we still do not get any I/O speedup. The fact is that there are way too many computations involved and if it does not give any I/O speedup, given our limited time, we should be looking at methods that can. The most crucial element here is that the DCT proved that larger compression is allowed, which we already know from previous work done on seismic images, but we needed to look at faster DCT algorithms to make it work for our cause, namely I/O acceleration. The fact that there are faster algorithms for the DCT is why it is widely used in codecs for film and audio, without the fast DCT we would not be able to stream the data in the way we do today in video formats such as MPEG. To sum up, even though we did not perform the naive DCT in three dimensions, we did so for the AAN/fast DCT algorithm and the naive implementation identifying the possibilities.

How does this method compare to the lossless methods?

Clearly, this method gives better compression ratios than that of the lossless algorithms with little loss. But, given that there is this much potential in the compression ratio in terms of maximum I/O speedup one is able to reach. The lossless algorithms are better percentage-wise than the naive DCT algorithm, even when run on the GPU, and this comes from the computational complexity of the transform. To clarify, the compression ratio of the Huffman encoding algorithm is 1.4 and this gives a 1.2 I/O speedup compared to normal I/O time. This means that about 86% of the potential is met with the algorithm, after optimization of course. While for the two dimensional DCT which gave the best I/O speedup, the maximum I/O speedup is at 3.01, which is the compression ratio. On the other hand the I/O speedup is at 2.4, which is about 77 %. This clearly shows the point we made earlier that the DCT has the potential, but given a faster algorithm we could better use the advantage this method presents. One can also argue that overall the transform coding method with the naive DCT has more speedup with little loss, and therefore is better, but it is good to look beyond that as well.

Why use RLE and not Huffman in the encoding step of the transform coding?

The main reason for this is because RLE with our optimization is more efficient at compressing the transform. Without our optimization it would be the other way around. But, since we in the quantization step as mentioned earlier zero out the higher frequencies, more compression is achieved. Besides the Huffman algorithm would not be as efficient as it is now, because we would have to take two runs through the data to create a Huffman tree. This is mainly because we do not know before hand how the data would look like and therefore must perform more work to compress it. This would make the Huffman transform less efficient when it comes to execution time as well.

How does the Fermi architecture effect the DCT algorithm?

The Fermi architecture has the advantage of having a cached structure, more cores and faster memory, and therefore it is safe to assume a speedup when using it. But, what sort of speedup can be expected and why did we get the speedup that we did by simply changing graphical processors. The NVIDIA FERMI tesla card c2050 has double the cores, which means that it has double

the computational power, but it only has 40% faster memory access since it uses DDR5 memory. This means that the algorithms run on this platform that are not dependent on caching, which is the case with transforms, a maximum of 2 times speedup is the limit. This is of course a great improvement, but one should not be surprised if this is not always achieved. This is because the algorithm has a lot of memory access where every access is to a different address, such as it is in transforms, the time spent on memory access can only be speeded up to 1.4 times. This means that depending on the distribution between memory access and computation one can get a speedup between 1.4 and 2. This is of course given that one optimizes for the platform. For algorithms that benefit from caching the scenario differs significantly, and even greater speedups can be achieved. But, in our case a speedup of 1.8 is achieved, and this is because of the memory bandwidth holding back computational power. Having said that a speedup of this magnitude is very significant and compared to the CPU a speedup of 43 is achieved for the two dimensional case in contrary to the older architecture and the 24 times speedup.

6.3.4 AAN Implementation Benchmarks

The results of execution wall clock times of the AAN algorithm used with our RLE algorithm for transform encoding are shown in Figures 6.7, 6.8 and 6.9. Since the AAN algorithm is very efficient we have performed it up to 3 dimensions, where the GPU implementation is a cooperation between the GPU and the CPU. Here the GPU performs the transformation and quantization steps and the CPU is encoding using the RLE algorithm. One can clearly see that there is an advantage of using the GPU over the CPU, but what is interesting is how the algorithm actually scales across the dimensions. For the one and two dimensional case the computations are so few compared, that compared to the time spent on transfers the CPU is actually able to outperform the GPU. It is not before the three dimensional case where the computations become so heavy that the GPU excels.

As mentioned earlier, some of the interesting results are that the algorithm is computationally fast that even the CPU is able to catch up to the GPU, given the transfer penalty the GPU suffers on the PCI express bus. In Figures 6.10, 6.11 and 6.12, we show how the balance between computation and communication when performing the AAN algorithm. One tends to forget about one of the major limitations of the GPU when working on larger data sets, namely the memory transfer from host to device. This is very time consuming, and when the computations are less dominant we clearly see its effects.

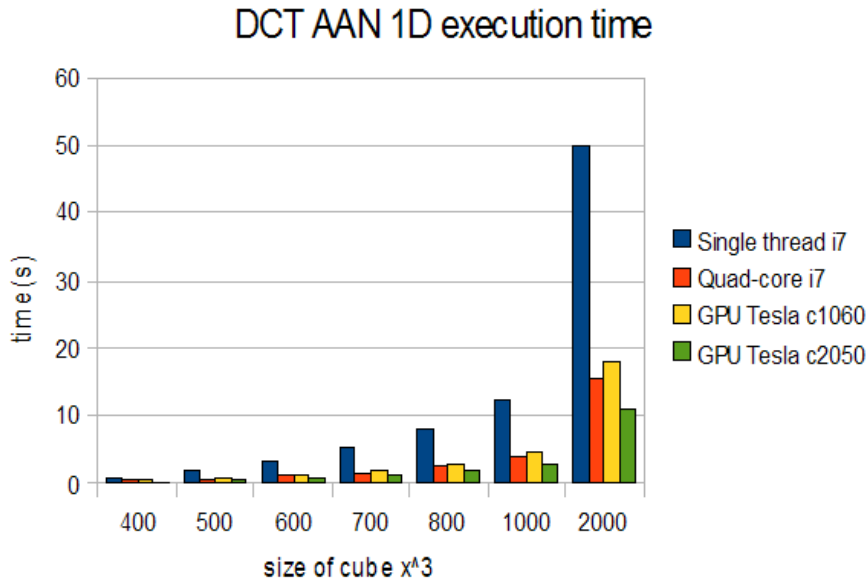


Figure 6.7: Execution time results for AAN DCT 1D algorithm

Another aspect worth noting is the error and visual aspect of the 3D transform, which we performed now that it is useful for the I/O speedup purpose. In Figure 6.13, we show the visual results of all the compressions performed and reconstructed using the AAN algorithms in all three dimensions. The mean error is still quite low and is at 0.0046, but more spikes are present here than before, which is why the rMSE is at 81.2, but the results are still within the reasonable visual perspective as mentioned earlier and this only emphasizes our point on that the image only lacks some minor parts of the image and only some minor spike are missing, which is why the average is not effected. But the values that do change, change dramatically, which pushes the rMSE up. The images illustrate the point better.

6.3. COMPRESSION ALGORITHMS PERFORMANCE AND VISUAL RESULTS

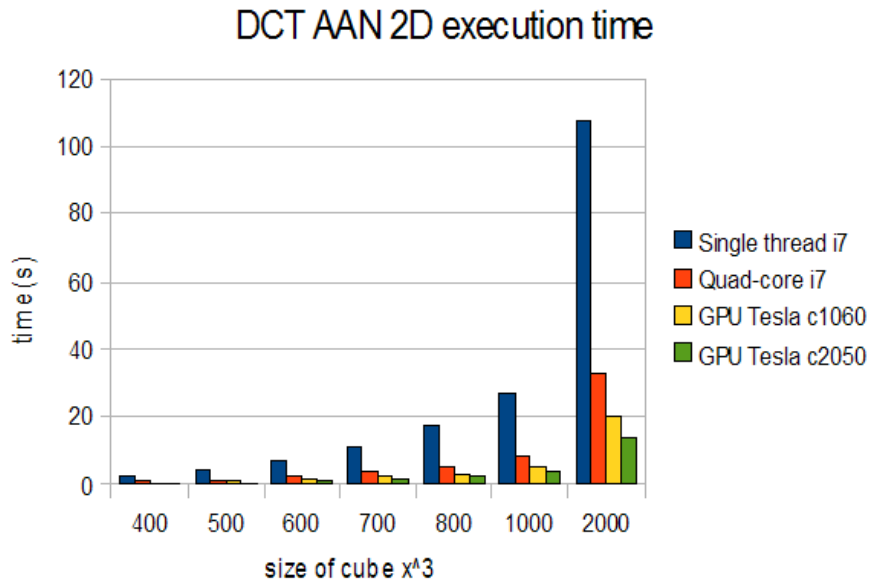


Figure 6.8: Execution time results for AAN DCT 2D algorithm

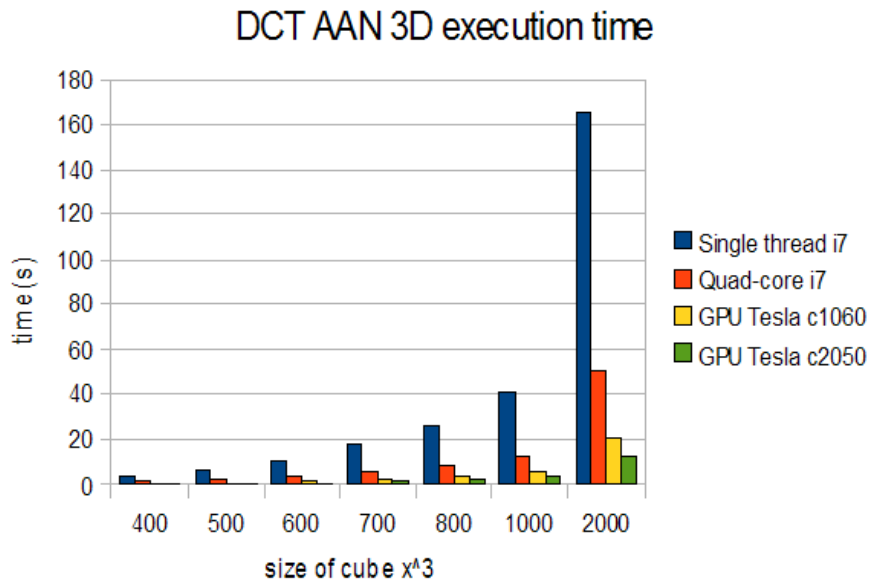


Figure 6.9: Execution time results for AAN DCT 3D algorithm

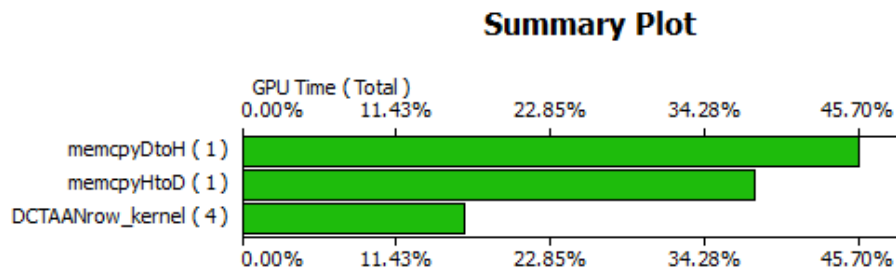


Figure 6.10: CUDA profiler snapshot of the DCT AAN 1D execution

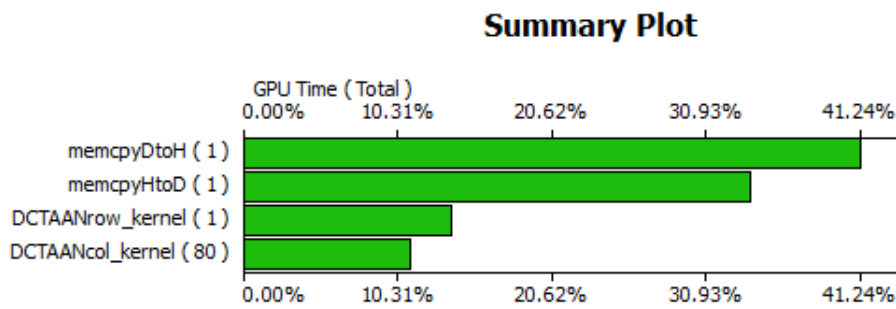


Figure 6.11: CUDA profiler snapshot of the DCT AAN 2D execution

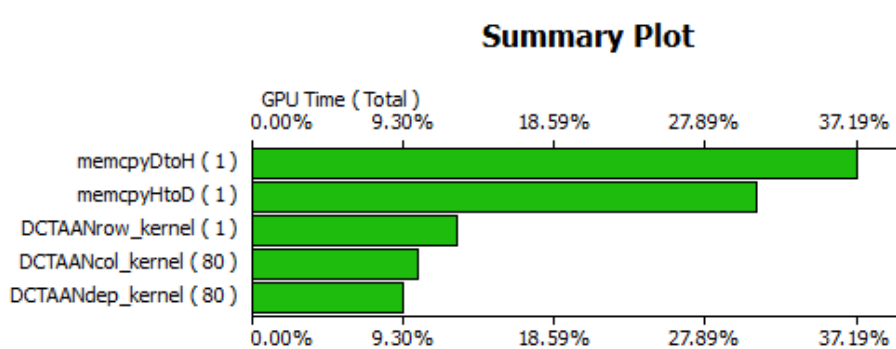


Figure 6.12: CUDA profiler snapshot of the DCT AAN 3D execution

6.3. COMPRESSION ALGORITHMS PERFORMANCE AND VISUAL RESULTS

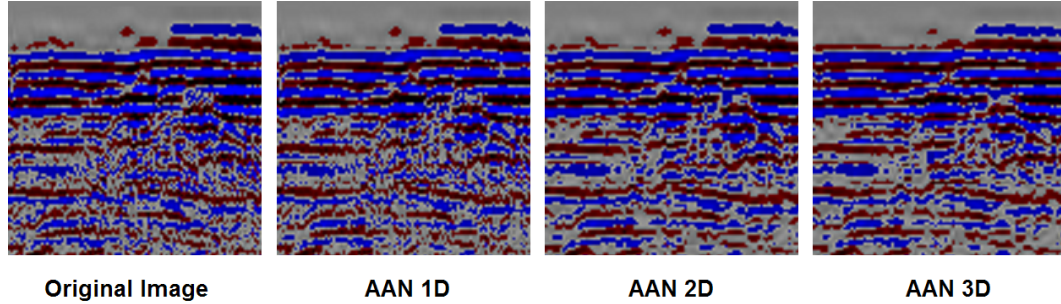


Figure 6.13: Before and after images of the transform encoding process using the AAN DCT algorithm in several dimensions, generated by our framework

How does the AAN algorithm differ in computation complexity from the naive DCT?

The naive method of computing the DCT has an asymptotic running time of $\Theta(n^2)$ for the one dimensional case and $\Theta(n^2m^2)$ for the two dimensional case, where each letter represents the size of the block to be calculated in each dimension. Since one is calculating n elements using n traversal per element, this gives $n^2m^2 + nm$ multiplications including the scaling multiplication performed in the end. While there are n additions for each element calculated and it takes $(nm-1)$ operations to add those number there will be $nm(nm-1)$ additions. That means that computing a block of size 8×8 using the naive method requires 8192 floating point operations when not using the separability of the DCT. On the other hand we have the AAN algorithm that is built upon the FFT which has a complexity of $\Theta(n \lg(n))$, and scales similarly. Given the flow graph of the AAN algorithm we see that there are 29 additions performed and 13 multiplications per 8 elements in a single block in the one dimensional case, which gives 336 floating point operations. While in the two dimensional case we will get a 672 floating point operations per cube and one can clearly see how much more effective this method is. Our results in run time also reflect this, but we must take into account memory bandwidth and other such variables to fully reflect the situation since floating point operations only reflect computations performed. But comparing our analysis we see that the AAN algorithm is about 12 times faster in theory, and in practice it is about 9 times faster on a single thread CPU. This is mainly because of the memory bandwidth which is a similar communication cost for both.

Why does the AAN algorithm scale better?

The AAN algorithm as mentioned earlier is not as computationally expensive as the naive DCT, and therefore when performed on the GPU the transfer limitation is more present. The GPU has a major drawback in that it needs the data to be transferred to it from the system memory to its global memory before computations are performed. This is not an issue for computationally intensive algorithms since the transfer time is then masked by the computation time. But, for larger data sets this is a major issue in that the transfer time is about half the time spent on the GPU, as we showed earlier in Figures 6.10, 6.11 and 6.12. This is a problem because here again the bus bandwidth is an issue for further optimization.

How does the possibility to expand to three dimensions effect the compression ratio and error?

The compression rate achieved with the three dimensional DCT are greater than those before it and give a breakthrough in the purpose of beating I/O time. The compression rate is at 6.58 for larger sets. Here again the compression rate does not vary drastically, which means that its stable for compressing various sizes and examples of seismic data. And with the efficiency of the AAN algorithm on the GPU for the three dimensional case, a I/O speedup of 6.2 is achieved on a fast HDD disk (70MB/s transfer rate).

How does the Fermi architecture effect the AAN algorithm?

When running the CUDA implementation on the NVIDIA Tesla Fermi card c2050, we could see little change in the speedup. This is mainly because the FERMI architecture accelerates the computations done on the card, but the transfer time, which is the bottleneck for this algorithm is the same for most graphical accelerators. The exception are those that are integrated and share the system memory with the CPU. Since the transfer bandwidth is the bottleneck and about half the time is spent on transfer, the speedup attained by switching graphic cards in this case was limited to 1.3. this a minor speedup given the resource difference in the two graphic cards, the tesla c1060 and c2050. This is return gives a speedup of 10 when compared to a single thread CPU implementation on an Intel i7 machine. Some of the reasons this is so is because the algorithm used here is less computationally intensive than the naive DCT and so lesser speedup is gained between computation platforms.

But, the important matter here is that the I/O speedup gained with this complexity is substantial. This proves that the graphical processor does not excel in all cases, but it does give room to offload the CPU and create possibilities to accelerate the I/O process.

6.3.5 LOT Implementation Benchmarks

Results of the wall clock execution times for our LOT algorithm are presented in Figure 6.14. Some of the first notable results are that the compression rate has not changed much, rather it has increased slightly because of the increase in size of the compressed LOT, because of the addition of the zero blocks. Another aspect worth noting is that the execution time has doubled for the single core CPU, but only effected slightly the GPU implementation which is good because this will give more speedup on the GPU and it means that the extra computations are being hidden well with the parallelization. However, given that the transformation did not give more compression, there is no I/O speedup to gain from this method. Rather it is a method that will achieve less I/O speedup, which is a contradiction to our goal. Making this approach not that viable. The only reason this method would then be viable is if the effects on the error would be greater such that we can compress more given the same error as the DCT.

Visually the LOT does look better than the DCT compressed blocks as one can see in Figure 6.15. It helps greatly against the blocking effects. However the mean error produced is slightly larger for the one dimensional case it went from 0.0024 to 0.0031, but on the other hand the root mean square error (rMSE) has dropped. This indicates that there are several values with errors, which results in the mean error rising, but the errors are less spikey, which leads to a lower values when squared.

Why use the LOT for compression?

The LOT is a solution for the DCT transform in that it uses overlapping basis to solve the visual blocking issue. The visual results show that this has worked quite well in that it has lowered blocking effect, but the main reason as to why the LOT is used on seismic data is because it is able to decrease error such that it gives room for more compression. We see that it does reduce error, but not significantly, and the mean error is actually higher. Visually the error seems to be less since the blocking has vanished, but is the replaced

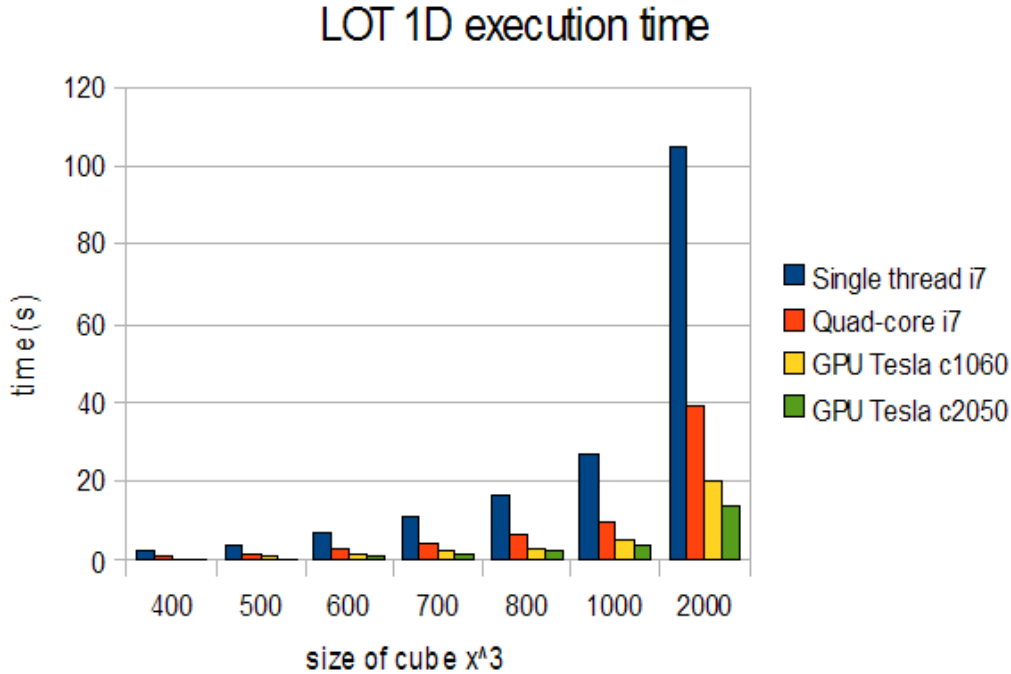


Figure 6.14: Execution time results for LOT algorithm

with minor spikes. The compression achieved with the LOT was not better, it was actually more or less the same. Therefore the motivation was to use the reduction of error to increase compression, but the error was not reduced significantly and the compression is similar. Knowing the the LOT uses more execution time because of the computational complexity, we are set back in reaching I/O speedup and therefore this is not a viable option for our goal.

Why should we not explore further GenLOT on the basis of our LOT results?

Given what we discussed in the precious question. We now know that there are valid motivations in pursuing more advanced lapped orthogonal transforms such as the GenLOT, but given our results that show that we are sacrificing a lot of computation time and get little return in compression. This strategy will not work for our purpose of beating I/O time even through it might work generally for better compression, which is also shown by Duval et.al [14] and given our results is valid. But, nevertheless we cannot pursue this because we are limited by the normal I/O time and the GenLOT will add to many

6.3. COMPRESSION ALGORITHMS PERFORMANCE AND VISUAL RESULTS

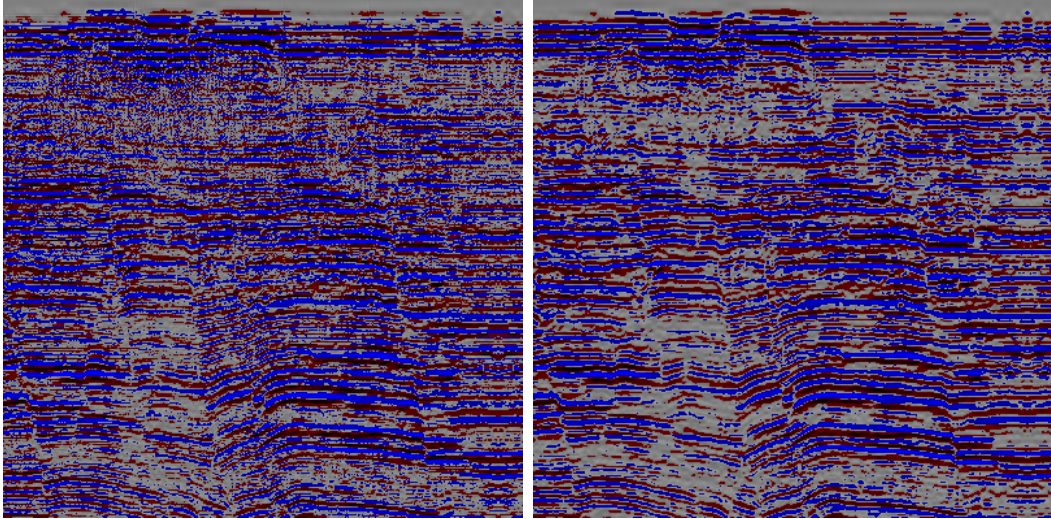


Figure 6.15: Seismic data after transform coding LOT and AAN in 1D, generated by our framework

computation with little return for compression. Hence the decision not to use it based on our results.

How does the LOT effect the error and the compression rate?

The mean error for the one dimensional LOT is 0.0031, which is larger than that of the DCT. But, the root mean square error (rMSE) dropped from 22.3 to 21.5. This indicates that there are more values with errors, which results in the mean error rising, but the errors are of less magnitude, which leads to a lower values when squared and reduce the rMSE. This is emphasized in the visuals as well. Given these results one can see that the error is not reduced enough for us to compress more, and to achieve this reduction demanded such computational sacrifice that we were not able to reach the same I/O speedup as we did with the DCT. Making the AAN algorithm the better option in this case. The compression rate was similar for the LOT as the DCT because they have the same quantization step and therefore compress similarly. The LOT has its advantages, but does not serve our goal.

How does the Fermi architecture effect the LOT algorithm?

The LOT algorithm being quite similar to the AAN is effected in the same way. The results show a greater speedup in this case because of the higher

computation complexity, which in turn reduces the effects of the transfer of data to the GPU. A speedup of 8.1 is achieved using the Fermi architecture on the one dimensional case contra the 5.3 achieved on the NVIDIA tesla c1060. To put this into perspective the AAN algorithm gave a speedup of 2.8 for the one dimensional case. This proves how more computationally intensive this algorithm actually is compared to the AAN algorithm. No further dimensions were explored for the LOT when it was discovered not to be useful for the goal of this thesis.

How does the LOT compare to the AAN computationally?

In the previous section about the AAN algorithm, we showed how the AAN algorithm demands 672 floating point operations per 8x8 block that is to be calculated. We chose to analyse the two dimensional case since the naïve method of the DCT was only developed for the two dimensions given that it showed to be limiting and the research moved forward to a faster algorithm, mainly the AAN. To build further on this comparison we will analyze the computational needs for a 8x8 LOT block that is to be transformed. Knowing that the LOT is built upon the DCT, we have used the AAN DCT as a foundation for the fast LOT, which means that before starting we would need 672 floating point operations to get to the starting point of the algorithm. Following the flow graph we then note that there are 14 multiplications, 16 additions and 6 multiply-adds performed as well. Since these are performed on 8 variables in 2 dimensions then the complexity of a two dimensional block of size 8 is $672 + 2 * 8 * 14 + 2 * 8 * 16 + 2 * 8 * 6 = 1248$ floating point operations, which is almost double the arithmetic complexity. This is well reflected in the results we have obtained. And the GenLOT for each step will only add to this complexity making less useful for us.

6.4 Image Processing Algorithms Performance and Visual Results

In this section, we will be presenting, discussing and analyzing our results of benchmarks. The focus here will be to look at the results of individual image processing and filtering algorithms run on different platforms. This way we can analyze performance on a smaller scale. In contrast to later sections where we discuss the performance and effects the algorithms have on larger

processes such as I/O or the seismic filtering process. The layout of section is divided by the algorithms we have implemented, and within those subsections we comment implementations on the platforms of CPU, Quad-CPU and GPU.

6.4.1 3D Convolution Benchmarks

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 explored are 5, 9 and 13. One can clearly see the effects of using a bigger filter and the increase in computation time. Our findings generally focus on a filter of size 13 because of the computational complexity and it being exemplary representative. Execution time results for convolution are presented in Figure 6.16.

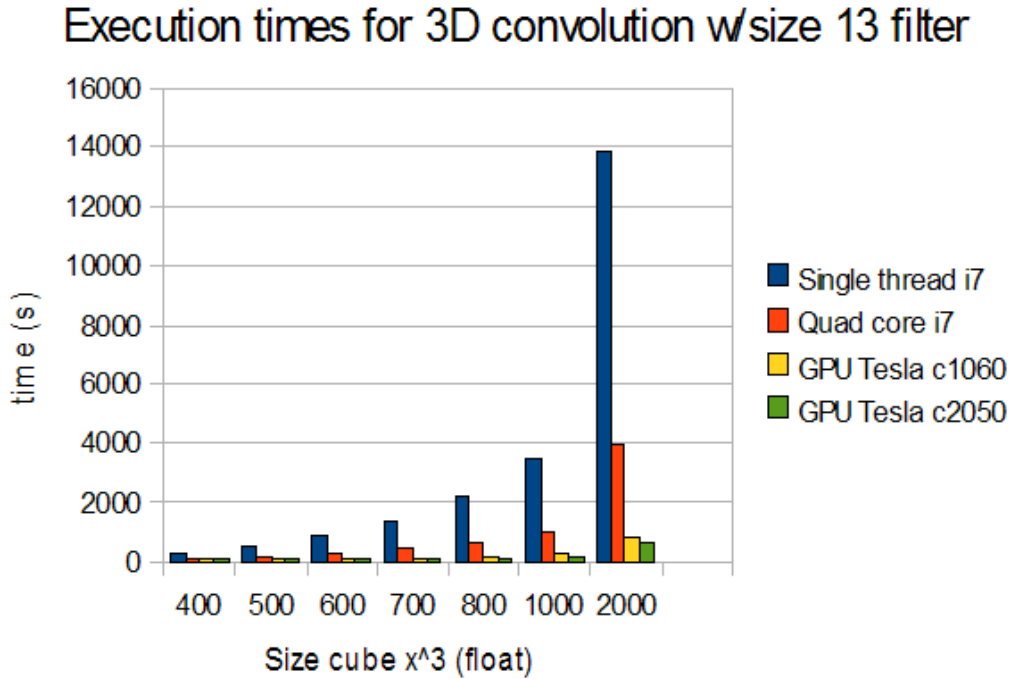


Figure 6.16: Execution time results for 3D convolution algorithm with filter size 13^3

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.

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 [9] 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 compiler 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. On the Fermi architecture on the other hand we have 100% occupancy because of the advantages the architecture permits.

Visually we will see a blurred image after convolution because of the use of a gaussian filter. The larger the filter the more blurred the image gets. For sample result see Figure 6.17.

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 [36]. 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 2.1) in that each pixel is completely independent of the others. Another interesting aspect in convolution

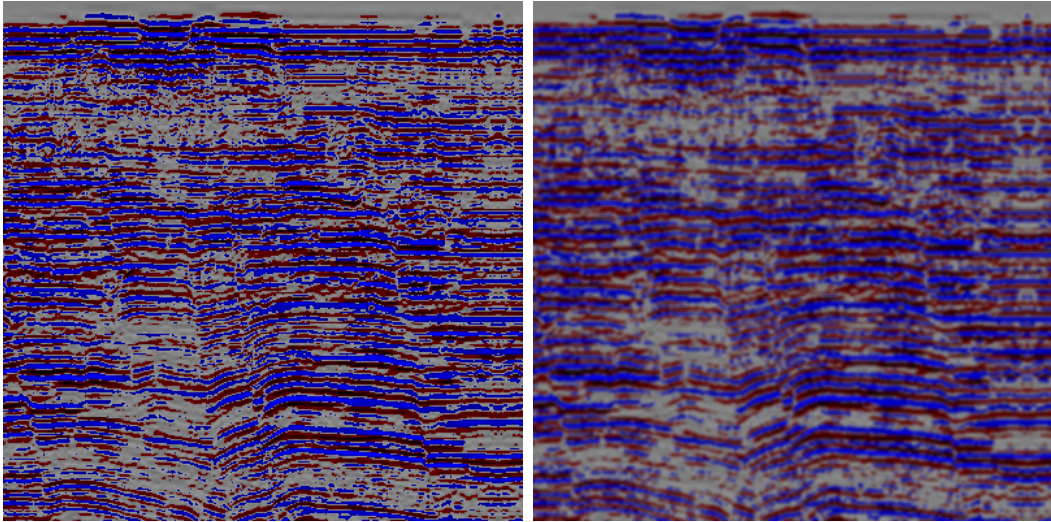


Figure 6.17: Blur using 3D convolution with filter size 13^3 , generated by our framework

is that calculations performed per pixel are actually quite time consuming in that each pixel must traverse 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.

Spatial vs. Fourier Domain

This thesis 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 performed in the fourier domain, but has its limitations. For example, The fourier domain is asymptotically faster than the spatial domain, but has limitations such as the data set dimensions have to be a power of two [1]. This is usually not the case for seismic data, and therefore one would have to pad the data, which could cause distortion or noise in the filtered image. This gives justification that performing convolution in the spatial domain is also useful in some cases, and to account for the limitations of the time available, our project focuses on the spatial domain.

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.

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 thesis, 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.

Memory Access Patterns in GPU Implementation

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 transferred from global memory [23]. 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 [23].

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 be 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.

What if Convolution Filter is Larger Than Constant Memory?

For filter sizes larger than $25 \times 25 \times 25$ 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.

6.4.2 Hough Transform Benchmarks

The Hough transform implementation is significantly different from 3D convolution. It does not have any overlapping subproblems, which makes it more parallel friendly. In this way each thread is entirely independent of the other. The other is that this algorithm is more memory access dependent, each thread does more memory writes than computations. For convolution one calculated all values within the filter to calculate one pixel, which lead to high computation and low memory bandwidth per pixel calculated. But, here for each pixel a set of angles are presented and a r value is calculated and for each r calculation a write is performed to add to the hough domain. This just means that it is more dependent on the bottleneck that the memory bandwidth imposes than the computation capabilities of the GPU and CPU even if we are to scale the number of point that should be calculated.

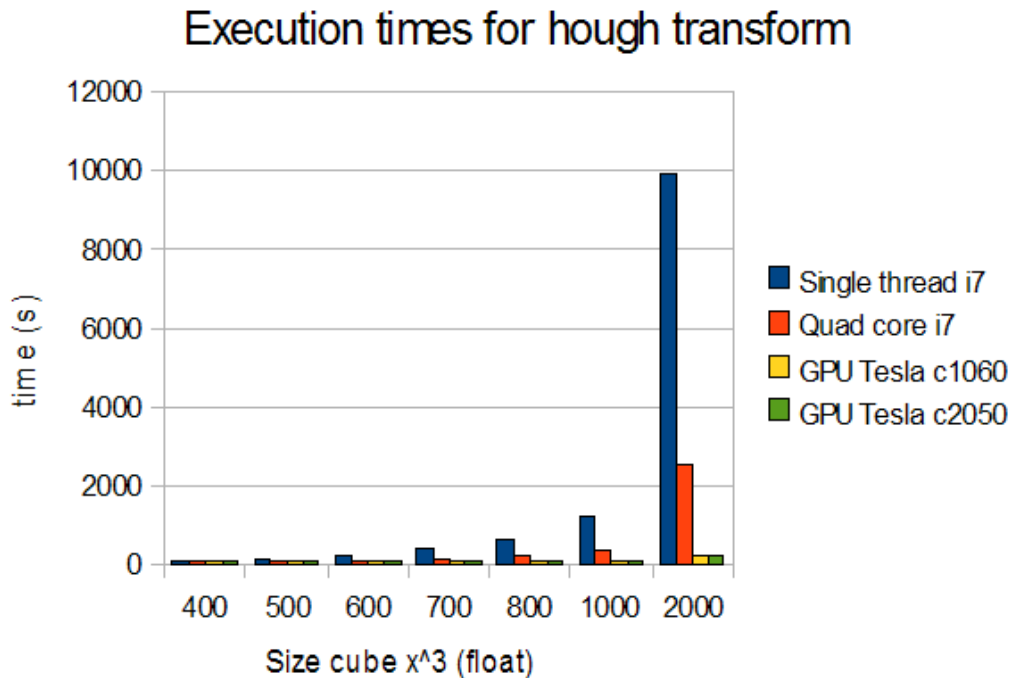


Figure 6.18: Execution time results for Hough transform algorithm

The results for visuals and the execution times of the Hough transform are presented in Figures 6.19 and 6.18 respectively. As we can see the algorithm scales well on the CPU in that a speedup of about 4 is gained for using 4 threads even on larger data sets. This mainly because the overhead of creating

the threads becomes irrelevant as one scales. Other significant speedup results are that of the GPU. By using the NVIDIA tesla c1060 we achieved a speedup of 50 compared to the single core CPU. This is a significant change, which is more or less expected in because the algorithm is so parallelizable. For the Fermi graphic card tesla c2050 it was even more, at a speedup of 64 from a single core CPU and a speedup of 1.28 from the older GPU architecture. This might be surprising because as we know the Fermi architecture provides both faster memory access and double as many computation core, and not to mention caching. Why did these aspects not have a greater influence on the end result?

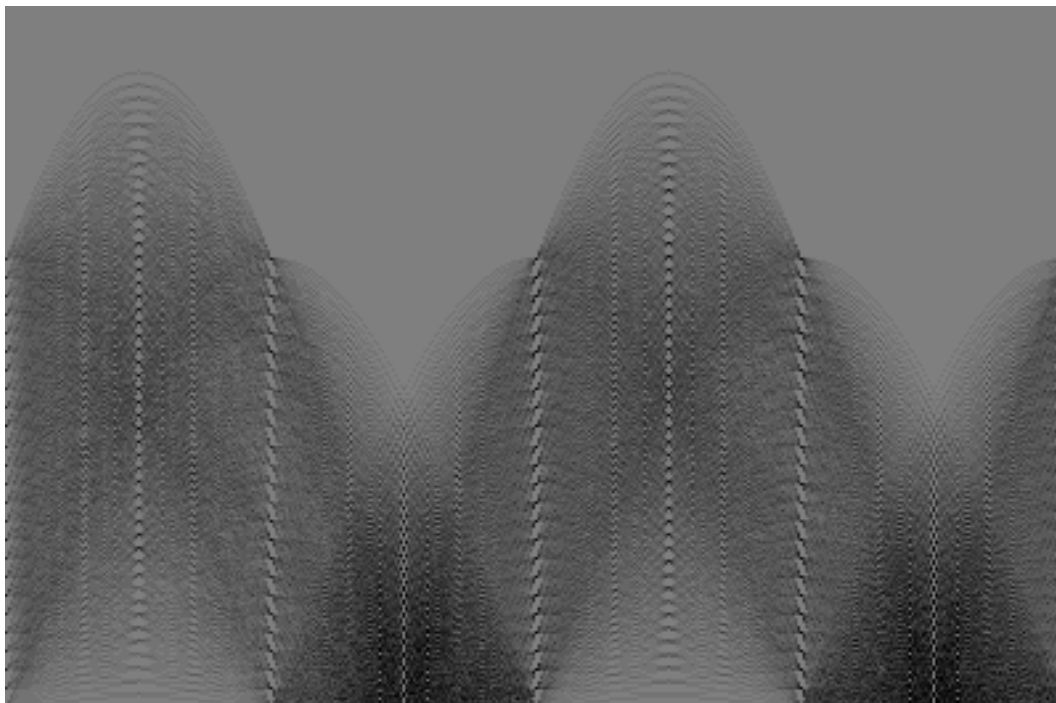


Figure 6.19: Visual results of the Hough transform, generated by our framework

Why do we achieve this speedup on the GPU? and why is there no significant difference on the Fermi architecture?

Given that we do have a 100% occupancy we know that all the cores on the GPU are performing computations. And given the parallel nature of the algorithm discussed earlier we expect a linear speedup when it comes to computations. When using the CPU and making use of several cores the speedup is linear, but this is only because 4 threads writing simultaneously does not

occupy the bandwidth of the memory. As we expand the algorithm on the GPU we see that of course a gain in computation is achieved in that more computation can be hidden, but the algorithm does not scale linearly because of the bottleneck of the memory. The bus to the memory is a 384 bit bus, which means that 48 floats can be transferred simultaneously. In contrast to the one float for the single thread and sequential algorithm that it is compared to, which results in a 48 times speedup on only this basis. But, this is just a rough estimate there are other factors here at play such as hidden computations, latency, queues of cores ready to write and their delay, but all of these together give a speedup of 50 for the GPU implementation.

How does the memory access pattern effect performance?

The issue with this algorithm is that it is more data intensive than computation intensive. This means that all the cores would need access to the bus connecting the memory and the computation units on the GPU. The bus and memory bandwidth will then be limiting the algorithm rather than the number of cores available and the computation intensities available. This is evident in the speedup gained by switching graphic card platforms. The fact that the spread nature of the writes of the algorithms creates a deficiency in the caching performed and thereby renders caching useless as an optimization this is both seen on the CPU and Fermi implementation in that the results do not benefit from the caching available. The memory access pattern of the algorithm however makes it dependent on memory bandwidth rather than computational efficiency.

Why do we not get a greater speedup when using the Fermi architecture?

The memory bandwidth is increased by 40% on the Fermi GPUs, and the execution time of the Hough transform is 30% more effective, given that some overhead is lost to latency and computation percentage, the results clearly show that the algorithms bottleneck is memory and bus bandwidth. Another interesting aspect is that the algorithm does not profit from the caching system available on the CPU and Fermi GPU because of the non predictive and spread nature of the algorithms writes. Values are cached for access but are not accessed often enough when in cache to make an impact. It might even be a source of latency that we mentioned earlier. Knowing that the Fermi architecture should give speedup because it has double the cores and faster

and cached memory is a viable hypothesis, but the extent to which it does so varies. In the case of the Hough transform it is limited by the memory bandwidth and the memory access nature of the algorithm.

6.5 Effects of Compression on the Seismic Filtering Process

Now that we have discussed the algorithms individually, in this section, we look at how the compression and filtering algorithms implemented on the GPU will effect the seismic process. The first section, Section 6.4.1, discusses how compression algorithms influenced I/O speedup on several platforms such as HDD disk, SSD disk, Quad core CPU and GPU, including the new Fermi architecture of GPUs designed by NVIDIA. Here the focus is on how this effected both synchronous and asynchronous I/O compared to normal I/O access. In Section 6.4.2, a closer look at the predictive model is presented. Here an analysis of the validity of the model and how well its predictions scale on various platforms are presented. In the final section, Section 6.4.3, a detailed look at how this I/O speedup can be used in two given scenarios of filtering is presented. One being with overlapping sub-problems and the other without, and an analysis of how much speedup can be gained form the whole process will also be discussed. Another aspect of the compression that will be taken into account here is the effects of the compression format on the process.

6.5.1 I/O speedup

The I/O speedup is measured as in Equation 6.2 where we compare the new I/O time using compression to the normal sequential I/O time. The aim is to study the advantage the compression algorithms give for disk access, and to map which option is the most effective. We have studied two scenarios of I/O one being synchronous and the other asynchrnous. Pur test were also conducted on differet disk platforms. Two hard disk drives (HDD) of varying transfer rates. One being 40 MB per second and the other 70 MB per second. We have also tested on a solid state disk (SSD), which is of newer technology and has a speed of 140 MB per second for transfers. That is double the speed of the faster HDD disk.

6.5. EFFECTS OF COMPRESSION ON THE SEISMIC FILTERING PROCESS

$$I/O \text{ speedup} = \frac{\text{Normal Sequential I/O time}}{\text{New I/O time using compression}} \quad (6.2)$$

In Figures 6.20 and 6.21 we have displayed the I/O speedup results for the different platforms. In Figure 6.20, we have displayed the results for the synchronous runs, and Figure 6.21 presents the asynchronous results. The resulting execution times that are presented are the only the fastest, which means that some are performed on only the CPU and other with the aid of the GPU. The lossless algorithms as discussed earlier are the ones that benefit mostly from the CPU, and the transform encoding algorithms are run on the GPU for the same reasons. The CPU used to benchmark these results is the Intel i7 965, and the the GPU is the NVIDIA tesla c2050. Again this is the case because they gave the best performance, and to see how they performed compared to other alternatives see Section 6.3

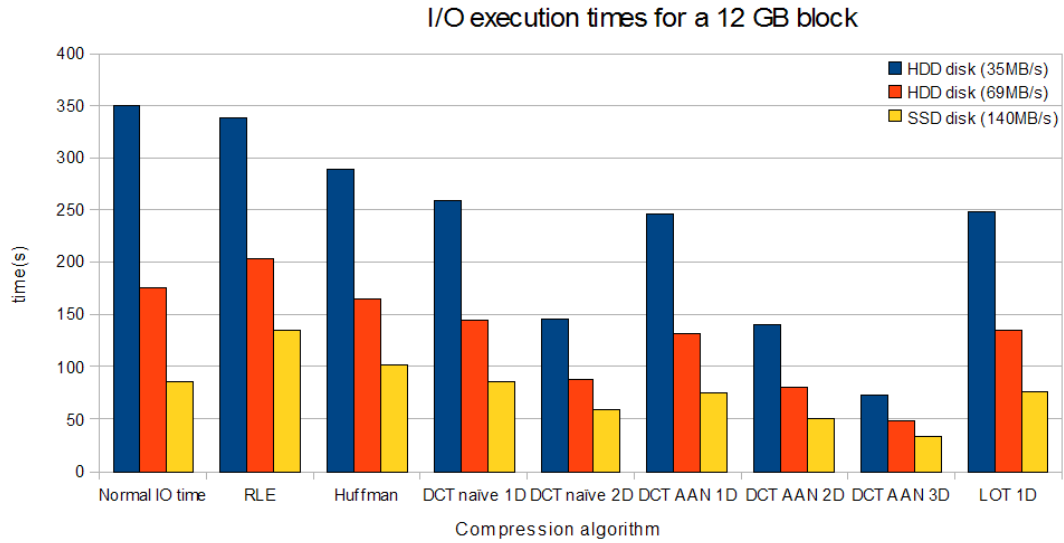


Figure 6.20: Execution time results for synchronous I/O

One can clearly see the advantages of using asynchronous I/O when it comes to hiding computations and communication times. In most cases, the speedup achieved for the asynchronous I/O was very close to the maximum speedup one is able to gain. And this comes from little delays in the beginning and end of the asynchronous I/O. For these tests we have used a compression format of dividing the input data into smaller pieces such that the larger files were 200 pieces large. This decreased the maximum compression rate slightly, but gave the great speedup that we achieved.

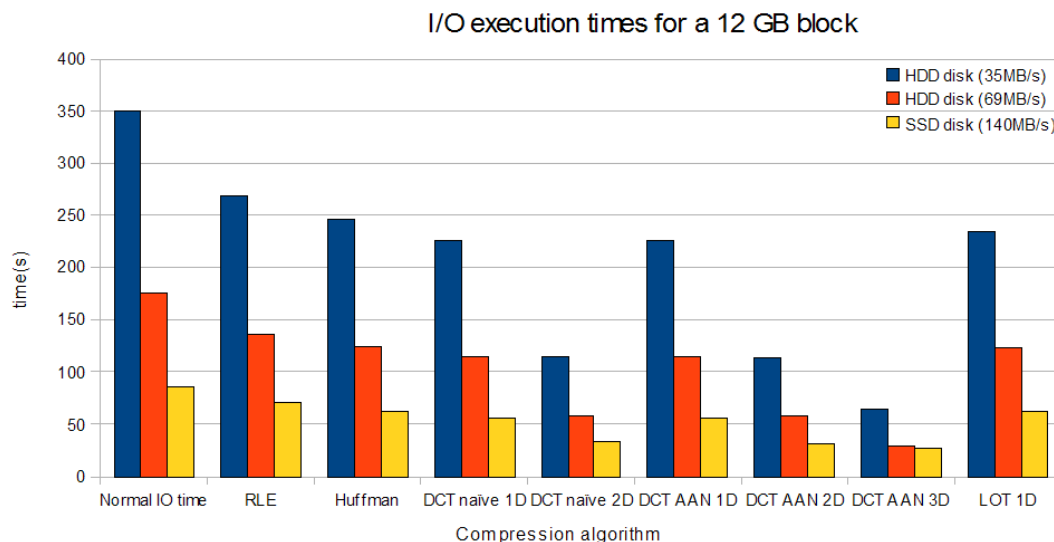


Figure 6.21: Execution time results for Asynchronous I/O

Why does Asynchronous I/O not give desired speedup results on faster I/O platforms for some of the compression algorithms?

There are some trends here that need a closer look such as the 3D AAN actually becomes faster as we increase disk speeds to begin with, but becomes slower on the SSD disk. This is actually not that surprising for the asynchronous case. As we explained earlier in the asynchronous case, for each sub block one of the threads will be waiting for the other to finish. Here one of the threads is used for reading from disk, while the other is performing computations. If the computations are more time consuming than the reads then they will dominate the execution time and the speedup will become less. This is because the speedup is compared to disk speed, and if the computations are not fast enough than the speedup will be limited. This is what we actually see happen for the SSD disk, where the disk is faster than the algorithm and therefore we get a speed down compared to when using a slower disk. But, in the other case where the I/O time is still dominant over the computations such as they are for the HDD disks tested, a speedup is obtained by going to a faster disk, and resulting in our best result yet, which is an I/O speedup of 6. Another example of the effects of a too slow computation algorithm is the naive 2D DCT and AAN 2D DCT, where they both have the same execution time, but one is dominated by computation and the other by communication time.

What effects do the different platforms have on I/O speedup?

The tests are conducted on several platforms to explore how the speed of the disk affects the outcome of the speedup. The trend for the synchronous approach is the speedup is less for faster platform, which is expected since the platforms it runs on are quite fast to begin with. But, when comparing the average throughput of the algorithm from disk to memory the algorithms with compression run on faster disks are much faster. In fact, comparing this change to the original slow disk we get speedups of about 10. Nevertheless we compare the speedups to the current disk one uses. This is because if that is the disk available on the machine, than that will be the disk to beat for I/O time. Some similar tendencies are experienced for the asynchronous implementations, but with the exception that here the results depend on which element is mostly dominant in, communication or computations. We explain how this affects speedup earlier, but it is worth noting that the trend explained earlier is present for all the algorithm. As they are performed on the SSD disk the speedup is limited differently, but the throughput is even greater than that of the synchronous.

Why does asynchronous I/O out perform the synchronous approach?

The major advantage of the asynchronous I/O is that one gets to hide computation and communication time in parallel. This actually gives the advantage of being efficient in all cases as long as the computations are not dominating the communication. This means that as long as one has a compression and is reading less data, then by performing computations in the background of reading the compressed data, one will get speedups. Making asynchronous an approach that is useful even for lossless and slow algorithms that give little maximum speedup. This is seen when performing RLE on all three platforms. The maximum speedup of the algorithm, which is the compression rate is at 1.3. And the speedup obtained from the different platforms is 1.3(HDD1), 1.29(HDD2) and 1.24(SSD), which is really close to optimal. This is what makes asynchronous I/O beat the synchronous approach. There are of course limitations, such as if the computations are dominant and the disk one runs on is even faster than that which we are testing one then the speedup will be limited, but in all cases it will beat synchronous I/O time because one gets to hide the communication time. And in the extreme case of having really fast close to zero disk access time, the two approaches will be similar in execution time, but otherwise asynchronous will always outperform the synchronous approach.

6.5.2 Predicted Model

The focus in this section is on how well the predicted model matches the actual results. We have tested the model by using a couple of measurements on a smaller block, and given those results we estimate the execution time of the larger blocks of data. A graph representing the estimated and actual execution times for the synchronous model is shown in Figure 6.22. While a graph representing the asynchronous predicted and actual results is shown in Figure 6.23. The mathematical definitions used to estimate these predictions are from Chapter 5, and were tested to estimate all the behavior of the various algorithms.

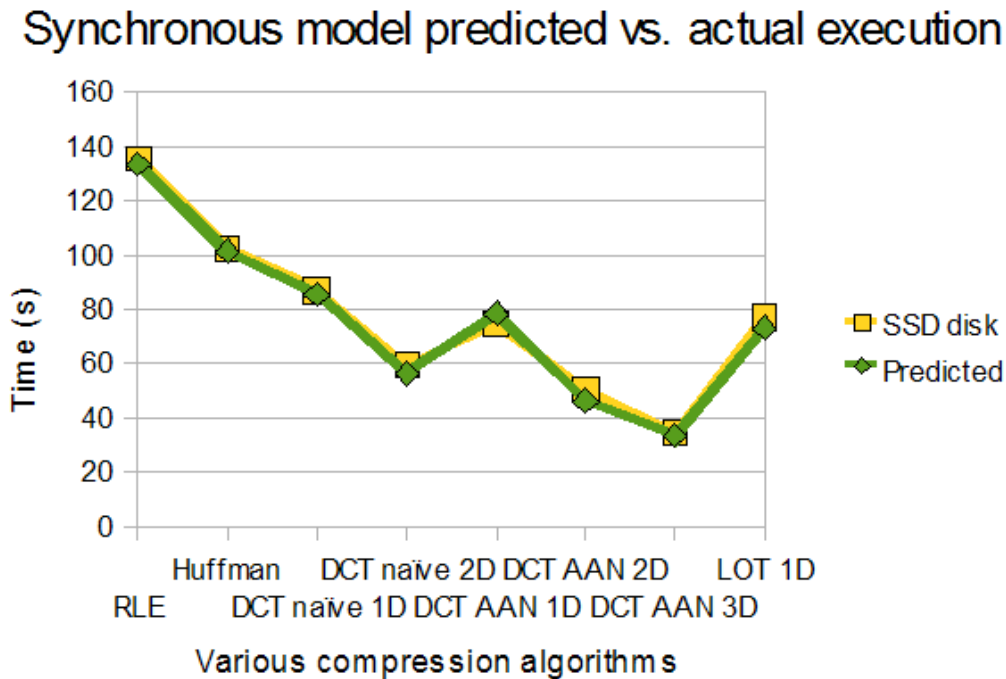


Figure 6.22: Predicted execution time for synchronous model

Given that the models used are quite simple, the predictions were very close to the actual execution times and performed well. There are of course some error that can occur in that some functions become more evident in their scaling such as standrac c functions such a memcpy() and memset(). But, the results prove that these estimates can be used with an error of $\pm 2\%$, which is quite accurate given the simplicity of the models.

Asynchronous model predicted vs. actual execution

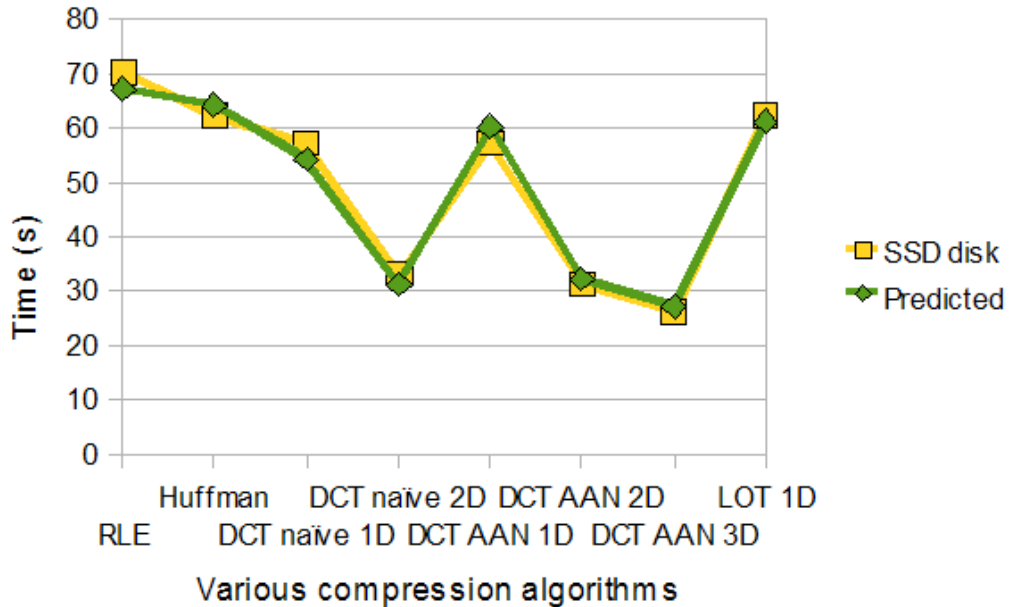


Figure 6.23: Predicted execution time for asynchronous model

What are the sources of error for the predictive model?

The predictive model takes into account the simple interactions of the disk and computation units, but there are other factors that effect performance that are harder to predict. One of the major source of error is the effects of the operating system. All processes in the operating system are given a priority and are cued such that they get their share of computations. What type of cue of the algorithms behind this are not predictable since they adapt and change, and in such cases as for the windows OS they are not known for the public. These effects are not visible for the short executions of the program, which are used for the model to predict the outcome. This is why the prediction might be over estimating the execution times, but this is a changing process. Depending on what is running and which operating system its running on, one will get different behavior.

Another source of error is that the algorithms used do not scale as simply as our model predicts. We run short tests were we try to use the throughput of the algorithms to estimate for larger sizes of data. This can be useful for simple estimation, but at the same time some of the algorithms scale differently

and therefore give different throughput as they scale. Most likely the change will not be severe enough to cause a totally erroneous estimate, but it has its effects.

Why are the errors more evident in the asynchronous model?

When estimating the asynchronous model there are more uncertain factors. Like in the case of having a compression algorithm and disk with similar throughput on smaller datasets then as they scale the dominance is changed. When estimating the asynchronous model we use the MAX function to see which is dominant I/O or compression and to use that as an estimate for which process will overshadow the other. This might not be the same case for smaller and larger datasets and therefore adds more error to the prediction. But, generally speaking the error will again not be that evident. Another source of error is that one would have two threads and thereby more uncertainty from the operating system when it comes to scheduling as explained previously.

Given the accuracy of the model what can it be used for?

The accuracy of the model is within 2% error and this is quite good for a prediction of such long execution times. The prediction is accurate enough to use for scheduling such tasks and even for managerial planning. This estimation can be used to see if the method about to be used is useful or if there are better approaches without having to run for hours on larger sets of data. It can also be used to aid in the planning phases of certain projects that will need this type of I/O acceleration. The model can be used in a scheduler or load balancing locally in some software that can estimate given a type of data and the compression available, the execution times that are to be expected, and on this basis choose and execute the most efficient. The model however can not be used for estimating smaller sets of data in which other factors play a role, such as latency and bandwidth, OS scheduling and so on. In these cases, one would need a more accurate and replicative model for a good estimation. But, in our case of large datasets, these factors are overshadowed by the long computation and communication times.

How can the accuracy be improved?

The accuracy of the model can be improved by taking into account the factors mentioned previously. Factors such as the latency and bandwidth, OS scheduling, GPU transfer times, and so on. In other words the model should take into account more of the representative stages in the computation and communication processes. As the model is now it is simplistic, but also useful for the cause we wish to use it. This of course does not mean that it should not be improved or tweaked for special use in other fields.

6.5.3 Seismic Filtering Process Speedup

In this section, an analysis of the compression algorithms to speedup the seismic filtering process is presented. The focus is on the use of the different formats to perform convolution and hough transform on the seismic data. The significance of these tests is the fact that the overlapping sub problems of the convolution process will cause difficulties for the compressed I/O process and to solve this we would have to compromise the efficiency of the Hough transform. This way we would find a middle ground for the nature of the filtering process of seismic data and see what speedups we could expect from this.

Figure 6.24 presents the execution times of the hough transform in under normal circumstances and using the different compression formats discussed earlier. Format 1, is where we have large blocks of data compressed at a time. This gives good compression, but makes it hard for overlapping sub problems. Format 2 is where we include the neighboring pixels in the block of data as we compress, this makes it efficient for overlapping sub problems, but creates issues for the in that the compression rate goes down and slows the problems that do not overlap. And Format 3 is where we compress small blocks of data at a time, this effects compression rate a bit, but opens up possibilities for both over lapping and non overlapping sub problem filtering algorithms.

The results show that for the Hough transform, the most effective format is format 1, and the least effective is format 2. In Figure 6.24 we present similar results, but for the 3D convolution algorithm. Here we see that the most effective format is format 2, and the least effective is format 1. While in both cases, for Hough transform and convolution, format 3 is a good middle ground solution.

In Figure 6.25 we show the speedups of both the hough transform and 3D convolution as part of the entire seismic filtering process are presented for

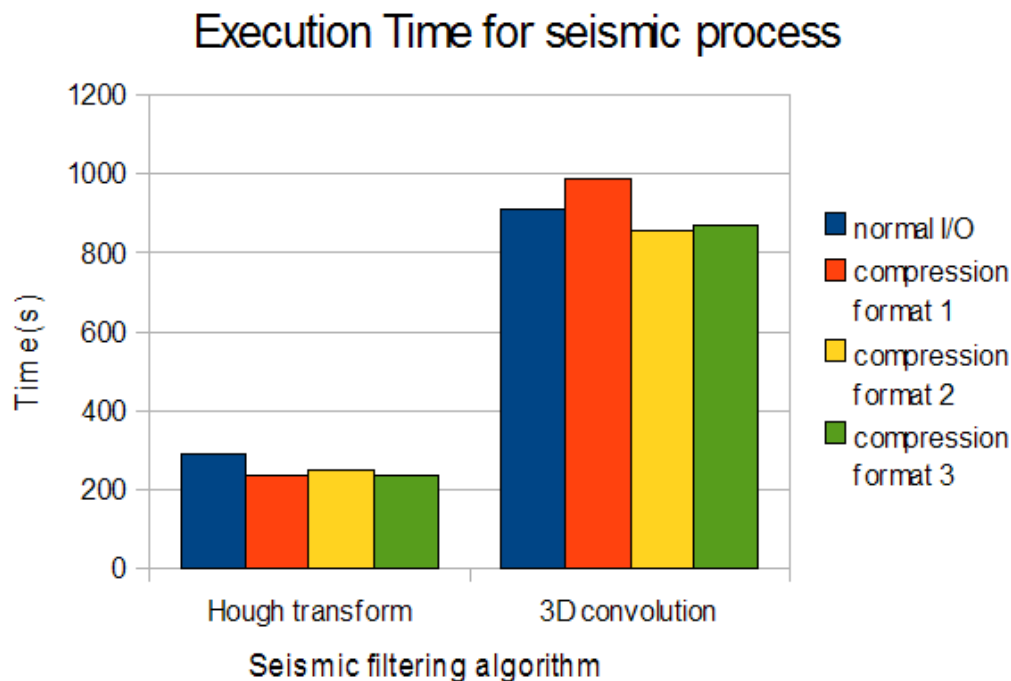


Figure 6.24: Effects of compression formats on seismic process for the seismic filtering algorithms 3D convolution and Hough transform

scaling data sizes. The speedup is compared to the alternative of reading the data sequentially from disk and performing the calculations on the CPU. Our solution to speed this process involves using the GPU and compression. One can see that the speedup for reading and filtering the seismic data with the hough transform is 25, and for 3D convolution it is 16.5. In both cases this speedup is consistent as the problem size scales. This is the case when using HDD disks. While when using SSD disks a speedup of 19 is achieved for the hough transform and a speedup of 13 is achieved for 3D convolution.

When should one use the different formats?

From the results it is obvious that the first format works well for processing non overlapping sub problem filtering algorithms such as the hough transform. This is not surprising because this format gives the highest compression rates and therefore the most speedup, and since there are no dependencies forced by the algorithms, this format should be used for all algorithms that match the nature of the hough transform. The second format, which includes neighboring

6.5. EFFECTS OF COMPRESSION ON THE SEISMIC FILTERING PROCESS

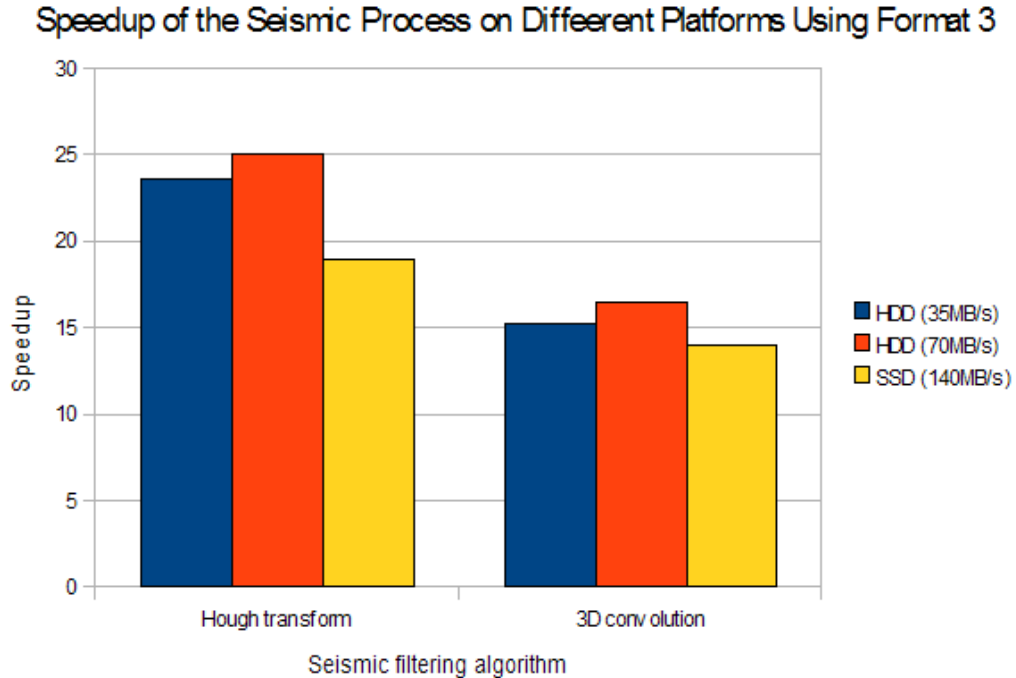


Figure 6.25: Speedups of the seismic process given a seismic filtering algorithm for compression format 3

values, limits compression in that there is more to be compressed and therefore the compression rate drops. This reduction in the compression rate results in a reduction of the I/O speedup attained from using compression. But, this is faster than having to decompress several blocks to compute the filter for a single block. And therefore the tradeoff works only positively on the process and this format is recommended for all algorithms that need neighboring data. If one is to use both overlapping and non-overlapping sub problem algorithms then the third format is the best compromise. But, there is a thin line there as well. If the dominant calculations are those of an overlapping nature then format 2 is recommended. However an alternative such as format 3 is good to have as one adapts to the needs of the process.

Why does format 2 have a major effect on the non-overlapping sub problem algorithms?

By including the neighboring values in the block to be compressed one is duplicating data in the compressed blocks. This will then result in more data to

be compressed than the actual amount of data. This effects the I/O process in that the compression rates will be reduced, and by reducing them the maximum I/O speedup will be limited. Compromising a bit of I/O speedup to accelerate a slow process such as overlapping sub problems is a good tradeoff, but for algorithms such as the hough transform that do not need the neighboring data this only has a negative effect.

What do the speedups achieved entail?

the speedups achieved for the seismic filtering process prove that by the use of the GPU and compression the existing seismic process can be improved dramatically. The compressibility of seismic data with the use of transform encoding is able to accelerate I/O and the parallel nature of the filters applied to the data the GPU is able to accelerate them several times over. Our finding then entail that it is wise to adapt the GPU as part of the seismic process and that compression can be an alternative to accelerating disk access through hardware changes. The results also entail that the cooperation between CPU in GPU can prove to be more useful than simple comparisons between the two, and performance can be greatly accelerated by identifying and using the advantages each platform brings. Such as in the case of compression.

*6.5. EFFECTS OF COMPRESSION ON THE SEISMIC FILTERING
PROCESS*

Conclusions and Future Work

One of the main challenges of developing high performance applications on modern computer architectures is to overcome limitations in memory and disk bandwidth and latency. Today these are much slower than computation speeds and are often the bottleneck in computer systems. In this thesis, we focused on optimizing compression methods for signal and image data to accelerate disk access. Large seismic datasets were used as test cases. Our methods were developed on both CPU and GPU to see if there are any advantages in the architectural features to be exploited. When the files are compressed than the bandwidth limitation would apply to only reading the compressed file. Then by utilizing computation capabilities of the CPU and GPU for decompression, one would accelerate access to these data. Both lossless and lossy compression methods were implemented and tested within a developed framework that runs both synchronous and asynchronous I/O. We also looked at using both the GPU and CPU to gain an even further advantage. A mathematical model and a compression library were also produced. the I/O speedup gained and the effects on the seismic process.

The next section, Section 7.1, summerizes our results, including algorithm performance, I/O speedup and our results impact on the seismic process. Possible future work within the field is presented in Section 7.2.

7.1 Conclusions

The lossless compression algorithms that were tried were RLE (run length encoding) and Huffman encoding. These were tweaked and optimized for compressing of seismic data, and resulted in a compression ratio of 0.83 and 0.71 on

7.1. CONCLUSIONS

all platforms tested. Both of these algorithms were chosen on the criteria that they are fast to perform. However, since they do not compress much of the data, the bandwidth bottleneck was still evident. Nevertheless, they resulted in 1.08 and 1.1 speedup respectively in disk access in the synchronous method and 1.3 and 1.4 speedup in the asynchronous method. This was tested on a HDD disks that have an average transfer rate of 35MB/s and 70MB/s. Note, However, they both gave negative speedup results when run on faster platforms such as SSD disk which averaged 140MB/s transfer rates, which proves one of two things either the algorithms are slow or the compression is little. When tested on the GPU these algorithms performed slower than on the CPU. This has much to do with the fact that the GPU is slower on bit-wise operations and since both algorithms have a sequential nature, we are not able to use the vast parallel computation capabilities of the GPU. In other words, the CPU is superior in this case.

We were, however, much more successful when it came to lossy compression. Seismic data is typically noisy data, which makes it hard to compress. However, by filtering the noise and transforming the data to the frequency domain, one can achieve great compression rates with little error. We experimented with transformations in several dimensions and used algorithms that were usually used in image compression such as the DCT (discrete cosine transform), and the LOT (lapped orthogonal transform). The best compression rates were achieved by using the DCT in 3 dimensions and combining this with a modified RLE. This gave a compression ratio of 0.16. the transform in this case was performed on the GPU because of its parallel nature (which showed an 8 time speedup compared to the CPU), and the RLE was performed on the CPU because of its sequential nature. Testing on HDD and SSD platforms we were able to achieve respectively 3.7 and 2.5 speedup on the synchronous model, and a speedup of 6 and 3.3 on the asynchronous model. This was done with an average error of 0.46% per float in the seismic data, which is within a reasonable loss of two decimal places. We later tested for LOT to see if we could reduce the error, but results show that this effects speedup and compression size more than the error term and that is why we did not continue to look at this any further.

A mathematical model was further on developed and empirically tested with our results. The model proved to be accurate, with up to 5% error in some cases, despite its simplistic nature. This model can be used to estimate running larger data sets, by measuring variables on smaller sets and adapting the model to the device it runs on. The intention of the model is to give an estimate of execution time for I/O on a system using compression both synchronously and

asynchronously, and it proved to be reasonably accurate for both.

Our Compression implementation was tested on a larger scale in the seismic filtering process. We tested it on two typical algorithms that were used on seismic data. One being 3D convolution, which is used for filtering in general. The other is the Hough transform, which is used to detect lines in the seismic data. By executing these algorithms on the GPU (NVIDIA Tesla c2050 Fermi) we achieved a speedup of 23 for 3D convolution and a speedup of 62 for Hough transform. The Hough transform is a more parallelizable algorithm in the sense that there are no dependencies and a good non-colliding memory structure. Whereas 3D convolution requires a lot of memory jumps and is harder to accelerate. Nevertheless we felt it necessary to test and see how the GPU could accelerate both I/O and computations on seismic data. The end result when performing convolution was a total speedup of the system by 16.5. This is because 3D convolution is time consuming that most of the time is used on computations anyway. While for the Hough transform the total speedup was 25. This means that by using compression and the GPU to do calculation on seismic data this would result in a speedup of 16.5 or 25 on the entire process for overlapping and non overlapping sub problems respectively.

7.2 Future Work

Presented below are suggestion for future work within the researched field.

Autotuning: One of the major challenges GPU programmers are facing is the blocking of the data such that separate threads can perform calculations on them. The fact is that per today it is the programmer that has to chose the block size and thread count and match it with the capabilities of the GPU and the kernel running. This effects performance greatly and sometimes by running with more threads or a rewrite of the kernel to distribute the work between the threads differently, can make the difference. This is a problem for both CUDA and OpenCL. NVIDIA has its CUDA calculator to help choose an optimal thread count to gain most occupancy, but it is also stated that anything above 50% occupancy will give an optimal execution situation. This comes from the scheduler that runs these kernels. A very useful thing to work on further is an autotuning program that is able to chose these factors for the programmer to optimize the running of the kernels on the GPU. This will make GPU programming easier and will automatically give the optimal running. One could even experiment with assembly level autotuning that could help rewrite the kernels to be more optimal aswell.

7.2. FUTURE WORK

3D visualization: During this thesis we have been producing 2D images of the compressed data by decompressing and filtering then storing the data uncompressed before visualizing it. This is of course not the case when one will be dealing with this data in a seismic application. This is why one should look at ways to present the data in its compressed format. The challenge here is that the compression is non uniform, meaning that one does not know which compressed block has which part of the original image. This makes even 3D representation challenging and lesser effective in some cases. 3D representation can be done with the help of the level of detail algorithms such that one selectively selects what to show given the scope. Visualization is definitely an area where compression will have great effects and present challenges for large data sets, where one has to decompress several blocks to select a few datapoints.

GenLOT: During this thesis we have been performing many transforms on the GPU in CUDA, and have mentioned how building upon the fast DCT would give room for more speedup. In our case of compression to beat I/O it did not help speedup when we increased to the next step, namely LOT. But when looking at general compression of seismic data the GenLOT is used, and a CUDA implementation of this would be interesting for testing how well the algorithm runs on the GPU. The preliminary tests show that good speedup is expected in running on the GPU.

Hardware: because of time limitations in this thesis, we were not able to optimize our code to run on the Fermi architecture. Although we have tested it and showed how well it performs one can look at optimization for the use of caches in the new architecture. Another aspect that can be tested is multiple GPUs for compression. Although we tested that convolution scales almost perfectly by using several GPUs [4], one should also test for the other transforms we have created in this thesis. Preliminary results show that the problem will scale well. Luckily, with the creation of the compression library it should be easier.

Model: The predictive model in some cases gives an error up to 5%. This can be improved by adding more complexity to the model and expanding on traits such as estimating latency and communication times.

OpenCL: Since the implemented algorithms are to be run on multiplatforms in the library, one can look at using OpenCL to make the transition easier. There can be some optimization limitations since the compilers for C and CUDA are more optimal than that of OpenCL at the moment, which is why we avoided using it. But, in the future when this is no longer an issue it is an

good alternative to look into.

7.3 Closing Remark

By optimizing filtering of seismic data we quickly learned that the systems I/O bandwidth and latency was a limiting factor. This is the case for most optimizations performed on newer platforms because of the gap between memory bandwidth and processor performance. We believe that our work approaches a viable solution to narrowing this gap with an alternative other than a hardware solution. This work can create further possibilities for compression acceleration in that it has laid a foundation that others can build upon, especially with our library. We will not be surprised to see more approaches to narrowing the bandwidth/throughput gap in the future as it is one of the main bottlenecks on modern systems.

7.3. *CLOSING REMARK*

Bibliography and References

- [1] V. Aarre. Schlumberger stavanger. personal communication.
- [2] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transform. *IEEE Trans Computing*, 1974.
- [3] S. Akhter and Jason Roberts. *Multi-Core Programming: Increasing performance through software multi-threading*. Intel press, first edition, 2006.
- [4] A. A. Aqrawi. 3d convolution of large datasets on modern gpus. Norwegian University of Science and Technology, 2009.
- [5] Y. Arai, T. Agui, and M. Nakajima. A fast dct-sq scheme for images. *Transactions of the Institute of Electronics, Information and Communication Engineers*, 1988.
- [6] P. Cassereau, D. Staelin, and G. de Jager. Encoding of images based on a lapped orthogonal transform. *IEEE Transactions on Communication*, 1989.
- [7] IEEE Circuits and Systems Society. Ieee standard no.1180-specifications for implementation of 8x8 inverse cosine transform. *IEEE Technical report*, 1991.
- [8] Microsoft Corporation. Microsoft msdn c++ library. <http://msdn.microsoft.com/en-us/library/default.aspx>, accessed 2010-03-02.
- [9] Nvidia Corporation. Nvidia cuda occupancy calculator [online]. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls, accessed 2010-04-31.
- [10] Nvidia Corporation. Nvidia cuda reference manual [online]. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUDA_Reference_Manual_2.3.pdf, accessed 2009-12-02.
- [11] G. Davis and A. Nosratinia. wavelet based image coding: an overview. *applied and computational control, signals, and circuits*, 1998.

BIBLIOGRAPHY AND REFERENCES

- [12] R. L. de Queiroz, T. Q. Nguyen, and K. R. Rao. The genlot: Generalized linear-phase lapped orthogonal transform. *IEEE Transactions on Image Processing*, 1996.
- [13] R. Duda and P. Hart. Use of the hough transform to detect lines and curves in pictures. *Communications of the ACM*, 1972.
- [14] L. C. Duval, V. Bui-Tran, T. Q. Nguyen, and T. D. Tran. Genlot optimization techniques for seismic data compression. *IEEE Transactions on Image Processing*, 2000.
- [15] L. C. Duval and T. Q. Nguyen. Seismic data compression: a comparative study between genlot and wavelet compression. *Proceedings of SPIE conference on wavelet applications in signal and image processing*, 1999.
- [16] R. Eidisen. Comparing cg and cuda implementations of selected transform algorithms. Norwegian University of Science and Technology, 2008.
- [17] B. Flury. *A First Course in Multivariate Statistics*. Springer Verlag, first edition, 1997.
- [18] J. E. Fowler and R. Yagelt. Lossless compression of volume data. *IEEE Transactions on Image Processing*, 1995.
- [19] R. Gerber, A. Bik, K. Smith, and X. Tian. *The software optimization cookbook*. Intel press, second edition, 2006.
- [20] D. Haugen. Seismic data compression and gpu memory latency. Norwegian University of Science and Technology, 2009.
- [21] J. Hennessey and D. Patterson. *Computer Architecture a quantitative*. Morgan Kaufman, third edition, 2003.
- [22] R. Hovland. Latency and bandwidth impact on gpu systems. Norwegian University of Science and Technology, 2009.
- [23] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors*. Elsevier INC., first edition, 2010.
- [24] P. Komma, J. Fischer, F. Duffner, and D. Bartz. Lossless volume data compression schemes. *Tagung conference on simulation and visualization*, 2007.
- [25] E. Kreyszig. *Advanced Engineering Mathematics*. Peter Janzow, 8th edition, 1999.
- [26] C. Larsen. Schlumberger stavanger. personal communication.

BIBLIOGRAPHY AND REFERENCES

- [27] C. Larsen. Utilizing gpus on cluster computers. Norwegian University of Science and Technology, 2006.
- [28] J. Makhoul. A fast cosine transform in one and two dimensions. *IEEE Transactions on Acoustics, Speech, and Signal Processings*, 1980.
- [29] H. S. Malvar. *Signal Processing with Lapped Transforms*. Artech house, first edition, 1992.
- [30] H. S. Malvar and D. H. Staelin. The lot: Transform coding without blocking effect. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1989.
- [31] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Nvidia Corporation*, 2008.
- [32] OpenMP. <http://openmp.org>, accessed 2009-09-11.
- [33] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J Phillips. Gpu computing. *Proceedings of the IEEE*, 2008.
- [34] D. Patterson. University of berkley. personal communication.
- [35] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression*. Van Nostrand Reinhold, 1st edition, 1993.
- [36] R. E. Woods R. C. Gonzales. *Digital Image Processing*. Prentice-Hall PTR, third edition, 2008.
- [37] P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. *IEEE Computer society*, 2006.
- [38] J. Reinders. *VTune Performance Analyzer Essentials: Measurement and tuning techniques for software developers*. Intel press, first edition, 2005.
- [39] S. Saha. Image compression from dct to wavelets: a review. *ACM*, 2000.
- [40] D. Salomon. *Data Compression the complete reference*. Springer, fourth edition, 2007.
- [41] Seg Technical Standards Committee. *SEG Y rev 1 Data Exchange Format*, 2002.
- [42] H. Sorenson, D. Jones, M. Heideman, and C. Burrus. real valued fast foirier transforms. *IEEE Transactions on Acoustics, Speech, and Signal Processings*, 1987.

BIBLIOGRAPHY AND REFERENCES

- [43] D. Spampinato. Modeling communication on multi-gpu systems. Norwegian University of Science and Technology, 2009.
- [44] G. Statchev *et al.* Using graphics processors for high-performance computations and visualization of plasma turbulence. *Computing in Science and Engineering*, 2009.
- [45] B. D. Tseng and W. C. Miller. On computing the discret cosine transform. *IEEE Transactions on Computers*, 1978.
- [46] A. B. Watson. Image compression using the discrete cosine transform. *Mathematica*, 1994.
- [47] M. Wilkes. The memory gap and the future of high performance memories. *ACM Sigarch computer architecture news*, 2001.
- [48] B. Wilkinson and M. Allen. *Parallel Programming*, pages 3–26. Prentice-Hall PTR, second edition, 2005.
- [49] S. Winograd. On computing the discret fourier transform. *Mathematics of Computation*, 1978.
- [50] X. Xie and Q. Qin. Fast lossless compression of seismic floating-point data. *IEEE Computer society*, 2009.

APPENDIX A

Orthogonal Transform Theory

This chapter includes more detailed descriptions of the DCT, AAN and LOT algorithms. They are presented in that order.

A.1 DCT (Discrete Cosine Transform)

Like all Fourier-related transforms, the DCT expresses functions as the sum of sinusoid with different frequencies and amplitudes. And like the DFT the DCT operates on a function at a finite number of discrete data points. The distinction between the two, which is apparent from the name, is that the DCT only uses cosine functions, while the DFT uses both sine and cosine in the form of complex exponentials. However, a deeper distinction is that the DCT implies different boundary conditions than other related transforms. DCTs are essentially DFTs of real-even data, meaning that one can design a fast DCT algorithm by taking an FFT and eliminating the redundant operations due to the symmetry. This is what makes the DCT compute efficient.

There are many definitions of this transform, in this thesis we will be using the IEEE standard 1180 definition [7], which is the most common and even used in the JPEG standard [35].

Given n real numbers x_t for $t=0, \dots, n-1$

The forward discrete cosine transform of x_t , Y_f for $f=0, \dots, n-1$, is given by

$$Y_f = \sqrt{\frac{2}{n}} C_f \sum_{t=0}^{n-1} x_t \cos\left(\frac{(2t+1)f\pi}{2n}\right) \quad (\text{A.1})$$

A.1. DCT (DISCRETE COSINE TRANSFORM)

The inverse of the discrete cosine transform of Y_f , x_t ($t=0, \dots, n-1$) is given by

$$x_t = \sqrt{\frac{2}{n}} \sum_{f=0}^{n-1} C_f Y_f \cos\left(\frac{(2t+1)f\pi}{2n}\right) \quad (\text{A.2})$$

Where in both the forward and inverse C_f is given by

$$C_f = \begin{cases} 1/\sqrt{2} & \text{for } f = 0 \\ 1 & \text{for } f > 0 \end{cases} \quad (\text{A.3})$$

Another interesting aspect about the definition of the DCT is that it is separable. That makes it easy to expand to l dimensions. This is done by performing the transform on a dimension and then use the transformed data to transform in another dimension. The important thing is that one performs the inverse on the transformed data in the same order it is transformed. This can be performed up to l dimensions.

What makes the DCT better for compression than DFT? First of all the DCT is better at compacting energy at low frequencies compared to the DFT that spreads them throughout a larger frequency spectrum. This results in more loss when cutting frequency values in the quantization process. This is what makes the DCT more efficient for compression/encoding. The fact that there will be many zeroes in a row in an image that will result in better compression. Another aspect worth noting is the fact that the DCT is a real transform, meaning that it only has real numbers unlike the DFT, which has complex numbers. This makes it even easier to encode than the DFT because of not having to account for the complex numbers. For a more in depth study of the comparison of the DCT and DFT is shown Davis and Nosratinia [11]

This of course does not mean that the DCT does not have its limitations and drawbacks. One of its major flaws is that even without the quantization, by simply transforming and inverting the transform of an image using the DCT on block sizes smaller than the image will result in data loss. This was however proven to be insignificant for a block size of 8, as shown by Pennebaker [35]. Another aspect worth noting that when using block sizes smaller than that of the dimension of the image one gets a blocking effect. This comes from the lack of overlap between the neighboring blocks basis functions. What this means that as one compresses more of the data, the block sizes used are more visible because of the sharp edges between blocks due to the lack of overlapping basis functions. This is something that we will later discuss when looking at the solution, mainly the lapped orthogonal transform LOT.

A.2 Fast DCT: The AAN Algorithm

The AAN algorithm is a 8-point 1D DCT algorithm that relies on the fact that one can obtain a N element DCT from 2N element DFT. The definition of the DFT for N real numbers is

$$F(u) = \sum_{x=0}^{k-1} s(x)e^{-\frac{j2\pi ux}{k}} \quad (\text{A.4})$$

To extend this definition to a 2N real number definition one can extend $s(x)$ symmetrically about $(2N - 1)/2$ with the relation of $s(x) = s(2N - x - 1)$ for x values from N to $2N - 1$. It is proven by Tseng et.al in [45] that if $F(u)$ is a 2N element DFT of $s(x)$, then for we have the equation

$$\frac{\Re(F(u))}{2 \cos(\frac{\pi u}{2N})} = \sum_{x=0}^{N-1} s(x) \cos\left(\frac{(2x + 1)u\pi}{2N}\right) \quad (\text{A.5})$$

Looking at the Equation A.5, we see that the right hand side is similar to the the definition of the DCT in Equation A.1, but missing the adjusting constant that is multiplied by it. One can then argue that by using the real values of the extended DFT and scaling them one is able to calculate DCT values, which is expressed in Equation A.1. This will of course result in 2N DFT values that need to be calculated, but since the values are symmetrical and we only need to use the real values much of the computations can be avoided as shown by Winograd in [49]. This method of calculating the DCT is known as the AAN algorithm and is one of the quickest methods, and is used in the JPEG libraries such as the one developed by the independent JPEG library.

In Figure A.1 one can find a flow graph showing the operations that are to be performed on a 8 point AAN DCT. The flow graph can be interpreted in the manner that black dots are an add function, the arrows are negations and boxes are multiplication functions. Other legends like the line and white circle are just a way to simplify tracing the values needed in the calculations. By following the graph from left to right one would be calculating the DCT and by following it from right to left one would be calculating the IDCT. This is because of the orthogonal nature of the transform.

An aspect worth noting is that the AAN algorithm only uses 13 multiplications of two types. One being the scaling and normalization factors of s_i , and the other are necessary to compute the DCT and are expressed as a_x . The values

A.2. FAST DCT: THE AAN ALGORITHM

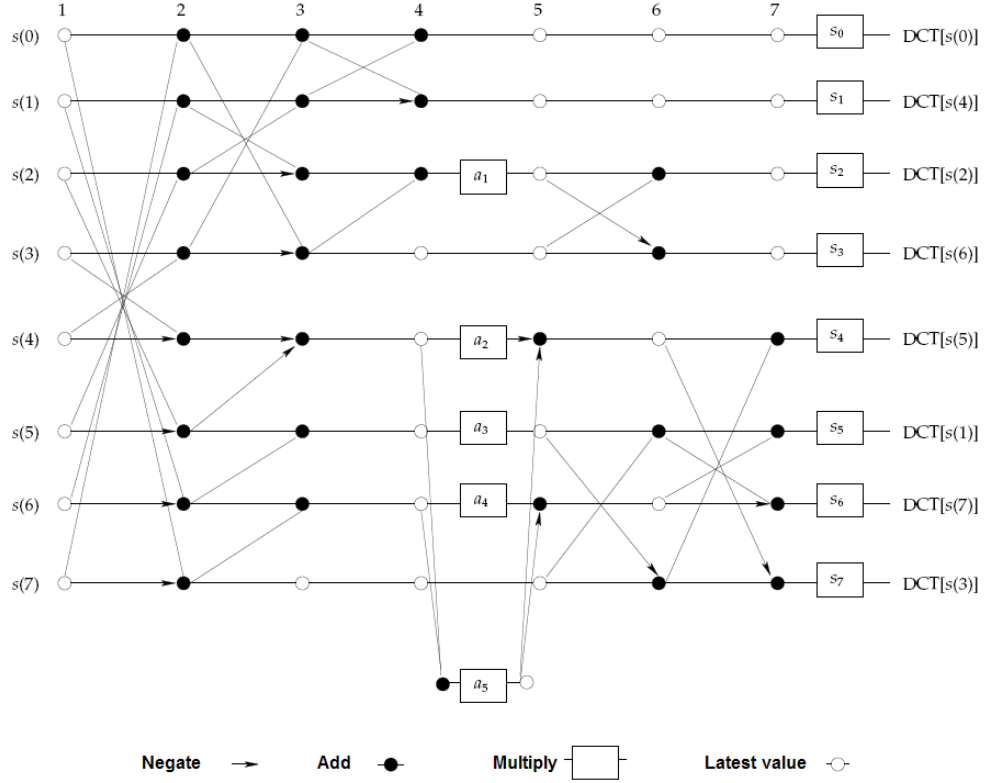


Figure A.1: Flowgraph of the AAN algorithm based on Pennebaker [35] and Arai et.al. [5] design, and obtained from [27], legends were added to simplify understanding

for these are constant and are shown here (these definitions are obtained from [35]):

$$a_1 = \frac{\sqrt{2}}{2} \quad (\text{A.6})$$

$$a_2 = \sqrt{2} \cos\left(\frac{3\pi}{8}\right) \quad (\text{A.7})$$

$$a_3 = a_1 = \frac{\sqrt{2}}{2} \quad (\text{A.8})$$

$$a_4 = \sqrt{2} \cos\left(\frac{\pi}{8}\right) \quad (\text{A.9})$$

$$a_5 = \cos\left(\frac{3\pi}{8}\right) \quad (\text{A.10})$$

The s_i values are used to convert the DFT coefficients to DCT coefficients and are calculated as follows.

$$s_0 = \frac{\sqrt{2}}{4} \quad (\text{A.11})$$

$$s_i = \frac{1}{4 \cos\left(\frac{i\pi}{2N}\right)} \quad (\text{A.12})$$

For further information regarding the AAN and the scaling factors, read Pennebaker [35] and/or Arai et.al [5] work.

A.3 Lapped Orthogonal Transform

The definition of the LOT is quite simple since it is a basic matrix multiplication, see Equation A.13, where T is the transformation matrix and x is the input data. T is here a $MN \times NM$ matrix, where N is the number of elements in a block and M is the number of blocks. The transformation matrix then is consisted of smaller matrices that reflect the even and odd values of the DCT basis functions of the given block. This is done in a way such that each block in the input data is multiplied with a P_x matrix. Here the matrix P_0 is the one used for all blocks except the ones on the edges because they have fewer neighboring blocks.

$$y = Tx \quad (\text{A.13})$$

$$P_0 = PZ \quad (\text{A.14})$$

where

$$P = \begin{bmatrix} D_e - D_o & D_e - D_o \\ J(D_e - D_o) & J(D_o - D_e) \end{bmatrix} \quad (\text{A.15})$$

A.3. LAPPED ORTHOGONAL TRANSFORM

and

$$Z = \begin{bmatrix} I & 0 \\ 0 & \tilde{Z} \end{bmatrix} \quad (\text{A.16})$$

Here, I and J are respectively identity and counter identity matrices. Whereas D_e and D_o are matrices containing even and odd DCT basis functions of the current block. \tilde{Z} is a matrix approximated by $N/2-1$ sequence of rotations R_i . Where each rotation contains a plain rotation $Y(\theta_i)$. These are defined as follows:

$$\tilde{Z} = R_1 R_2 \dots R_{N/2-1} \quad (\text{A.17})$$

and

$$Y(\theta_i) = \begin{bmatrix} \cos(\theta_i) & \sin(\theta_i) \\ -\sin(\theta_i) & \cos(\theta_i) \end{bmatrix} \quad (\text{A.18})$$

For a LOT of Block size 8, there are three optimal angles used in the rotation matrix, as shown in [29], and are:

$$[\theta_1 \ \theta_2 \ \theta_3] = [0.13\pi \ 0.16\pi \ 0.13\pi] \quad (\text{A.19})$$

One the advantages that the LOT has is that it is separable just like the DCT meaning that it works well for k-dimensions. By just calculating the input one dimension at a time. Other advantages, as mentioned earlier, it helps in removing the blocking effect and this in return can give more compression and less error in the reconstructed image after performing the inverse operation. Another aspect worth noting is that since the transform is orthogonal it is easy to perform the inverse operation. The disadvantage this has is that it is computationally more demanding than the DCT and therefore slower.

Now we will discuss the algorithm discussed by Malvar [29] also known as the Type I fast LOT. In this algorithm one computes first the DCT and then uses two blocks of size 8 from the DCT to calculate a block of the LOT transform. In Figure A.2, we show a flowgraph of the algorithm where the multiplication with P and Z are shown to show the connection to the definition discussed earlier, and how the two blocks of DCT are used to calculate the LOT block.

APPENDIX A. ORTHOGONAL TRANSFORM THEORY

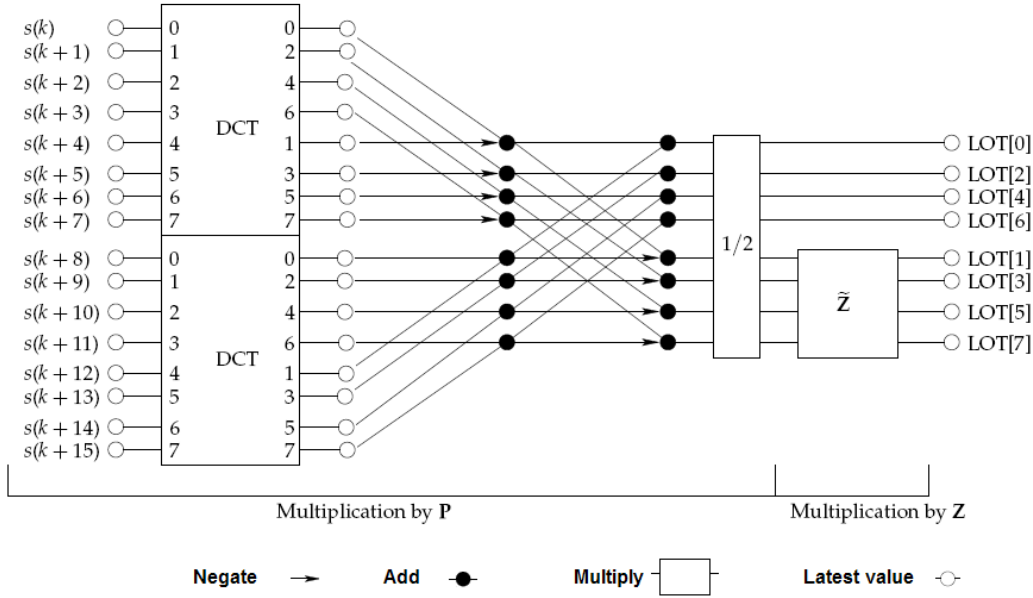


Figure A.2: Flowgraph of the LOT algorithm based on Malvar and Staelin [30] design, and obtained with permission from [27], legends were added to simplify understanding

Figures A.3 and A.4 show how the calculations for the Z and Y rotation matrices are performed. While Figure A.5 illustrates how the whole process works, and there are some aspects worth noting. The first DCT data to be transformed using the scheme has to be padded. The padding, a block of zeros, is added as the first and last block of the DCT data. The reason for the padding is because one needs two blocks of DCT data to produce a block of LOT data. This also can cause a problem for compression because the LOT output is one block larger than the original DCT data as seen in Figure A.5. Results in Malvar’s article and book [30] [29] show that the blocking effect is solved by using this method. But, for compression, it is worth noting that this is not necessarily the case for all images, but the blocking effect is always reduced. Another aspect worth noting is that the LOT is an orthogonal transform such as the DCT, and can be decoded by simply tracing the flow graphs in reverse. Making both the forward and backward transform equally efficient.

A.3. LAPPED ORTHOGONAL TRANSFORM

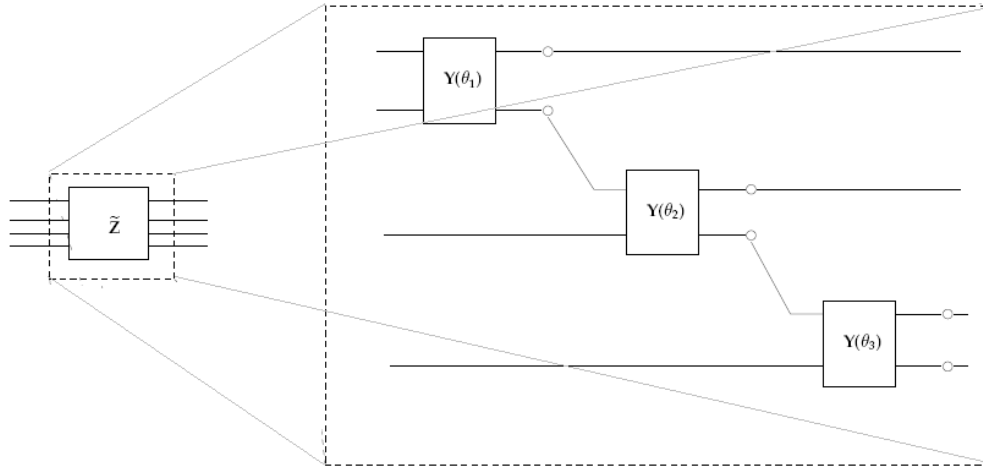


Figure A.3: Flowgraph showing the Z matrix based on Malvar and Staelin [30] design, and obtained with permission from [27], see legends from Figure A.2 to simplify understanding

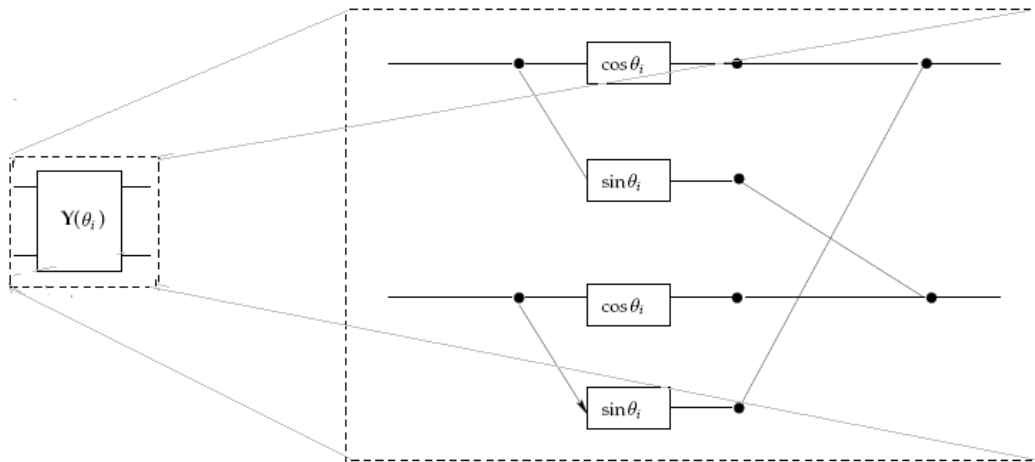


Figure A.4: Flowgraph showing the Y rotation matrix based on Malvar and Staelin [30] design, and obtained with permission from [27], see legends from Figure A.2 to simplify understanding

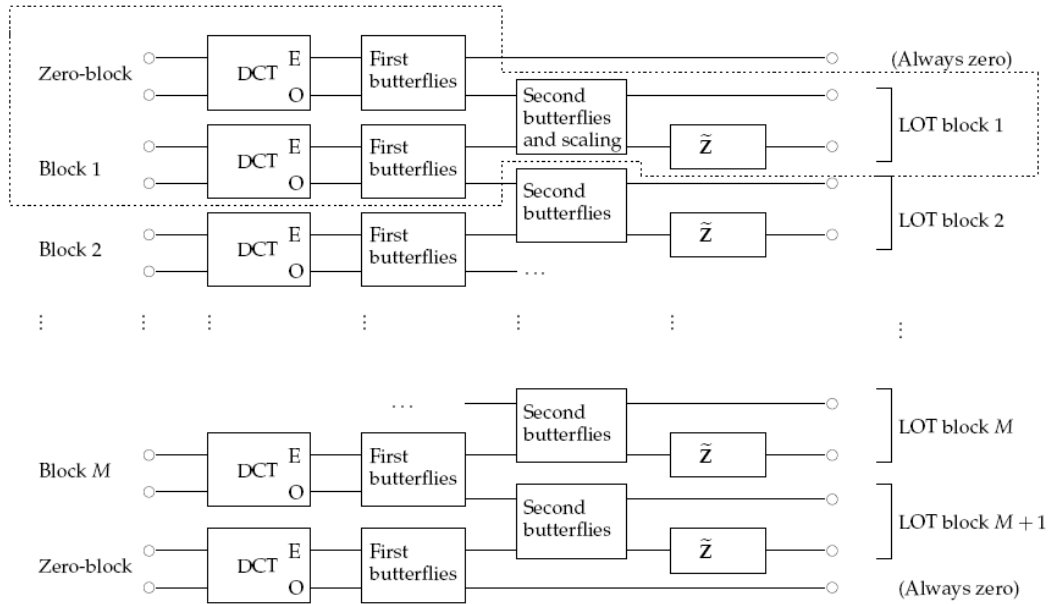


Figure A.5: Flowgraph showing the larger process of the transform based on Malvar and Staelin [30] design, and obtained with permission from [27], see legends from Figure A.2 to simplify understanding

A.3. LAPPED ORTHOGONAL TRANSFORM

APPENDIX B

Annotated Bibliography

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 volume data compression, seismic data compression, lossless and lossy data compression, and various transforms. All these topics are relevant to this thesis, and have been researched to lay a foundation for the implementations performed.

Below is an annotated bibliography of significant prior work done in adjacent fields of this thesis.

Lossless volume data compression schemes. P. Komma *et al.* [24]

This is a recent paper published at the Tagung conference, in Germany, on simulation and visualization. Here Komma *et al.* [24] describe different schemes and methods for lossless volume data compression. They categorize these in six different categories and test algorithms from each category to compare how effective these methods can be. The data used here is primarily medical CT, MRI and ultra sound scan data. This is relevant for this thesis because parallels can be seen between seismic and medical data, and since they are both volume data these algorithms and methods are applicable. A difference between medical and seismic data is that medical data has to be lossless whereas seismic data can be lossy to a certain degree. Nevertheless if a lossless alternative is fast and can compress data well then it should be of interest. A closer look at the categories considered will be presented in later chapters in this chapter.

Lossless compression of volume data. J. E. Fowler and R. Yagelt. [18]

This is a paper from IEEE transactions on image processing, which focuses on entropy encoding as a category and explains in detail the different steps one has to take to achieve compression of volume data using known entropy encoding standards such as zip, gzip, compress, and others. Fowler and Yagelt also introduce a custom compression algorithm, which makes use of Huffman encoding, which is interesting for this thesis since it shows the potential of using existing techniques in a new way. In contrast to this thesis parallelism is not addressed and not to mention the use of accelerators such as the GPU to improve runtime.

**Fast Lossless Compression of Scientific Floating-Point Data. P. Ratana-
worabhan *et al.* [37]**

This is a paper from IEEE Computer society, that discusses a method within arithmetic encoding to compress the floating point data type. This as well is a lossless compression method that seems to work well given a good prediction algorithm. The method focuses on predicting the next float and then by performing an XOR operation between the original and predicted float one will get a float that contains many zeros in a sequence. Then by using run length encoding or huffman one can compress each number to a much smaller bit size. This can prove to be very efficient if one is able to predict the correct float, and being lossless it is very interesting to see how well it is able to compress the data. The downside with this algorithm is that it builds upon other compression methods making the execution time longer. This might not be optimal for our purposes.

**Fast Lossless Compression of Seismic Floating-Point Data. X. Xie
and Q. Qin. [50]**

This is also a paper from IEEE Computer society that builds upon the findings of [37]. Here they discuss using the arithmetic encoding on seismic data with a combination of both huffman and run length encoding. They introduced a prediction algorithm that works well for seismic data and was able to beat other lossless methods in compression ratio, but has the flaw of being slower. The compression ratio they achieved was 1.77, which is quite good for lossless compression. This is interesting for this thesis because it is directly related to

compressing seismic data, and the results show that it is a better method than conventional lossless compression.

Discrete cosine transform, N. Ahmed *et al.* [2]

This is a heavily cited paper published in 1974 in IEEE transactions on computers. Here Ahmed *et al.* explain the discrete cosine transform and its use in the area of digital processing for the purpose of pattern recognition. Here they introduce a fast algorithm for the transform and compare it to other transforms used in similar applications such as the Haar transform, slant transform, discrete Fourier transform, and others. This is a useful paper because the discrete cosine transform is used in data compression and this paper explains it and its nature in image processing.

Image compression using the discrete cosine transform. A. B. Watson. [46]

This is a paper written for the journal *mathematica*, which explains how the cosine transform can be used in image compression. First it speaks generally about orthogonal transforms and compression, then it goes in depth in the subject of using the discrete cosine transform (DCT) for compression. This is a useful paper in that it shows in detail how the transform can compress images and how the pattern recognition discussed in Ahmed *et al.* [2] is useful. Algorithms and run-times are also discussed here. This is good to use as a foundation for orthogonal transformations and their use in compression, which is very relevant for this thesis. The limitations here are that the DCT and orthogonal transforms are lossy methods in compression.

A Fast DCT-SQ Scheme for Images. Y. Arai *et al.* [5]

This is a paper from transactions of the institute of electronics, information and communication engineers, which is really popular in the field of compression and image processing. It is about a fast DCT algorithm, known as the AAN algorithm. Arai *et al.* are able to calculate the DCT without having to go through all the naïve computations and matrix multiplications. Rather they reuse calculated values and scale them to match the results of the DCT. This paper is originally written in Japanese, but has been used in many image and sound standards including JPEG. That is why a reproduction of their work in

a book by Pennebaker and Mitchell [35] is where the algorithm is retrieved for this thesis. This is important for this thesis because their fast implementation can be useful when expanding lossy compression in several dimensions.

The LOT: Transform Coding Without Blocking Effect. H. S. Malvar and D. H. Staelin. [30]

This paper is from IEEE transactions on acoustics, speech, and signal Processing, and approaches a common problem when filtering and compressing using the DCT. One of the major issues with using the DCT is that as files are compressed more and more they start getting blocking effects. This means as more details are filtered away, depending on the block size used to perform the DCT, one will be able to see the edging and blocking effects. In this paper they discuss an approach that builds upon the DCT and is able to filter images to the same extent, but avoids blocking effects. This is done by expanding the basis functions of the DCT such that they overlap. This method is interesting for this thesis since it means that by compressing as much with this method as the DCT one is able to get a lower error term, and the end result after decompression is more correct and closer to the original image. This comes of course with the price of more calculations, but there are fast LOT algorithms that we will consider.

The GenLOT: Generalized Linear-Phase Lapped Orthogonal Transform. R. L. de Queiroz [12]

This is a paper from IEEE transactions on image processing that approaches the idea of having a generalized lapped orthogonal transform. Here they show that one can perform a generalized LOT on several levels or degrees. Just as the LOT builds upon DCT, the GenLOT in the same manner builds upon it but now takes into consideration more than just neighboring blocks depending on the degree one chooses. The GenLOT of the 1st degree is basically a standard DCT, while in the second degree it is the LOT and so on. For degrees of 2 or higher the GenLOT gives even better results than the LOT when it comes to image quality, and better compression rates. This is useful for this thesis because it means that one can experiment to which degree the GenLOT can still give little error and still be able to compress more than the other methods. It is of course very computationally demanding since it performs more calculation than both DCT and LOT in higher degrees, which would imply that running it on the GPU should be interesting since it is very parallelizable.

GenLOT Optimization Techniques for Seismic Data Compression.
L. C. Duval *et al.* [14]

This paper is also from IEEE transactions on image processing, and builds upon the paper by Queiroz *et al.* [12]. The focus here is to apply the GenLOT on seismic data and find an optimal scheme for compression with little loss in data. This will prove to be useful since their result can be used to optimize compression in our GenLOT implementation, but their can also be a tradeoff of execution speed, which they do not account for. This is really interesting for us since we intend not only to compress the data as much as possible, but to do so with a fast execution time or at least find an optimal balance between compression ratio and execution time.

APPENDIX C

Benchmarking Tables

Below is a list of the Benchmark tables that we have included in the order they are presented

- Run Length Encoding (RLE)
- Huffman encoding
- 1D naive DCT
- 2D naive DCT
- 1D AAN DCT
- 2D AAN DCT
- 3D AAN DCT
- Lapped orthogonal transform (LOT)
- Filtering Hough transform and 3D convolution
- Synchronous I/O
- Asynchronous I/O

RLE Single Core

Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	I/O Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	4.2	4.2	185.16	1.32	1.08	3.51
500x500x500	476.84	14.3	14.8	8.3	8.3	362	1.32	0.75	3.24
600x600x600	823.98	24.3	24.3	14.5	14.6	626.92	1.31	0.73	3.17
700x700x700	1308.44	38.5	39.1	23.4	23.2	993.71	1.32	0.73	3.13
800x800x800	1953.13	57.5	57.7	34.8	34.7	1489.19	1.31	0.73	3.06
800x1000x1000	3051.76	91.2	92.4	55.6	54.8	2337.49	1.31	0.73	3.05
800x2000x2000	12207.03	351.9	352.1	210.5	209.8	9395.54	1.3	0.73	3.05

RLE Quad Core

Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	I/O Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	1.2	1.2	185.16	1.32	1.08	3.51
500x500x500	476.84	14.3	14.8	2.47	2.56	362	1.32	1.07	3.24
600x600x600	823.98	24.3	24.3	4.59	4.61	626.92	1.31	1.05	3.17
700x700x700	1308.44	38.5	39.1	7.32	7.37	993.71	1.32	1.05	3.15
800x800x800	1953.13	57.5	57.7	11.2	11.1	1489.19	1.31	1.05	3.13
800x1000x1000	3051.76	91.2	92.4	17.98	17.92	2337.49	1.31	1.04	3.06
800x2000x2000	12207.03	351.9	352.1	68.2	68.7	9395.54	1.3	1.04	3.05

APPENDIX C. BENCHMARKING TABLES

Huffman Single Core										
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	2.46	2.46	174.57	1.4	1.4	1.4	0.94
500x500x500	476.84	14.3	14.8	4.81	4.81	340.47	1.4	1.4	1.4	0.95
600x600x600	823.98	24.3	24.3	8.31	8.31	586.47	1.4	1.4	1.4	0.95
700x700x700	1308.44	38.5	39.1	13.1	13.1	928.81	1.41	1.41	1.41	0.95
800x800x800	1953.13	57.5	57.7	19.7	19.7	1386.84	1.41	1.41	1.41	0.95
800x1000x1000	3051.76	91.2	92.4	30.6	30.6	2160.01	1.41	1.41	1.41	0.96
800x2000x2000	12207.03	351.9	352.1	122.4	122.4	8648.83	1.41	1.41	1.41	0.95
Huffman Quad Core										
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	0.82	0.81	174.57	1.4	1.4	1.4	1.21
500x500x500	476.84	14.3	14.8	1.57	1.57	340.47	1.4	1.4	1.4	1.21
600x600x600	823.98	24.3	24.3	2.76	2.76	586.47	1.4	1.4	1.4	1.21
700x700x700	1308.44	38.5	39.1	4.35	4.31	928.81	1.41	1.41	1.41	1.22
800x800x800	1953.13	57.5	57.7	6.48	6.45	1386.84	1.41	1.41	1.41	1.22
800x1000x1000	3051.76	91.2	92.4	9.98	9.91	2160.01	1.41	1.41	1.41	1.22
800x2000x2000	12207.03	351.9	352.1	40.5	39.8	8648.83	1.41	1.41	1.41	1.21
Huffman GPU										
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	2.01	2.1	174.57	1.4	1.4	1.4	0.99
500x500x500	476.84	14.3	14.8	1.57	4.01	340.47	1.4	1.4	1.4	1.01
600x600x600	823.98	24.3	24.3	2.76	6.81	586.47	1.4	1.4	1.4	1.01
700x700x700	1308.44	38.5	39.1	4.35	10.74	928.81	1.41	1.41	1.41	1.01
800x800x800	1953.13	57.5	57.7	6.48	16.42	1386.84	1.41	1.41	1.41	1.01
800x1000x1000	3051.76	91.2	92.4	9.98	25.29	2160.01	1.41	1.41	1.41	1.02
800x2000x2000	12207.03	351.9	352.1	40.5	101.16	8648.83	1.41	1.41	1.41	1.21

DCT naïve single Core											
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	CPU Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	7.88	7.98	157.56	1.55	1.55	1.55	1.37	13.53
500x500x500	476.84	14.3	14.8	15.6	15.6	307.84	1.55	1.55	1.55	1.38	13.45
600x600x600	823.98	24.3	24.3	26.9	26.8	534.3	1.54	1.54	1.54	1.37	13.54
700x700x700	1308.44	38.5	39.1	42.8	42.7	848.26	1.54	1.54	1.54	1.37	13.56
800x800x800	1953.13	57.5	57.7	63.7	63.8	1268.95	1.54	1.54	1.54	1.37	13.55
800x1000x1000	3051.76	91.2	92.4	99.5	99.5	1986.55	1.54	1.54	1.54	1.37	13.45
800x2000x2000	12207.03	351.9	352.1	397.5	398.2	7955.24	1.53	1.53	1.53	1.36	13.41
DCT naïve quad Core											
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	CPU Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	2.1	2.1	157.56	1.55	1.55	1.55	1.06	3.8
500x500x500	476.84	14.3	14.8	4.1	4.1	307.84	1.55	1.55	1.55	1.07	3.8
600x600x600	823.98	24.3	24.3	7.2	7.1	534.3	1.54	1.54	1.54	1.06	3.77
700x700x700	1308.44	38.5	39.1	11.2	11.3	848.26	1.54	1.54	1.54	1.06	3.78
800x800x800	1953.13	57.5	57.7	16.8	16.9	1268.95	1.54	1.54	1.54	1.06	3.78
800x1000x1000	3051.76	91.2	92.4	26.3	26.4	1986.55	1.54	1.54	1.54	1.06	3.77
800x2000x2000	12207.03	351.9	352.1	107.4	107.5	7955.24	1.53	1.53	1.53	1.05	3.7
DCT naïve GPU											
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	CPU Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	0.57	0.59	157.56	1.55	1.55	1.55	1.37	13.53
500x500x500	476.84	14.3	14.8	1.18	1.16	307.84	1.55	1.55	1.55	1.38	13.45
600x600x600	823.98	24.3	24.3	1.97	1.98	534.3	1.54	1.54	1.54	1.37	13.54
700x700x700	1308.44	38.5	39.1	3.13	3.15	848.26	1.54	1.54	1.54	1.37	13.56
800x800x800	1953.13	57.5	57.7	4.72	4.71	1268.95	1.54	1.54	1.54	1.37	13.55
800x1000x1000	3051.76	91.2	92.4	7.5	7.4	1986.55	1.54	1.54	1.54	1.37	13.45
800x2000x2000	12207.03	351.9	352.1	29.9	29.7	7955.24	1.53	1.53	1.53	1.36	13.41

APPENDIX C. BENCHMARKING TABLES

DC T2D naïve single Core														
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	CPU Speedup
400x400x400	244,14	7,1	7,2	12,2	12,1	79,76	3,06	3,06	0,49					
500x500x500	476,84	14,3	14,8	23,5	23,4	155,76	3,06	3,06	0,51					
600x600x600	823,98	24,3	24,3	40,7	40,8	270,32	3,05	3,05	0,5					
700x700x700	1308,44	38,5	39,1	65,1	65,3	429,03	3,05	3,05	0,49					
800x800x800	1953,13	57,5	57,7	97,7	97,5	641,84	3,04	3,04	0,49					
800x1000x1000	3051,76	91,2	92,4	153,4	152,3	1004,5	3,04	3,04	0,5					
800x2000x2000	12207,03	351,9	352,1	595,2	596,4	4018,15	3,04	3,04	0,49					
DC T2D naïve quad Core														
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	CPU Speedup
400x400x400	244,14	7,1	7,2	3,2	3,12	79,76	3,06	3,06	1,3					3,87
500x500x500	476,84	14,3	14,8	6,2	6,1	155,76	3,06	3,06	1,33					3,84
600x600x600	823,98	24,3	24,3	10,8	10,7	270,32	3,05	3,05	1,3					3,81
700x700x700	1308,44	38,5	39,1	16,9	17,1	429,03	3,05	3,05	1,3					3,82
800x800x800	1953,13	57,5	57,7	25,7	25,6	641,84	3,04	3,04	1,29					3,81
800x1000x1000	3051,76	91,2	92,4	39,9	39,8	1004,5	3,04	3,04	1,31					3,83
800x2000x2000	12207,03	351,9	352,1	158,1	157,8	4018,15	3,04	3,04	1,29					3,78
DC T2D naïve GPU														
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	CPU Speedup
400x400x400	244,14	7,1	7,2	0,68	0,66	79,76	3,06	3,06	2,38					18,24
500x500x500	476,84	14,3	14,8	1,28	1,29	155,76	3,06	3,06	2,4					18,14
600x600x600	823,98	24,3	24,3	2,23	2,18	270,32	3,05	3,05	2,39					18,72
700x700x700	1308,44	38,5	39,1	3,38	3,39	429,03	3,05	3,05	2,4					19,26
800x800x800	1953,13	57,5	57,7	5,05	5,1	641,84	3,04	3,04	2,4					19,12
800x1000x1000	3051,76	91,2	92,4	7,7	7,8	1004,5	3,04	3,04	2,41					19,53
800x2000x2000	12207,03	351,9	352,1	30,5	30,2	4018,15	3,04	3,04	2,41					19,75

DCT AAN single Core										
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup		
400x400x400	244.14	7.1	7.2	0.98	0.99	157.56	1.55	1.27		
500x500x500	476.84	14.3	14.8	1.95	1.98	307.84	1.55	1.28		
600x600x600	823.98	24.3	24.3	3.3	3.38	534.3	1.54	1.27		
700x700x700	1308.44	38.5	39.1	5.3	5.37	848.26	1.54	1.27		
800x800x800	1953.13	57.5	57.7	7.9	8	1268.95	1.54	1.27		
800x1000x1000	3051.76	91.2	92.4	12.37	12.5	1986.55	1.54	1.27		
800x2000x2000	12207.03	351.9	352.1	49.8	50	7955.24	1.53	1.26		
DCT AAN quad Core										
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	CPU Speedup	
400x400x400	244.14	7.1	7.2	0.39	0.4	157.56	1.55	1.43	2.48	
500x500x500	476.84	14.3	14.8	0.6	0.59	307.84	1.55	1.46	3.36	
600x600x600	823.98	24.3	24.3	1	1.03	534.3	1.54	1.45	3.28	
700x700x700	1308.44	38.5	39.1	1.6	1.6	848.26	1.54	1.45	3.36	
800x800x800	1953.13	57.5	57.7	2.49	2.45	1268.95	1.54	1.44	3.27	
800x1000x1000	3051.76	91.2	92.4	3.78	3.76	1986.55	1.54	1.44	3.32	
800x2000x2000	12207.03	351.9	352.1	15.53	15.58	7955.24	1.53	1.44	3.21	
DCT AAN GPU										
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	CPU Speedup	
400x400x400	244.14	7.1	7.2	0.4	0.4	157.56	1.55	1.43	2.48	
500x500x500	476.84	14.3	14.8	0.74	0.75	307.84	1.55	1.43	2.64	
600x600x600	823.98	24.3	24.3	1.32	1.3	534.3	1.54	1.42	2.6	
700x700x700	1308.44	38.5	39.1	1.98	2.01	848.26	1.54	1.43	2.67	
800x800x800	1953.13	57.5	57.7	2.87	2.91	1268.95	1.54	1.43	2.75	
800x1000x1000	3051.76	91.2	92.4	4.45	4.49	1986.55	1.54	1.43	2.78	
800x2000x2000	12207.03	351.9	352.1	17.6	17.8	7955.24	1.53	1.43	2.81	

APPENDIX C. BENCHMARKING TABLES

DCT 2D AAN single Core										
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	2.14	2.15	82.76	2.95	2.95	1.56	
500x500x500	476.84	14.3	14.8	4.17	4.18	162.46	2.94	2.94	1.58	
600x600x600	823.98	24.3	24.3	7.23	7.25	280.51	2.94	2.94	1.57	
700x700x700	1308.44	38.5	39.1	11.3	11.4	446.82	2.93	2.93	1.57	
800x800x800	1953.13	57.5	57.7	16.9	17.1	666	2.93	2.93	1.57	
800x1000x1000	3051.76	91.2	92.4	26.7	26.8	1049.02	2.91	2.91	1.57	
800x2000x2000	12207.03	351.9	352.1	107.3	107.2	4196.08	2.91	2.91	1.54	
DCT 2D AAN quad Core										
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	0.67	0.66	82.76	2.95	2.95	2.32	3.26
500x500x500	476.84	14.3	14.8	1.3	1.29	162.46	2.94	2.94	2.32	3.24
600x600x600	823.98	24.3	24.3	2.19	2.23	280.51	2.94	2.94	2.31	3.25
700x700x700	1308.44	38.5	39.1	3.5	3.5	446.82	2.93	2.93	2.31	3.26
800x800x800	1953.13	57.5	57.7	5.2	5.21	666	2.93	2.93	2.32	3.28
800x1000x1000	3051.76	91.2	92.4	8.22	8.22	1049.02	2.91	2.91	2.3	3.26
800x2000x2000	12207.03	351.9	352.1	32.8	32.9	4196.08	2.91	2.91	2.29	3.26
DCT 2D AAN GPU										
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	0.41	0.42	82.76	2.95	2.95	2.51	5.12
500x500x500	476.84	14.3	14.8	0.8	0.81	162.46	2.94	2.94	2.52	5.16
600x600x600	823.98	24.3	24.3	1.39	1.4	280.51	2.94	2.94	2.51	5.18
700x700x700	1308.44	38.5	39.1	2.23	2.13	446.82	2.93	2.93	2.52	5.35
800x800x800	1953.13	57.5	57.7	3.23	3.19	666	2.93	2.93	2.52	5.36
800x1000x1000	3051.76	91.2	92.4	5.1	5.01	1049.02	2.91	2.91	2.51	5.35
800x2000x2000	12207.03	351.9	352.1	20.1	19.9	4196.08	2.91	2.91	2.49	5.39

DCT 3D AAN single Core												
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	GPU Speedup	GPU Speedup	
400x400x400	244.14	7.1	7.2	3.27	3.27	36.06	6.77	6.77	4.74	7.27		
500x500x500	476.84	14.3	14.8	6.48	6.5	69.35	6.88	6.88	4.82	7.3		
600x600x600	823.98	24.3	24.3	11	11.03	121.25	6.8	6.8	4.82	7.5		
700x700x700	1308.44	38.5	39.1	17.5	17.6	190.92	6.85	6.85	4.8	7.33		
800x800x800	1953.13	57.5	57.7	26.1	26.2	289.92	6.74	6.74	4.82	7.71		
800x1000x1000	3051.76	91.2	92.4	41.1	41.1	453.01	6.74	6.74	4.82	7.61		
800x2000x2000	12207.03	351.9	352.1	165.3	165.2	1853.94	6.58	6.58	4.73	7.9		

DCT 3D AAN quad Core												
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	GPU Speedup	GPU Speedup	
400x400x400	244.14	7.1	7.2	1.08	1.05	36.06	6.77	6.77	3.38	3.11		
500x500x500	476.84	14.3	14.8	2.1	2.1	69.35	6.88	6.88	3.42	3.1		
600x600x600	823.98	24.3	24.3	3.48	3.57	121.25	6.8	6.8	3.4	3.09		
700x700x700	1308.44	38.5	39.1	5.77	5.71	190.92	6.85	6.85	3.4	3.08		
800x800x800	1953.13	57.5	57.7	8.45	8.42	289.92	6.74	6.74	3.39	3.11		
800x1000x1000	3051.76	91.2	92.4	13.21	13.16	453.01	6.74	6.74	3.42	3.12		
800x2000x2000	12207.03	351.9	352.1	50.7	50.8	1853.94	6.58	6.58	3.38	3.25		

DCT 3D AAN GPU												
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	GPU Speedup	GPU Speedup	
400x400x400	244.14	7.1	7.2	0.45	0.45	36.06	6.77	6.77	4.74	7.27		
500x500x500	476.84	14.3	14.8	0.91	0.89	69.35	6.88	6.88	4.82	7.3		
600x600x600	823.98	24.3	24.3	1.5	1.47	121.25	6.8	6.8	4.82	7.5		
700x700x700	1308.44	38.5	39.1	2.4	2.4	190.92	6.85	6.85	4.8	7.33		
800x800x800	1953.13	57.5	57.7	3.4	3.4	289.92	6.74	6.74	4.82	7.71		
800x1000x1000	3051.76	91.2	92.4	5.5	5.4	453.01	6.74	6.74	4.82	7.61		
800x2000x2000	12207.03	351.9	352.1	21.1	20.9	1853.94	6.58	6.58	4.73	7.9		

APPENDIX C. BENCHMARKING TABLES

DCT AAN w/LOT single Core										
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	2.1	2.1	160.78	1.52	1.52	1.05	
500x500x500	476.84	14.3	14.8	4.01	3.98	307.84	1.55	1.55	1.08	
600x600x600	823.98	24.3	24.3	6.98	7.08	534.3	1.54	1.54	1.06	
700x700x700	1308.44	38.5	39.1	11.2	11.1	848.26	1.54	1.54	1.07	
800x800x800	1953.13	57.5	57.7	16.8	16.9	1268.95	1.54	1.54	1.06	
800x1000x1000	3051.76	91.2	92.4	26.9	26.8	1986.55	1.54	1.54	1.06	
800x2000x2000	12207.03	351.9	352.1	105.1	104.9	7955.24	1.53	1.53	1.05	
DCT AAN w/LOT quad Core										
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	0.79	0.78	157.56	1.55	1.55	1.32	2.69
500x500x500	476.84	14.3	14.8	1.5	1.5	307.84	1.55	1.55	1.33	2.65
600x600x600	823.98	24.3	24.3	2.7	2.7	534.3	1.54	1.54	1.32	2.62
700x700x700	1308.44	38.5	39.1	4.1	4.1	848.26	1.54	1.54	1.32	2.71
800x800x800	1953.13	57.5	57.7	6.22	6.22	1268.95	1.54	1.54	1.32	2.72
800x1000x1000	3051.76	91.2	92.4	9.89	9.89	1986.55	1.54	1.54	1.32	2.71
800x2000x2000	12207.03	351.9	352.1	39.3	39.3	7955.24	1.53	1.53	1.31	2.67
DCT AAN w/LOT GPU										
Seismic cube dims	File size (MB)	Read	Write	Compression time	Decompression time	Compressed file size	Compression ratio	IO Speedup	IO Speedup	CPU Speedup
400x400x400	244.14	7.1	7.2	0.43	0.45	157.56	1.55	1.55	1.41	4.67
500x500x500	476.84	14.3	14.8	0.81	0.81	307.84	1.55	1.55	1.42	4.91
600x600x600	823.98	24.3	24.3	1.4	1.4	534.3	1.54	1.54	1.42	5.06
700x700x700	1308.44	38.5	39.1	2.13	2.13	848.26	1.54	1.54	1.42	5.21
800x800x800	1953.13	57.5	57.7	3.19	3.19	1268.95	1.54	1.54	1.42	5.3
800x1000x1000	3051.76	91.2	92.4	5.01	5.01	1986.55	1.54	1.54	1.42	5.35
800x2000x2000	12207.03	351.9	352.1	19.9	19.9	7955.24	1.53	1.53	1.41	5.27

Hough transform on each 10th pixel							
Seismic cube dims	Single	Quad	Quad Speedup	GPU	GPU Speedup	GPU/quad Speedup	
400x400x400	38.2	9.5	4.02	0.79	48.35	12.03	
500x500x500	93.1	23.2	4.01	1.8	51.72	12.89	
600x600x600	193	48	4.02	3.75	51.47	12.8	
700x700x700	357	90.5	3.94	6.9	51.74	13.12	
800x800x800	611	155	3.94	12.3	49.67	12.6	
800x1000x1000	1195	305	3.92	23.8	50.21	12.82	
800x2000x2000	9936	2547.5	3.9	201.1	49.41	12.67	
3D Convolution with filter size 13							
Seismic cube dims	Single	Quad	Quad Speedup	GPU	GPU Speedup	GPU/quad Speedup	
400x400x400	239	75.8	3.15	15.1	15.83	5.02	
500x500x500	490	150	3.27	31.2	15.71	4.81	
600x600x600	851	259	3.29	53.3	15.97	4.86	
700x700x700	1362	414	3.29	85.3	15.97	4.85	
800x800x800	2123.8	603	3.52	129.5	16.4	4.66	
800x1000x1000	3472	976	3.56	206.5	16.81	4.73	
800x2000x2000	13892	3886	3.57	821.8	16.9	4.73	

APPENDIX C. BENCHMARKING TABLES

Test On HDD1	Exec time	Speedup	Average disk rate MB/s
Normal IO time	351	1	34,76
RLE	339	1,04	35,99
Huffman	289,8	1,21	42,1
DCT <u>naïve</u> 1D	259	1,36	47,1
DCT <u>naïve</u> 2D	146	2,4	83,56
DCT AAN 1D	247	1,42	49,39
DCT AAN 2D	141	2,49	86,52
DCT AAN 3D	74	4,74	164,86
DCT AAN LOT 1D	249	1,41	49
Test On HDD2	Exec time	Speedup	Average disk rate MB/s
Normal IO time	176	1	69,32
RLE	203	0,87	60,1
Huffman	165	1,07	73,94
DCT <u>naïve</u> 1D	144,6	1,22	84,37
DCT <u>naïve</u> 2D	88	2	138,64
DCT AAN 1D	132,3	1,33	92,21
DCT AAN 2D	80,6	2,18	151,36
DCT AAN 3D	47,6	3,7	256,3
DCT AAN LOT 1D	134,6	1,31	90,64
Test On SSD	Exec time	Speedup	Average disk rate MB/s
Normal IO time	87	1	140,23
RLE	135,2	0,64	90,24
Huffman	102,1	0,85	119,49
DCT <u>naïve</u> 1D	86,6	1	140,88
DCT <u>naïve</u> 2D	58,8	1,48	207,48
DCT AAN 1D	74,3	1,17	164,2
DCT AAN 2D	50	1,74	244
DCT AAN 3D	34,11	2,55	357,67
DCT AAN LOT 1D	76,6	1,14	159,27

Test On HDD1	Exec time	Speedup	Average disk rate MB/s
Normal IO time	351	1	34,76
RLE	269	1,3	45,35
Huffman	247	1,42	49,39
DCT naïve 1D	227	1,55	53,74
DCT naïve 2D	115	3,05	106,09
DCT AAN 1D	227	1,55	53,74
DCT AAN 2D	114	3,08	107,02
DCT AAN 3D	65	5,4	187,69
DCT AAN LOT 1D	235	1,49	51,91
Test On HDD2	Exec time	Speedup	Average disk rate MB/s
Normal IO time	176	1	69,32
RLE	136	1,29	89,71
Huffman	125	1,41	97,6
DCT naïve 1D	115	1,53	106,09
DCT naïve 2D	58	3,03	210,34
DCT AAN 1D	115	1,53	106,09
DCT AAN 2D	58	3,03	210,34
DCT AAN 3D	29,5	5,97	413,56
DCT AAN LOT 1D	123	1,43	99,19
Test On SSD	Exec time	Speedup	Average disk rate MB/s
Normal IO time	87	1	140,23
RLE	70	1,24	174,29
Huffman	62	1,4	196,77
DCT naïve 1D	57	1,53	214,04
DCT naïve 2D	33	2,64	369,7
DCT AAN 1D	57	1,53	214,04
DCT AAN 2D	31	2,81	393,55
DCT AAN 3D	26	3,35	469,23
DCT AAN LOT 1D	62	1,4	196,77

APPENDIX D

Source Code

This chapter focuses on presenting the source code for the most important algorithms developed in this thesis.

D.1 RLE

```
1 void compressRLE(float* DCT_data, float*output_data, unsigned int*
   RLE_dic){
2
3   long long int counter = 1;
4   dic_addrss = 0;
5   bool foo = false;
6
7   float current_element = DCT_data[0];
8   float next_element = DCT_data[1];
9
10  out_addrss = 0;
11
12  for(unsigned int i = 2; i < block_size; i++){
13
14    if(current_element == 0 && current_element == next_element){
15      current_element = next_element;
16      next_element = DCT_data[i];
17      counter++;
18      foo = true;
19    }
20    else{
21      if(foo){
22        RLE_dic[dic_addrss] = out_addrss;
23        output_data[out_addrss] = counter;
24        out_addrss++;
```

D.1. RLE

```
25     dic_addrss++;
26     current_element = next_element;
27     next_element = DCT_data[i];
28     counter = 1;
29     foo = false;
30 }
31 else{
32     output_data[out_addrss] = current_element;
33     out_addrss++;
34     current_element = next_element;
35     next_element = DCT_data[i];
36     counter = 1;
37 }
38 }
39 }
40 if(foo){
41     RLE_dic[dic_addrss] = out_addrss;
42     output_data[out_addrss] = counter;
43 }
44 else{
45     output_data[out_addrss] = current_element;
46 }
47 }
```

```
1 void decompressRLE(float* input_data, float*output_data, unsigned
2   int*RLE_dic){
3     long long int first = 0;
4     long long int second = 0;
5     unsigned int k = 0;
6     unsigned int rle_value = 0;
7     while(RLE_dic[k] != 0 || k == 0){
8         rle_value = RLE_dic[k];
9         for(long long int i = first; i < rle_value; i++){
10            input_data[second] = output_data[i];
11            second++;
12            first++;
13        }
14        if(first == rle_value) first++;
15
16        for(int i = 0; i < output_data[rle_value]; i++){
17            input_data[second] = 0;
18            second++;
19        }
20
21        k++;
22    }
23 }
```

D.2 Huffman

```
1 void generateHuffTable(){
2
3     hf_codes = (unsigned int *) malloc(sizeof(int)*256);
4     hf_lengths = (unsigned int *) malloc(sizeof(int)*256);
5
6     unsigned int* hf_codes_temp = (unsigned int *) malloc(sizeof(int)
7         )*4);
8     unsigned int* hf_lengths_temp = (unsigned int *) malloc(sizeof(
9         int)*4);
10
11     hf_codes_temp[0] = 0; // 0.....0
12     hf_codes_temp[1] = 6<<(32-3); // 110...0
13     hf_codes_temp[2] = 2<<(32-2); // 10....0
14     hf_codes_temp[3] = 14<<(32-4); // 1110..0
15
16     hf_lengths_temp[0] = 1;
17     hf_lengths_temp[1] = 3;
18     hf_lengths_temp[2] = 2;
19     hf_lengths_temp[3] = 4;
20
21     unsigned char* tuples = (unsigned char *) malloc(sizeof(unsigned
22         char)*8);
23     tuples[0] = 192;
24     tuples[1] = 48;
25     tuples[2] = 12;
26     tuples[3] = 3;
27
28     unsigned int temp;
29     unsigned int current_int;
30     unsigned int bit_counter;
31     unsigned int length;
32     unsigned char tuple;
33     unsigned char mychar = 0;
34
35     #pragma omp parallel for
36     for(int i=0; i<256; i++){
37         current_int = 0;
38         bit_counter = 0;
39
40         for (int tuple_i = 0; tuple_i < 4; tuple_i++) {
41             tuple = mychar & tuples[tuple_i];
42             tuple = tuple >> (3-tuple_i)*2;
43             temp = hf_codes_temp[tuple];
44             length = hf_lengths_temp[tuple];
45             temp = temp >> bit_counter;
46             current_int |= temp;
47         }
48     }
49 }
```

D.2. HUFFMAN

```
44     bit_counter += length;
45     }
46     mychar++;
47
48     hf_codes[i] = current_int;
49     hf_lengths[i] = bit_counter;
50     }
51 }
```

```
1 void compressHuff(){
2
3     unsigned int bit_counter = 0;
4     unsigned int current_int = 0;
5     long long int int_counter = 0;
6     unsigned int hf_code;
7     unsigned int hf_length;
8     unsigned int rest_hf_code;
9     unsigned char tuple;
10    out_addrss = 0;
11    unsigned int * int_array;
12
13    #pragma omp parallel private(bit_counter, current_int, int_counter
14    , hf_code, hf_length, rest_hf_code, tuple, int_array) shared(
15    out_addrss) num_threads(4)
16    {
17        int threadid = omp_get_thread_num();
18        int num_threads = omp_get_num_threads();
19        int_counter = 0;
20        int_array = (unsigned int *) malloc(nbytes/num_threads);
21        //memset(int_array, 0, nbytes/num_threads);
22
23        for (int char_i = nbytes/num_threads*threadid; char_i < nbytes/
24            num_threads*(threadid+1); char_i++) {
25            tuple = input_data[char_i];
26            hf_code = hf_codes[tuple];
27            hf_length = hf_lengths[tuple];
28            // Shift
29            hf_code = hf_code >> bit_counter;
30            // Or
31            current_int |= hf_code;
32            // Update bit_counter
33            bit_counter += hf_length;
34
35            if (bit_counter > 32) {
36                // Shift left original hf_code into rest_hf_code
37                rest_hf_code = hf_codes[tuple] << hf_lengths[tuple]-(
38                    bit_counter-32);
39                int_array[int_counter] = current_int;
40                int_counter++;
41            }
42        }
43    }
```

APPENDIX D. SOURCE CODE

```
37     current_int = 0;
38     // Or
39     current_int |= rest_hf_code;
40     // Update bit_counter
41     bit_counter -= 32;
42 }
43 }
44
45 #pragma omp critical
46 {
47     memcpy(&output_data[out_addrss], int_array, int_counter*4);
48     out_addrss += int_counter*4;
49 }
50 free(int_array);
51 }
52 }
```

```
1 void compressHuffGPU(){
2     // allocate device memory
3     unsigned int *hf_codes_d;
4     unsigned int *hf_lengths_d;
5     unsigned int* output_data_d;
6     unsigned char* input_data_d;
7     unsigned int * int_array;
8
9     cudaMallocHost((void**)&int_array, nbytes);
10    cudaMalloc((void**)&hf_codes_d, 256*sizeof(unsigned int));
11    cudaMalloc((void**)&hf_lengths_d, 256*sizeof(unsigned int));
12    cudaMalloc((void**)&input_data_d, nbytes);
13    cudaMalloc((void**)&output_data_d, nbytes);
14
15    int * current_th;
16    cudaMalloc((void**)&current_th, sizeof(int)*4);
17
18    // set kernel launch configuration
19    dim3 threads = dim3(1, 1);
20    dim3 blocks = dim3(1, 1);
21
22    cudaMemcpy(hf_lengths_d, hf_lengths, 256*sizeof(unsigned int),
23              cudaMemcpyHostToDevice);
24    cudaMemcpy(hf_codes_d, hf_codes, 256*sizeof(unsigned int),
25              cudaMemcpyHostToDevice);
26    cudaMemcpy(input_data_d, input_data, nbytes,
27              cudaMemcpyHostToDevice);
28
29    huff_kernel(blocks, threads, input_data_d, output_data_d,
30              hf_codes_d, hf_lengths_d, current_th);
31
32    int * temp = (int *) malloc(nbytes);
```

D.3. NAIVE DCT

```
29  cudaMemcpy(temp, output_data_d, nbytes, cudaMemcpyDeviceToHost);
30
31  memcpy(output_data, temp, nbytes);
32
33  // release
34  cudaFree(input_data_d);
35  cudaFree(output_data_d);
36  cudaFree(hf_codes_d);
37  cudaFree(hf_lengths_d);
38 }
```

D.3 Naive DCT

```
1  void compressDCT2D(){
2
3  //printf("Compress \n");
4
5  float temp = 0;
6  float tempcos1 = 0;
7  float tempcos2 = 0;
8  float tempsqrt1 = 0;
9  float tempsqrt2 = 0;
10 float pi = 3.14159265358979323846f;
11 int i,j,l,m,x,y;
12
13 #pragma omp parallel for private(tempcos1, tempcos2, temp,
14   tempsqrt1, tempsqrt2, j, y, x, l, m) num_threads(4)
15 for( i = 0; i < (block_size/vc); i+=4){
16     for( j = 0; j < vc; j+=4){
17
18         for( x = 0; x < 2; x++){
19             for( y = 0; y < 2; y++){
20
21                 for( l = 0; l < 4; l++){
22                     for( m = 0; m < 4; m++){
23                         tempcos1 = (((float)2*(float)l)+(float)1)*(float)x*
24                             pi)/((float)8);
25                         tempcos2 = (((float)2*(float)m)+(float)1)*(float)y*
26                             pi)/((float)8);
27                         temp += input_data[(i+1)*vc+j+m] * cos(tempcos1) *
28                             cos(tempcos2);
29                     }
30                 }
31             }
32         }
33     }
34     if(x == 0) tempsqrt1 = 1.0f / sqrt((float)2);
35     else tempsqrt1 = 1.0f;
```

APPENDIX D. SOURCE CODE

```
29         if(y == 0) tempsqrt2 = 1.0f / sqrt((float)2);
30         else tempsqrt2 = 1.0f;
31         DCT_data[(i+x)*vc+j+y] = 0.5f * tempsqrt1 * tempsqrt2 *
           temp;
32         temp=0;
33     }
34 }
35 }
36 }
37 }
```

```
1
2 void compressDCT(){
3
4     float temp = 0;
5     float tempcos = 0;
6     float tempsqrt = sqrt((float)2/(float)8);
7     float pi = 3.14159265358979323846f;
8
9     //#pragma omp parallel for private(tempcos, temp, tempsqrt)
10    num_threads(4)
11    for(int i = 0; i < block_size; i+=8){
12        for(int j = 0; j < 4; j++){
13            for(int k = 0; k < 8; k++){
14                tempcos = (((float)2*(float)k)+(float)1)*(float)j*pi/((
15                    float)16);
16                temp += input_data[i+k] * cos(tempcos);
17            }
18            if(j == 0) tempsqrt = sqrt((float)2/(float)8) / sqrt((float)
19                2);
20            else tempsqrt = sqrt((float)2/(float)8);
21            DCT_data[i+j] = tempsqrt * temp;
22            temp=0;
23        }
24    }
25 }
```

```
1 void decompressDCT(){
2     float temp = 0;
3     float tempcos = 0;
4     float tempsqrt = sqrt((float)2/(float)8);
5     float pi = 3.14159265358979323846f;
6
7     #pragma omp parallel for private(tempcos,temp) num_threads(4)
8     for(int i = 0; i < block_size; i+=8){
9         for(int j = 0; j < 8; j++){
10             temp = DCT_data[i] / sqrt((float)2) ;
11             for(int k = 1; k < 8; k++){
```

D.3. NAIVE DCT

```
12     tempcos = (((float)2*(float)j)+(float)1)*(float)k*pi)/((
13         float)16);
14     temp += DCT_data[i+k] * cos(tempcos);
15 }
16     output_data[i+j] = tempsqrt * temp;
17 }
18 }
```

```
1  __global__ void DCT_kernel(float* input_data_d, float* DCT_data_d,
2     int mul, int ic)
3  {
4     int i = (blockIdx.y * gridDim.x * ic + blockIdx.x * ic +
5         threadIdx.x + blockDim.x * mul)*8;
6
7     float temp = 0;
8     float tempcos = 0;
9     float tempsqrt = sqrt((float)2/(float)8);
10    float pi = 3.14159265358979323846f;
11
12    for(int j = 0; j < 4; j++){
13        for(int k = 0; k < 8; k++){
14            tempcos = (((float)2*(float)k)+(float)1)*(float)j*pi)/((
15                float)16);
16            temp += input_data_d[i+k] * cos(tempcos);
17        }
18        if(j == 0)tempsqrt = sqrt((float)2/(float)8) / sqrt((float)2);
19        else tempsqrt = sqrt((float)2/(float)8);
20        DCT_data_d[i+j] = tempsqrt * temp;
21        temp=0;
22    }
23 }
```

```
1  __global__ void deDCT_kernel(float* input_data_d, float*
2     DCT_data_d, int mul, int ic)
3  {
4     int i = (blockIdx.y * gridDim.x * ic + blockIdx.x * ic +
5         threadIdx.x + blockDim.x * mul)*8;
6
7     float temp = 0;
8     float tempcos = 0;
9     float tempsqrt = sqrt((float)2/(float)8);
10    float pi = 3.14159265358979323846f;
11
12    for(int j = 0; j < 8; j++){
13        temp = DCT_data_d[i] / sqrt((float)2) ;
14        for(int k = 1; k < 8; k++){
15            tempcos = (((float)2*(float)j)+(float)1)*(float)k*pi)/((
16                float)16);
```


APPENDIX D. SOURCE CODE

```
14     temp += DCT_data_d[i+k] * cos(tempcos);
15     }
16     input_data_d[i+j] = tempsqrt * temp;
17 }
18 }
```

D.4 Fast DCT AAN

```
1 void compressDCTAAN(){
2
3     float s0 = 0;
4     float s1 = 0;
5     float s2 = 0;
6     float s3 = 0;
7     float s4 = 0;
8     float s5 = 0;
9     float s6 = 0;
10    float s7 = 0;
11
12    float temp0 = 0;
13    float temp1 = 0;
14    float temp2 = 0;
15    float temp3 = 0;
16    float temp4 = 0;
17    float temp5 = 0;
18    float temp6 = 0;
19    float temp7 = 0;
20    float tempA5 = 0;
21
22    float pi = 3.14159265358979323846f;
23
24    float a1 = sqrt(2.0f)/2;
25    float a5 = cos(3*pi/8.0f);
26    float a2 = sqrt(2.0f) * a5;
27    float a3 = a1;
28    float a4 = sqrt(2.0f) * cos(pi/8.0f);
29
30    float s_0 = sqrt(2.0f)/4.0f;
31    float s_1 = 1.0f / (4.0f * cos(1.0f * pi / 16.0f));
32    float s_2 = 1.0f / (4.0f * cos(2.0f * pi / 16.0f));
33    float s_3 = 1.0f / (4.0f * cos(3.0f * pi / 16.0f));
34    float s_4 = 1.0f / (4.0f * cos(4.0f * pi / 16.0f));
35    float s_5 = 1.0f / (4.0f * cos(5.0f * pi / 16.0f));
36    float s_6 = 1.0f / (4.0f * cos(6.0f * pi / 16.0f));
37    float s_7 = 1.0f / (4.0f * cos(7.0f * pi / 16.0f));
```

D.4. FAST DCT AAN

```
38
39 #pragma omp parallel for private(s0,s1,s2,s3,s4,s5,s6,s7, temp0,
    temp1,temp2,temp3,temp4,temp5,temp6,temp7,tempA5) num_threads
    (4)
40 for(int i = 0; i < block_size; i+=8){
41     temp0 = input_data[i+0];
42     temp1 = input_data[i+1];
43     temp2 = input_data[i+2];
44     temp3 = input_data[i+3];
45     temp4 = input_data[i+4];
46     temp5 = input_data[i+5];
47     temp6 = input_data[i+6];
48     temp7 = input_data[i+7];
49
50     s0 = temp0 + temp7;
51     s1 = temp1 + temp6;
52     s2 = temp2 + temp5;
53     s3 = temp3 + temp4;
54     s4 = -temp4 + temp3;
55     s5 = -temp5 + temp2;
56     s6 = -temp6 + temp1;
57     s7 = -temp7 + temp0;
58
59     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
    temp5 = s5; temp6 = s6; temp7 = s7;
60
61     s0 = temp0 + temp3;
62     s1 = temp1 + temp2;
63     s2 = -temp2 + temp1;
64     s3 = -temp3 + temp0;
65     s4 = -temp4 - temp5;
66     s5 = temp5 + temp6;
67     s6 = temp6 + temp7;
68
69     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
    temp5 = s5; temp6 = s6; temp7 = s7;
70
71     s0 = temp0 + temp1;
72     s1 = -temp1 + temp0;
73     s2 = (temp2 + temp3) * a1;
74     s4 = temp4 * a2;
75     s5 = temp5 * a3;
76     s6 = temp6 * a4;
77     tempA5 = (temp4+temp6)*a5;
78
79     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
    temp5 = s5; temp6 = s6; temp7 = s7;
80
81     s4 = -temp4 - tempA5;
```

APPENDIX D. SOURCE CODE

```

82     s6 = temp6 - tempA5;
83
84     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
      temp5 = s5; temp6 = s6; temp7 = s7;
85
86     s2 = temp2 + temp3;
87     s3 = temp3 - temp2;
88     s5 = temp5 + temp7;
89     s7 = -temp5 + temp7;
90
91     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
      temp5 = s5; temp6 = s6; temp7 = s7;
92
93     s4 = temp4 + temp7;
94     s5 = temp5 + temp6;
95     s6 = -temp6 + temp5;
96     s7 = temp7 - temp4;
97
98     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
      temp5 = s5; temp6 = s6; temp7 = s7;
99
100    DCT_data[i+0] = temp0 * s_0;
101    DCT_data[i+2] = temp2 * s_2;
102    DCT_data[i+1] = temp5 * s_5;
103    DCT_data[i+3] = temp7 * s_7;
104 }
105 }
}

1  _global__ void DCTAANrow_kernel(float* input_data_d, float*
      DCT_data_d, int mul, int ic)
2  {
3      int i = (blockIdx.y * gridDim.x * ic + blockIdx.x * ic +
          threadIdx.x + blockDim.x * mul)*8;
4
5      float s0 = 0;float s1 = 0;float s2 = 0;float s3 = 0;float s4 =
          0;float s5 = 0;float s6 = 0;float s7 = 0;
6      float temp0 = 0;float temp1 = 0;float temp2 = 0;float temp3 = 0;
          float temp4 = 0;float temp5 = 0;float temp6 = 0;float temp7 =
          0;float tempA5 = 0;
7      float pi = 3.14159265358979323846f;
8      float a1 = sqrt(2.0f)/2;float a5 = cos(3*pi/8.0f);float a2 =
          sqrt(2.0f) * a5;float a3 = a1;float a4 = sqrt(2.0f) * cos(pi
          /8.0f);
9      float s_0 = sqrt(2.0f)/4.0f;float s_1 = 1.0f / (4.0f * cos(1.0f
          * pi / 16.0f));float s_2 = 1.0f / (4.0f * cos(2.0f * pi /
          16.0f));float s_3 = 1.0f / (4.0f * cos(3.0f * pi / 16.0f));
10     float s_4 = 1.0f / (4.0f * cos(4.0f * pi / 16.0f));float s_5 =
          1.0f / (4.0f * cos(5.0f * pi / 16.0f));float s_6 = 1.0f /
          (4.0f * cos(6.0f * pi / 16.0f));float s_7 = 1.0f / (4.0f *

```

D.4. FAST DCT AAN

```
    cos(7.0f * pi / 16.0f));
11
12 temp0 = input_data_d[i+0];temp1 = input_data_d[i+1];temp2 =
    input_data_d[i+2];temp3 = input_data_d[i+3];temp4 =
    input_data_d[i+4];temp5 = input_data_d[i+5];temp6 =
    input_data_d[i+6];temp7 = input_data_d[i+7];
13
14 s0 = temp0 + temp7;s1 = temp1 + temp6;s2 = temp2 + temp5;s3 =
    temp3 + temp4;s4 = -temp4 + temp3;s5 = -temp5 + temp2;s6 = -
    temp6 + temp1;s7 = -temp7 + temp0;
15 temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
    temp5 = s5; temp6 = s6; temp7 = s7;
16
17 s0 = temp0 + temp3;s1 = temp1 + temp2;s2 = -temp2 + temp1;s3 = -
    temp3 + temp0;s4 = -temp4 - temp5;s5 = temp5 + temp6;s6 =
    temp6 + temp7;
18 temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
    temp5 = s5; temp6 = s6; temp7 = s7;
19
20 s0 = temp0 + temp1;s1 = -temp1 + temp0;s2 = (temp2 + temp3) * a1
    ;s4 = temp4 * a2;s5 = temp5 * a3;s6 = temp6 * a4;tempA5 = (
    temp4+temp6)*a5;
21 temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
    temp5 = s5; temp6 = s6; temp7 = s7;
22
23 s4 = -temp4 - tempA5;s6 = temp6 - tempA5;
24 temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
    temp5 = s5; temp6 = s6; temp7 = s7;
25
26 s2 = temp2 + temp3;s3 = temp3 - temp2;s5 = temp5 + temp7;s7 = -
    temp5 + temp7;
27 temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
    temp5 = s5; temp6 = s6; temp7 = s7;
28
29 s4 = temp4 + temp7;s5 = temp5 + temp6;s6 = -temp6 + temp5;s7 =
    temp7 - temp4;
30 temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
    temp5 = s5; temp6 = s6; temp7 = s7;
31
32 DCT_data_d[i+0] = temp0 * s_0;
33 DCT_data_d[i+1] = temp5 * s_5;
34 DCT_data_d[i+2] = temp2 * s_2;
35 DCT_data_d[i+3] = temp7 * s_7;
36 DCT_data_d[i+4] = temp1 * s_1;
37 DCT_data_d[i+5] = temp4 * s_4;
38 DCT_data_d[i+6] = temp3 * s_3;
39 DCT_data_d[i+7] = temp6 * s_6;
40 }
```

D.5 Fast LOT

```

1  __global__ void LOTrow_kernel(float* input_data_d, float*
   DCT_data_d, int mul, int ic)
2  {
3      int i = (blockIdx.y * gridDim.x * blockDim.x + blockIdx.x *
   blockDim.x + threadIdx.x)*(ic);
4      int i2 = (blockIdx.y * gridDim.x * blockDim.x + blockIdx.x *
   blockDim.x + threadIdx.x)*(ic+16);
5
6      float s0 = 0;float s1 = 0;float s2 = 0;float s3 = 0;float s4 =
   0;float s5 = 0;float s6 = 0;float s7 = 0;
7      float temp0 = 0;float temp1 = 0;float temp2 = 0;float temp3 = 0;
   float temp4 = 0;float temp5 = 0;float temp6 = 0;float temp7 =
   0;
8      float pi = 3.14159265358979323846f;
9      float cos13 = cos((float)0.13*pi);float sin13 = sin((float)0.13*
   pi);float cos16 = cos((float)0.16*pi);float sin16 = sin((
   float)0.16*pi);
10
11
12     temp0 = 0;
13     temp2 = 0;
14     temp4 = 0;
15     temp6 = 0;
16     temp1 = DCT_data_d[i+0] + DCT_data_d[i+1];
17     temp3 = DCT_data_d[i+2] + DCT_data_d[i+3];
18     temp5 = DCT_data_d[i+4] + DCT_data_d[i+5];
19     temp7 = DCT_data_d[i+6] + DCT_data_d[i+7];
20
21     s0 = (temp0+temp1)/2;s2 = (temp2+temp3)/2;s4 = (temp4+temp5)/2;
   s6 = (temp6+temp7)/2;s1 = (temp0-temp1)/2;s3 = (temp2-temp3)
   /2;s5 = (temp4-temp5)/2;s7 = (temp6-temp7)/2;
22     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
   temp5 = s5; temp6 = s6; temp7 = s7;
23
24     s1 = (temp1 * cos13) - (temp3 * sin13);
25     s3 = (temp1 * sin13) + (temp3 * cos13);
26     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
   temp5 = s5; temp6 = s6; temp7 = s7;
27
28     s3 = (temp3 * cos16) - (temp5 * sin16);
29     s5 = (temp3 * sin16) + (temp5 * cos16);
30     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
   temp5 = s5; temp6 = s6; temp7 = s7;
31
32     s5 = (temp5 * cos13) - (temp7 * sin13);
33     s7 = (temp5 * sin13) + (temp7 * cos13);

```

D.5. FAST LOT

```
34 temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
    temp5 = s5; temp6 = s6; temp7 = s7;
35
36 input_data_d[i2+4+0]=s0;
37 input_data_d[i2+4+1]=s1;
38 input_data_d[i2+4+2]=s2;
39 input_data_d[i2+4+3]=s3;
40 input_data_d[i2+4+4]=s4;
41 input_data_d[i2+4+5]=s5;
42 input_data_d[i2+4+6]=s6;
43 input_data_d[i2+4+7]=s7;
44
45 for(int j = 0; j < ic; j+=8){
46     temp0 = DCT_data_d[i+j+0] - DCT_data_d[i+j+1];
47     temp2 = DCT_data_d[i+j+2] - DCT_data_d[i+j+3];
48     temp4 = DCT_data_d[i+j+4] - DCT_data_d[i+j+5];
49     temp6 = DCT_data_d[i+j+6] - DCT_data_d[i+j+7];
50     temp1 = DCT_data_d[i+j+8] + DCT_data_d[i+j+9];
51     temp3 = DCT_data_d[i+j+10] + DCT_data_d[i+j+11];
52     temp5 = DCT_data_d[i+j+12] + DCT_data_d[i+j+13];
53     temp7 = DCT_data_d[i+j+14] + DCT_data_d[i+j+15];
54
55     s0 = (temp0+temp1)/2; s2 = (temp2+temp3)/2; s4 = (temp4+temp5)
        /2; s6 = (temp6+temp7)/2; s1 = (temp0-temp1)/2; s3 = (temp2-
        temp3)/2; s5 = (temp4-temp5)/2; s7 = (temp6-temp7)/2;
56     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
        temp5 = s5; temp6 = s6; temp7 = s7;
57
58     s1 = (temp1 * cos13) - (temp3 * sin13);
59     s3 = (temp1 * sin13) + (temp3 * cos13);
60     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
        temp5 = s5; temp6 = s6; temp7 = s7;
61
62     s3 = (temp3 * cos16) - (temp5 * sin16);
63     s5 = (temp3 * sin16) + (temp5 * cos16);
64     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
        temp5 = s5; temp6 = s6; temp7 = s7;
65
66     s5 = (temp5 * cos13) - (temp7 * sin13);
67     s7 = (temp5 * sin13) + (temp7 * cos13);
68     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
        temp5 = s5; temp6 = s6; temp7 = s7;
69
70     input_data_d[i2+12+j+0]=s0;
71     input_data_d[i2+12+j+1]=s1;
72     input_data_d[i2+12+j+2]=s2;
73     input_data_d[i2+12+j+3]=s3;
74     input_data_d[i2+12+j+4]=s4;
75     input_data_d[i2+12+j+5]=s5;
```

APPENDIX D. SOURCE CODE

```

76     input_data_d[i2+12+j+6]=s6;
77     input_data_d[i2+12+j+7]=s7;
78 }
79
80     temp0 = DCT_data_d[i+ic-8] - DCT_data_d[i+ic-7];
81     temp2 = DCT_data_d[i+ic-6] - DCT_data_d[i+ic-5];
82     temp4 = DCT_data_d[i+ic-4] - DCT_data_d[i+ic-3];
83     temp6 = DCT_data_d[i+ic-2] - DCT_data_d[i+ic-1];
84     temp1 = 0;
85     temp3 = 0;
86     temp5 = 0;
87     temp7 = 0;
88
89     s0 = (temp0+temp1)/2;s2 = (temp2+temp3)/2;s4 = (temp4+temp5)/2;
90     s6 = (temp6+temp7)/2;s1 = (temp0-temp1)/2;s3 = (temp2-temp3)
91     /2;s5 = (temp4-temp5)/2;s7 = (temp6-temp7)/2;
92     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
93     temp5 = s5; temp6 = s6; temp7 = s7;
94
95     s1 = (temp1 * cos13) - (temp3 * sin13);
96     s3 = (temp1 * sin13) + (temp3 * cos13);
97     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
98     temp5 = s5; temp6 = s6; temp7 = s7;
99
100    s3 = (temp3 * cos16) - (temp5 * sin16);
101    s5 = (temp3 * sin16) + (temp5 * cos16);
102    temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
103    temp5 = s5; temp6 = s6; temp7 = s7;
104
105    s5 = (temp5 * cos13) - (temp7 * sin13);
106    s7 = (temp5 * sin13) + (temp7 * cos13);
107    temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
108    temp5 = s5; temp6 = s6; temp7 = s7;
109
110    input_data_d[i2+12+ic+0]=s0;
111    input_data_d[i2+12+ic+1]=s1;
112    input_data_d[i2+12+ic+2]=s2;
113    input_data_d[i2+12+ic+3]=s3;
114    input_data_d[i2+12+ic+4]=s4;
115    input_data_d[i2+12+ic+5]=s5;
116    input_data_d[i2+12+ic+6]=s6;
117    input_data_d[i2+12+ic+7]=s7;
118 }
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
25
```

D.5. FAST LOT

```
4   int i = ((blockIdx.y * gridDim.x * blockDim.x+ blockIdx.x *
      blockDim.x + threadIdx.x)*(ic+16))+4;
5   int i2 = (blockIdx.y * gridDim.x * blockDim.x+ blockIdx.x *
      blockDim.x + threadIdx.x)*(ic);
6
7   float s0 = 0;float s1 = 0;float s2 = 0;float s3 = 0;float s4 =
      0;float s5 = 0;float s6 = 0;float s7 = 0;
8   float s00 = 0;float s01 = 0;float s02 = 0;float s03 = 0;float
      s04 = 0;float s05 = 0;float s06 = 0;float s07 = 0;
9   float temp0 = 0;float temp1 = 0;float temp2 = 0;float temp3 = 0;
      float temp4 = 0;float temp5 = 0;float temp6 = 0;float temp7 =
      0;
10  float temp00 = 0;float temp01 = 0;float temp02 = 0;float temp03
      = 0;float temp04 = 0;float temp05 = 0;float temp06 = 0;float
      temp07 = 0;float tempA5 = 0;
11  float pi = 3.14159265358979323846f;
12  float cos13 = cos((float)0.13*pi);float sin13 = sin((float)0.13*
      pi);float cos16 = cos((float)0.16*pi);float sin16 = sin((
      float)0.16*pi);
13
14  for(int j = 0; j < ic; j+=8){
15    s0 = DCT_data_d[i+j+0];s1 = DCT_data_d[i+j+1];s2 = DCT_data_d[
      i+j+2];s3 = DCT_data_d[i+j+3];s4 = DCT_data_d[i+j+4];s5 =
      DCT_data_d[i+j+5];s6 = DCT_data_d[i+j+6];s7 = DCT_data_d[i+
      j+7];
16    s00 = DCT_data_d[i+j+8];s01 = DCT_data_d[i+j+9];s02 =
      DCT_data_d[i+j+10];s03 = DCT_data_d[i+j+11];s04 =
      DCT_data_d[i+j+12];s05 = DCT_data_d[i+j+13];s06 =
      DCT_data_d[i+j+14];s07 = DCT_data_d[i+j+15];
17
18    temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
      temp5 = s5; temp6 = s6; temp7 = s7;
19    temp00 = s00; temp01 = s01; temp02 = s02; temp03 = s03; temp04
      = s04; temp05 = s05; temp06 = s06; temp07 = s07;
20
21    s7 = (temp7*cos13) - (temp5*sin13);
22    s5 = (temp7*sin13) + (temp5*cos13);
23    s07 = (temp07*cos13) - (temp05*sin13);
24    s05 = (temp07*sin13) + (temp05*cos13);
25    temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
      temp5 = s5; temp6 = s6; temp7 = s7;
26    temp00 = s00; temp01 = s01; temp02 = s02; temp03 = s03; temp04
      = s04; temp05 = s05; temp06 = s06; temp07 = s07;
27
28    s5 = (temp5*cos16) - (temp3*sin16);
29    s3 = (temp5*sin16) + (temp3*cos16);
30    s05 = (temp05*cos16) - (temp03*sin16);
31    s03 = (temp05*sin16) + (temp03*cos16);
```


APPENDIX D. SOURCE CODE

```
32     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
      temp5 = s5; temp6 = s6; temp7 = s7;
33     temp00 = s00; temp01 = s01; temp02 = s02; temp03 = s03; temp04
      = s04; temp05 = s05; temp06 = s06; temp07 = s07;
34
35     s3 = (temp3*cos13) - (temp1*sin13);
36     s1 = (temp3*sin13) + (temp1*cos13);
37     s03 = (temp03*cos13) - (temp01*sin13);
38     s01 = (temp03*sin13) + (temp01*cos13);
39     temp0 = s0/2; temp1 = s1/2; temp2 = s2/2; temp3 = s3/2; temp4
      = s4/2; temp5 = s5/2; temp6 = s6/2; temp7 = s7/2;
40     temp00 = s00/2; temp01 = s01/2; temp02 = s02/2; temp03 = s03
      /2; temp04 = s04/2; temp05 = s05/2; temp06 = s06/2; temp07
      = s07/2;
41
42     s0 = temp0 - temp1;
43     s2 = temp2 - temp3;
44     s4 = temp4 - temp5;
45     s6 = temp6 - temp7;
46     s1 = temp00 + temp01;
47     s3 = temp02 + temp03;
48     s5 = temp04 + temp05;
49     s7 = temp06 + temp07;
50     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
      temp5 = s5; temp6 = s6; temp7 = s7;
51
52     s1 = (temp0 - temp1);
53     s0 = temp1+temp0;
54     s3 = (temp2 - temp3) ;
55     s2 = temp3+temp2;
56     s5 = (temp4 - temp5);
57     s4 = temp5+temp4;
58     s7 = (temp6 - temp7);
59     s6 = temp7+temp6;
60     temp0 = s0; temp1 = s1; temp2 = s2; temp3 = s3; temp4 = s4;
      temp5 = s5; temp6 = s6; temp7 = s7;
61
62     input_data_d[i2+j+0] = temp0;
63     input_data_d[i2+j+1] = temp1;
64     input_data_d[i2+j+2] = temp2;
65     input_data_d[i2+j+3] = temp3;
66     input_data_d[i2+j+4] = temp4;
67     input_data_d[i2+j+5] = temp5;
68     input_data_d[i2+j+6] = temp6;
69     input_data_d[i2+j+7] = temp7;
70 }
71 }
```

D.6 Hough Transform

```
1  __global__ void hough_kernel(float* output_data_d, int mul, int ic
   , int xc, int vc){
2
3  float d = sqrt((float)xc*xc+vc*vc);
4  float pi = 3.14159265358979323846f;
5  float stepx = (float)xc/(float)d;
6  float stepy = (float)vc/(float)360;
7
8  int i = threadIdx.x + mul*ic;
9  int j = blockIdx.x * 10;
10 int k = blockIdx.y * 10;
11
12 //output_data_d[i]=10;
13
14 for(int l=0; l<vc; l++){
15     float r = (float)j*cos(((float)l/stepy)*(pi/180)) + (float)k*
        sin(((float)l/stepy)*(pi/180));
16     float pointer = abs(r)*stepx;
17     output_data_d[i*xc*vc+(int)pointer*vc+l]+=25;
18 }
19 }
```

D.7 Convolution

```
1
2  __constant__ float d_filter[8788];
3
4  __global__ void increment_kernel(float *d_data, float *d_token,
   int len, int x, int y, int offsetnrx, int offsetnry) {
5
6  //d_data[0]= 0;
7
8  float imagevalue =0;
9
10 for (int l = 0; l < len; l++) {
11     for (int m = 0; m < len; m++) {
12         for (int n = 0; n < len; n++) {
13
14             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))];
15             float temp2 = d_filter[(len*len)*l + (len)*m + n];
```

APPENDIX D. SOURCE CODE

```
16         imagevalue += temp * temp2;
17     }
18 }
19 }
20 }
21     d_data[(blockIdx.y)* x * y + (blockIdx.x +(gridDim.x*offsetnry
22         ))* x + (threadIdx.x)+(blockDim.x*offsetnrx)] = imagevalue
        / (len*len*len);
}
```

```
1 void createFilter(){
2     const float PI = 3.14159265358979323846f;
3     int len = standardDiv * 4 + 1;
4     int d_filterLength = len*len*len*4;
5     filter = (float *)malloc((len*len*len)*sizeof(float)) ;
6     float tempo;
7
8     for (int i = 0; i < len; i++) {
9         for (int j = 0; j < len; j++) {
10            for (int k = 0; k < len; k++) {
11                tempo = 0;
12                tempo = (1 / (pow(2 * PI, (float)1) * sqrt(
13                    standardDiv)));
14                tempo = tempo * (pow(exp((float)1), -((pow((i - (
15                    standardDiv * 2)), 2) + pow((j - (standardDiv *
16                    2)), 2) + pow((k - (standardDiv * 2)), 2)) /
17                    (2 * pow(standardDiv, 2)))));
18                filter[(len*len*i+len*j+k)] = (float)(50 * tempo);
19            }
20        }
21    }
22 }
```


APPENDIX E

AESC Library Overview

Here we introduce an overview of the functions that can be used within the seismic compression library AESC.

A list of all compression algorithms for the CPU are found below. these are implemented with openMP pragmas and use 4 threads to perform calculations

- compressRLE()
- decompressRLE()
- compressHuff()
- decompressHuff()
- compressDCT()
- decompressDCT()
- compressDCT2D()
- decompressDCT2D()
- compressDCTAAN()
- decompressDCTAAN()
- compressDCTAAN2D()
- decompressDCTAAN2D()
- compressDCTAAN3D()
- decompressDCTAAN3D()
- compressLOT()

-
- decompressLOT()

A list of all compression algorithms for the GPU are found below. these are implemented with NVIDIA CUDA, and perform the encoding step of the transform encoding on the CPU using a modified RLE for seismic data.

- compressDCTGPU()
- decompressDCTGPU()
- compressDCT2DGPU()
- decompressDCT2DGPU()
- compressDCTAANGPU()
- decompressDCTAANGPU()
- compressDCTAAN2DGPU()
- decompressDCTAAN2DGPU()
- compressDCTAAN3DGPU()
- decompressDCTAAN3DGPU()
- compressLOTGPU()
- decompressLOTGPU()

APPENDIX F

Short Paper for PARA 2010

This is an extended abstract (short paper), written to be accepted to give a talk at the PARA 2010 conference in Iceland. The actual paper will be written later as the deadline is in September. The abstract was accepted and the talk was given on Tuesday the 8th of June. For slides look up the HPC-LAB groups webpages.

Accelerating Disk Access Using Compression for Large Seismic Datasets on modern GPU and CPU

Ahmed A. Aqrawi* and Anne C. Elster†

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

Abstract One of the major challenges of modern architectures is to overcome the limitations of disk and memory bandwidth, which per today are much slower than computation speeds. In this paper, several compression methods for efficient disk access on both the CPU and GPU are described and empirically tested. To reduce I/O time we have tested both lossless and lossy compression algorithms and hardware alternatives. Our results show that an I/O speedup of 2 is achieved by using an SSD vs. HDD disk on seismic data. The use of compression for I/O gave a speedup of 1.08 and 1.2 with lossless compression methods of RLE and Huffman, respectively, and up to 6 for lossy methods with an average error of 0.46%. Lossy methods include performing variations of DCT, in several dimensions, and combining these with lossless compression methods such as RLE and Huffman. The speedup was achieved by enabling collaboration between the CPU and GPU.

Keywords GPGPU, CUDA, Compression, I/O acceleration, Large datasets, SSD Disk

1 Introduction

Processing large datasets is very challenging for modern architectures since memory and disk access is now much slower than computation speeds. 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 1 for an example of seismic data. However, these filter algorithms are very computationally intensive and the data needed for these calculations are larger than the space available in memory, causing several disk accesses that slow down processing of the seismic data. We are attempting to accelerate this process.

In recent years, it has been shown that the performance capabilities of the GPU, in many cases, have exceeded that of the CPU. This has led 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 process-

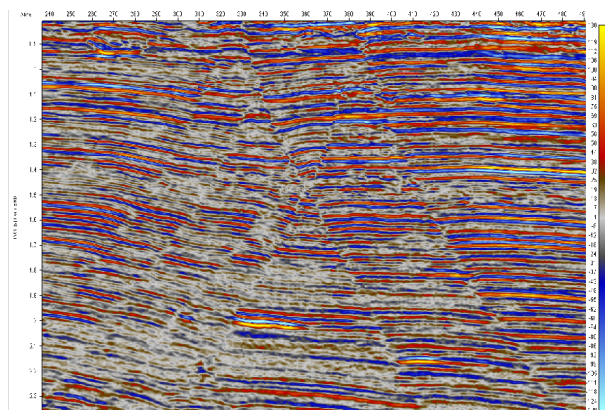


Figure 1: Seismic data example provided by Aarre [1]

ing architectures and manufacturers have started introducing new product lines specific for scientific calculations, including the recently announced NVIDIA s2070 which is to be based on their new Fermi GPU processor. Note also that using the GPU for computations, frees up CPU cycles for other parallel tasks. The GPU, can therefore, like other co-processors, be viewed as an accelerator that collaborates with the CPU.

Several tasks, such as image processing, seismic processing (Aksnes and Elster [3]) and other physical modeling as well as linear programming applications (Spampinato and Elster [16]) have proven to be well parallelizable on the GPU. In a previous project [4], we worked with accelerating seismic algorithms by using the GPU and CUDA. This resulted in a drastic change between the balance of computation and communication. To begin with 2% of the execution time is spent on disk access, while after accelerating the process 90% of the time is now used on communication. The goal of this project is to look at compression of large seismic data, and see if it can be used to accelerate I/O disk access to complement our previous work and accelerate both aspects of the seismic process.

2 Related Work

Data compression has existed for quite some time now and there have been a lot of progression in the field. There are mainly two categories of compression algorithms [9],

*Email: aqrawi@stud.ntnu.no

†Email: elster@idi.ntnu.no

lossless and lossy. Lossless algorithms are compression algorithms that do not lose any data in the compression process. Lossy algorithms are those that change the original data before compression and thereby resulting in a loss of data.

When it comes to lossless compression of volume data there are many ways to do so, but they mainly focus on entropy encoding. This means that they focus on the nature of the data. The most known here are run length encoding (RLE) and huffman encoding. Fowler and Yaglet [6] show ways to improve on the existing method of huffman encoding, which we have done as well. Other than entropy encoding, there is also arithmetic encoding that is used often in lossless compression. Ratanaworabhan et al. [13] introduce a method of compressing floating point data. Since seismic data is a collection of floating point data and it is interesting to compress with this method. Xie and Qin [18] buildt upon the method with respect to siesmic data and got compression ratio of 1.7.

Lossy compression is performed by combining a transform algorithm such as the fourier, cosine or lapped orthogonal transforms with a lossless compression method [10]. By transforming the data to another domain and removing the data with least effect on the original data, larger compression is attained with little loss of data. This technique was shown with the DCT by A.B. Watson [17]. One of the most common transforms to use is the discrete cosine transform, introduced by N. Ahmed et al. [2], and later on optimized in a SQ scheme for image processing by Y. Arai et al. [5]. This is now part of the JPEG image standard [12]. Malvar et al [11] introduced ways to improve on the DCT in compression with the use of the LOT, which removes blocking effects from images.

3 Hardware and Implementations

In Tables 1 and 2, are lists of the hardware used in the two machines that we benchmarked our results on. There are significant differences in both machnies. First is the disk on the second machine is an SSD disk and that in itself can improve I/O time. The other difference is that there is a more powerful CPU on the second machine, which will improve execution time results, and the GPU is actually a multiple GPU architecture which combines 4 Tesla 1060.

Type	Name
CPU	Intel Core 2 Quad Q9550 2.83GHz
GPU	Nvidia Tesla 1060
Memory	8GB DDR3
Disk	Samsung HDD disk HD735 500GB

Table 1: Hardware in first computer used in benchmark

3.1 Implementations

It is important to understand that in our case, when competing with I/O time, it is not only important that we compress

Type	Name
CPU	Intel Core i7 extreme 965 3.2GHz
GPU	Nvidia Tesla s1070
Memory	12GB DDR3
Disk	Cosair SSD 128GB

Table 2: Hardware in second computer used in benchmark

the data as much as possible, but it should be done efficiently. That is why sometimes one has to comprise compression rate for execution-time efficiency to even beat the I/O time. Another aspect worth mentioning is that all the implementations are focused on optimizing seismic compression, such that they might not be optimal for data of another nature.

seismic data is produced in the Segy format [15]. This format is very extensive to explain, for detailed descriptions read the manual [15]. The most important aspects for compression and I/O are the fact that the data is quite spread on disk and has a lot of detail to the extent that it looks noise. The fact that the data is spread on disk results in umps across the disk resulting in lower not optimal disk access. The noisy data means that it is hard to compress with traditional methods.

We first experimented with lossless compression methods such as RLE and Huffman without any modifications to fit the nature of seismic data. RLE resulted in no compression because of the noisy nature of seismic data. After analyzing several seismic datasets we realized that if we combined RLE with a dictionary lookup for the values that benefit from it, this results in the best compression rates for the RLE algorithm. When it comes to huffman it gave better compression rates, but used more time to compress because it needs to scan the whole data twice. First to create a statistic to create the tree and second to compress the data using this tree. This can be optimized given since we are looking at seismic data. This means that the tree and statistic will always be the same and therefore can be pre-calculated. This makes the algorithm much faster and able to beat RLE in both execution time and compression rates.

On the GPU both of these algorithms are challenging to perform since they have a sequential nature when combining the compressed values. In other words the compression can be done in parallel, but putting the pieces together is not and requires communication between the threads, which is another overhead. Another limiting factor is that both these algorithms and other lossless algorithms use bit-wise operation, which are really slow on the GPU. Non the less a GPU implementation of Huffman was produced to test these statements. The results show that the CPU is superior to the GPU in bit wise operations and that combined with the sequential nature of the algorithms gave poor results. In general there is a relation between compression

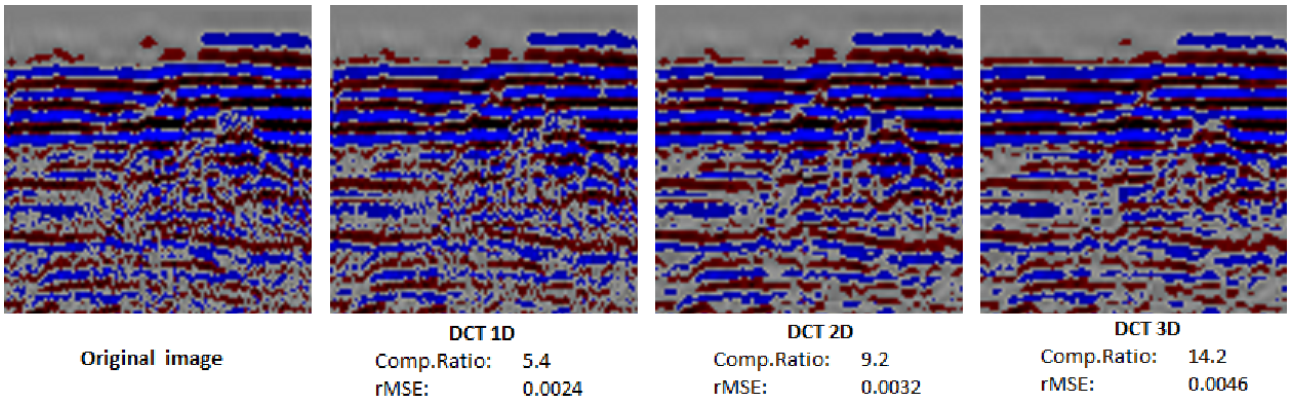


Figure 2: 5x zoomed into seismic images showing effects of lossy compression with DCT in several dimensions

ratio and speedup, and if the execution time of the compression algorithms approaches zero, the I/O speedup will be equal to the compression ratio. lossless compression has shown to give little compression ratio. That is why we did not implement floating point compression even though it is a good alternative for compression it is time consuming and therefore not fitting for this application.

Given that lossless compression is a combination of changing the data and then compressing it using a lossless method, we have the advantage of letting the CPU and GPU cooperate by offloading the data changing to the GPU and the lossless compression can be done on the CPU. In our case we have implemented the DCT as mentioned in [2] in several dimensions on the GPU and we remove the least significant data from the transformed data before compression. We used a block size of 8, which is common in the JPEG standard [12]. And removed the 4 least important values. The transform algorithm is combined with the lossless compression of RLE, which proved to be the best combination. We also implemented fast DCT algorithms such as the AAN [5] [12] and compared all implementations between CPU and GPU performance and the rate at which they are able to accelerate I/O.

When testing we are looking at large datasets. Our definition of a large dataset is that a dataset that does not fit into memory. That is why our largest dataset is at 12GB, and since we have to duplicate tables to store input and output data and the operating system needs some of the memory, even on the computer with 12 GB of memory only 4GB of data can be read at a time. The GPU we use also has a limiting factor there by only having 4GB of memory which means that when performing calculations on the GPU we have to transfer back and forth 3 times the input and output data. All buffered reads and paging functionalities are turned off such that the times recorded are representative.

$$rMSE = \sqrt{\frac{\sum_{x=0}^N (f(x) - g(x))^2}{N}} \quad (1)$$

Our tests are conducted such that the code is executed 10 times and the median execution time is chosen as a rep-

resentative. This is to avoid spike values and since we have a lot to test and large sets that take long time to test only 10 executions are performed, but we believe that this is representative. During executions we have also developed a framework that produces images from the seismic set (this was done with help from [1] [?]), produce the compressed file and we calculate the error for lossy algorithms using the formula for root mean square error as stated in [7], which is shown in Equation 1.

4 Results

When testing the different algorithms we have focused on testing on two platforms, namely an HDD disk and SSD disk. This is because the SSD is faster and it would be interesting to see if these algorithms are able to beat the I/O time of an SSD disk. The results from Figure 3 show that compression can in fact increase I/O performance with the use of the GPU, but only for lossy methods. The error calculated varied for each dimension we increase in the DCT. This is because as we increase the dimensions a bit more information is removed to increase compression. There is a balance here, which is shown in Figure 2.

The lossless methods proved to give some speedup in I/O, but given their lack of compression ability because of the noise in the seismic data, they proved to be of little efficiency. No significant results were obtained on either platforms (HDD and SSD), and when tested on the GPU it resulted in longer execution times. This is mainly because of the limiting factors of the GPU's bit-wise operation capabilities and the fact that both RLE and Huffman encoding have a sequential nature.

By transforming and low pass filtering the data, as we did with the DCT, we could remove the noise from the data and get much better compression rates with low errors. When this was performed the naive DCT method showed no significant results on the CPU, but on the GPU we were able to accelerate the transformation significantly and this resulted in a 5 time speedup in I/O for the two dimensional case. When testing for the three dimensional case of the naive DCT even the GPU could not produce sig-

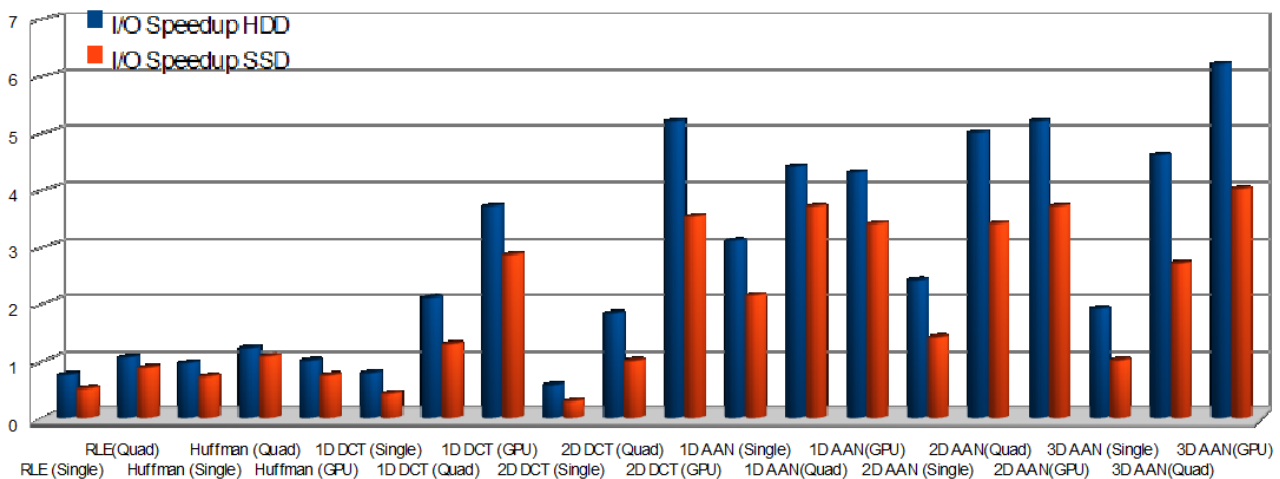


Figure 3: I/O speedup on HDD and SSD platforms

nificant results, which signaled the need for a faster DCT algorithm. Since the DCT is such a popular algorithm in image processing, there are significant advances. The most commonly recognized fast algorithm is the AAN algorithm [5]. Our implementation of the AAN gave good results even on the CPU, and the best results on the GPU, which is a 6 time speedup for the three dimensional case.

5 Conclusions and Future Work

Our results show that by compressing data one is able to increase the performance of I/O for seismic data. This was achieved with lossy compression that first filtered the seismic data to reduce noise, using DCT with a low pass filter, and then compressing using RLE. The implementation is done cooperatively between the CPU and GPU. Our results showed that the CPU is more efficient at lossless compression and that the GPU is faster at filtering, which is why we combined the two. Since seismic data is used in scientific calculations the lossy compression could not have a large error. We achieved a 6 time I/O speedup with an error of about 0.4%. Future work in this subject would be to implement LOT and GenLOT to reduce the error and to test the effects this speedup has on the seismic process as a whole.

References

- [1] V. Aarre. Schlumberger stavanger. personal communication.
- [2] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transform. *IEEE Trans Computing*, 1974.
- [3] E. Aksnes and A. C. Elster. Porus rock simulations and lattice boltzmann on gpus. *to be published in Parallel Computing: Architectures, Algorithms and Applications: Proceedings of the International Conference ParCo 2009, IOS Press*, 2010.
- [4] A. A. Aqrabi. 3d convolution of large datasets on modern gpus. Norwegian University of Science and Technology, 2009.
- [5] Y. Arai, T. Agui, and M. Nakajima. A fast dct-sq scheme for images. *Transactions of the Institute of Electronics, Information and Communication Engineers*, 1988.
- [6] J. E. Fowler and R. Yagelt. Lossless compression of volume data. *IEEE Transactions on Image Processing*, 1995.
- [7] R. C. Gonzales and R. E. Woods. *Digital Image Processing*. Prentice-Hall PTR, third edition, 2008.
- [8] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors*. Elsevier INC., first edition, 2010.
- [9] P. Komma, J. Fischer, F. Duffner, and D. Bartz. Lossless volume data compression schemes. *Tagung conference on simulation and visualization*, 2007.
- [10] C. Larsen. Utilizing gpus on cluster computers. Norwegian University of Science and Technology, 2006.
- [11] H. S. Malvar and D. H. Staelin. The lot: Transform coding without blocking effect. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1989.
- [12] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression*. Van Nostrand Reinhold, 1st edition, 1993.
- [13] P. Ratanaworabhan, J. Ke, and M. Burtcher. Fast lossless compression of scientific floating-point data. *IEEE Computer society*, 2006.
- [14] R. Eidissen. Comparing cg and cuda implementations of selected transform algorithms. *Norwegian University of Science and Technology*, 2008.
- [15] Seg Technical Standards Committee. *SEG Y rev 1 Data Exchange Format*, 2002.
- [16] D. Spampinato. Modeling communication on multi-gpu systems. Norwegian University of Science and Technology, 2009.
- [17] A. B. Watson. Image compression using the discrete cosine transform. *Mathematica*, 1994.
- [18] X. Xie and Q. Qin. Fast lossless compression of seismic floating-point data. *IEEE Computer society*, 2009.

APPENDIX G

Poster ISC 2010

This is a poster used at the International super computing 2010 (ISC 2010) conference in Hamburg Germany. This is a poster displayed on our stand with intentions to inform of our newly gained results with the newest technology with in the field, namely the FERMI architecture. the major focus here is to inform how the Fermi architecture boosted our results. We are happy to inform that the poster gain some attention, and participants were curious as to how we achieved our results and how much different was it to work with the technology.

The poster can be found on the next page.

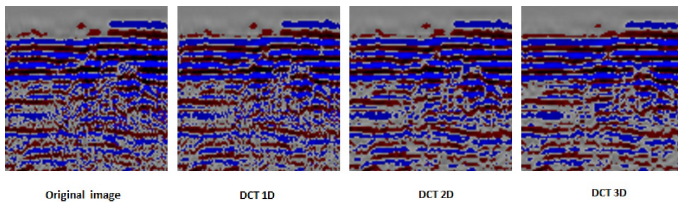
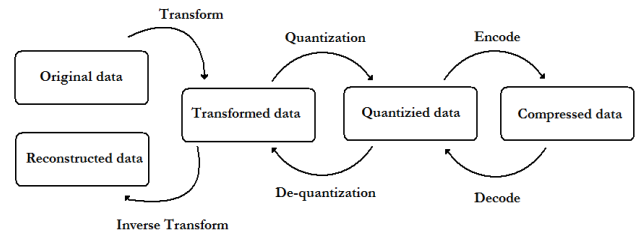
I/O Speedup for Large Datasets Using Compression on Modern CPU & GPU Architectures (FERMI)

Ahmed A. Aqrabi (Master Student) and Dr. Anne C. Elster (Advisor)
Department of computer and information science, NTNU

I/O bandwidth is often a major bottleneck for dataintensive algorithms. Our work looks at seismic filtering as an example of dataintensive algorithm, and aims at accelerating the process with the use of compression. By compressing the data, less data is read from the disk, and the computation capabilities of the GPU is then used for faster decompression.

Transform Encoding

The transform encoding process includes three steps shown in the figure to the right. The transformation and quantization steps, which are to transform the data into another domain and filter it, are best done using the GPU because of their parallelizable nature. While the encoding step, which in our case is a modified RLE, is best done on the CPU because of its sequential nature. We have used the GPU and CPU in asynchronous co-operation to perform the compression.



Visual Results

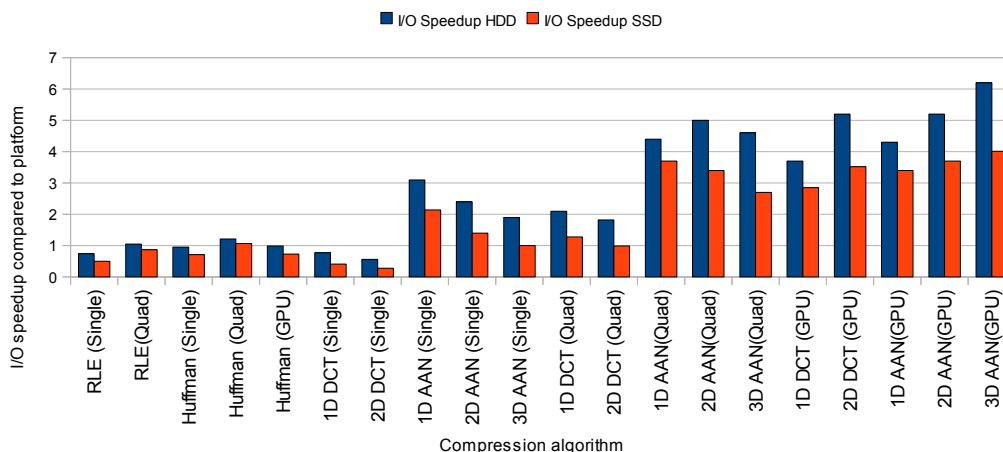
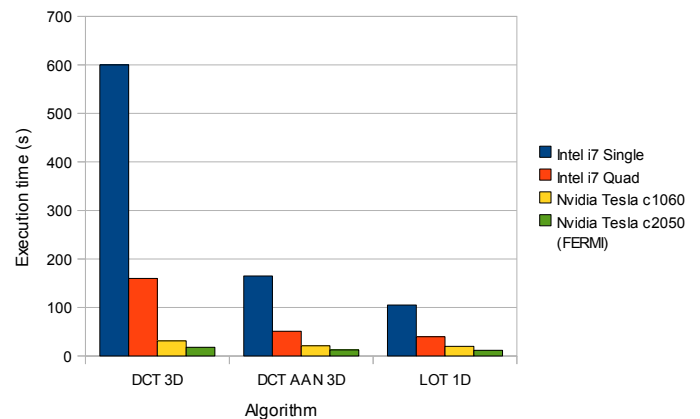
We performed our compression on seismic data of large sizes (32GB). Transform Coding is a lossy compression method, which means that some data is lost in the process. We have measured the amount of data lost by calculating the root mean square error (rMSE), and as we can see from the images to the left, little data is lost using this method. But, one can clearly see a visible loss.

Benchmarks

In the graph to the right, we have shown the various execution times for the main compression algorithms on different platforms. The results show that the GPU implementations are superior in computation power, and within the GPU results the FERMI GPU is about 1.7 times as fast.

In the graph below, we show the results of accelerating I/O on different platforms using compression. These are results for asynchronous executions, where the CPU is reading data and aiding the encoding, simultaneously as the GPU performs the transformations and quantizations. We have tested the process on both HDD and SSD disks.

Execution time comparison to FERMI architecture



Conclusions & Future Work

The results show that an I/O speedup of up to 6.2 can be achieved using this method with the use of the GPU in the compression process. Future work would involve looking at other compression alternatives, and more advanced hardware as it is made available. A Multi-GPU solution can also be explored.

Acknowledgements

We would like to thank NVIDIA for their contribution of hardware through their professor affiliates program. We would also like to thank Scumberger and Statoil for their contribution of seismic data. A special thanks to Christian Larsen for all his help.



HPC-Lab

NTNU

Norwegian University of Science and Technology