Norwegian University of
Science and Technology

Seismic Data Compression and GPU Memory Latency

Daniel Haugen

Master of Science in Computer Science
Submission date: June 2009
Supervisor: Anne Cathrine Elster, IDI
Co-supervisor: Tore Fevang, Schlumberger

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

Problem Description

Propose and evaluate different strategies to effectively move (/stream) large amounts of seismic 3D data from disk and/or RAM into memory on the graphics processor. The seismic data may be pre-processed, e.g. compressed or re-organized, to achieve satisfiable performance.

Assignment given: 27. January 2009
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of
Science and Technology

Master Thesis

Seismic Data Compression and GPU Memory Latency

Daniel Haugen

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

June 2009

Supervisor
Dr. Anne C. Elster

Co-Supervisor
Tore Fevang

Abstract

The gap between processing performance and the memory bandwidth is still increasing. To compensate for this gap various techniques have been used, such as using a memory hierarchy with faster memory closer to the processing unit. Other techniques that have been tested include the compression of data prior to a memory transfer. Bandwidth limitations exists not only at low levels within the memory hierarchy, but also between the central processing unit (CPU) and the graphics processing unit (GPU), suggesting the use of compression to mask the gap.

Seismic datasets are often very large, e.g. several terabytes. This thesis explores compression of seismic data to hide the bandwidth limitation between the CPU and the GPU for seismic applications. The compression method considered is subband coding, with both run-length encoding (RLE) and Huffman encoding as compressors of the quantized data. These methods has shown on CPU implementations to give very good compression ratios for seismic data.

A proof of concept implementation for decompression of seismic data on GPUs is developed. It consists of three main components: First the subband synthesis filter reconstructing the input data processed by the subband analysis filter. Second, the inverse quantizer generating an output close to the input given to the quantizer. Finally, the decoders decompressing the compressed data using Huffman and RLE. The results of our implementation show that the seismic data compression algorithm investigated is probably not suited to hide the bandwidth limitation between CPU and GPU. This is because of the steps taken to do the decompression are likely slower than a simple memory copy of the uncompressed seismic data. It is primarily the decompressors that are the limiting factor, but in our implementation the subband synthesis is also limiting. The sequential nature of the decompression algorithms used makes them difficult to parallelize to make use of the processing units on the GPUs in an efficient way.

Several suggestions for future work is then suggested as well as results showing how our GPU implementation can be very useful for data compression for data to be sent over a network. Our compression results give a compression factor between 27 and 32, and a SNR of 24.67dB for a cube of dimension 643. A speedup of 2.5 for the synthesis filter compared to the CPU implementation is achieved (2029.00/813.76 2.5). Although not currently suited for the GPU-CPU compression, our implementations indicate

that the transfer of seismic data over network can be improved by approximately a factor of 25.

Acknowledgments

I would like to thank the following persons and companies:

- Dr. Anne C. Elster, my supervisor, for feedback on the project.
- Tore Fevang, my co-supervisor at Schlumberger for invaluable input and guidance throughout the work on the master's thesis.
- NVIDIA, for donating GPUs to the HPC-lab of our department through their Professor Partnership Program with Elster.

Contents

1 Introduction **1**
1.1 Problem 2
1.2 Goals 3
1.3 Outline 3

2 Technical background **5**
2.1 Compression 5
2.1.1 Terms used discussing compression 5
2.1.2 Run length encoding 6
2.1.3 Huffman coding 7
2.1.4 Parallel approaches 8
2.1.5 Image compression 9
2.2 Subband coding 10
2.2.1 Overview of subband coding 10
2.2.2 Decimation and interpolation 11
2.2.3 Quantization and inverse quantization 11
2.2.4 Analysis stage 13
2.2.5 Synthesis stage 14
2.2.6 Separable filters 15
2.2.7 Filter extension 17
2.2.8 The “black box” stage 18

3 GPU programming **21**
3.1 NVIDIA’s Tesla architecture 21
3.2 NVIDIA CUDA 24
3.2.1 NVIDIA CUDA extensions to C 25
3.2.2 NVIDIA CUDA’s memory hierarchy 27
3.2.3 Shared memory 27
3.2.4 Global memory 28

4 Methodology **33**
4.1 Run-length encoding implementation . . . 33
4.1.1 Layout of the RLE data . . . 33
4.1.2 The RLE decoding kernel . . . 34
4.2 Subband transform implementation . . . 36
4.2.1 Description of the implementation . . . 36
4.2.2 Mirroring . . . 42
4.2.3 Memory handling . . . 43
4.2.4 Description of the interleaved format . . . 43
4.3 Huffman decoding implementation . . . 44
4.4 Transpose . . . 46
4.4.1 2-D transpose . . . 47
4.4.2 3-D transpose . . . 47

5 Results **53**
5.1 Testing environment . . . 53
5.2 Benchmarking . . . 53
5.3 Compression efficiency . . . 56
5.4 Discussing the results . . . 58
5.4.1 GPU memory accesses . . . 58
5.4.2 Branching in GPU . . . 59
5.4.3 Inverse quantization and alignment . . . 59
5.5 Proposing improvements to the implementation . . . 60
5.5.1 Planning tool . . . 61
5.5.2 How fast is the subband synthesis filter? . . . 62
5.5.3 Constant memory cache . . . 63
5.6 Our compression algorithms . . . 64
5.6.1 GPU versus CPU precision . . . 64

6 Conclusions **67**
6.1 Future work . . . 68

A Filter coefficients **71**

B NOTUR2009 poster **76**

List of Tables

2.1 Sobel operator of size 3×3. 15

4.1 File format of Huffman encoded data from `libhuffman`. 45
4.2 Node structure 46

5.1 Various timing results 55
5.2 NVIDIA CUDA Visual Profiler timing results. 56
5.3 Compression results 57

A.1 Analysis filter coefficients in the temporal direction 72
A.2 Synthesis filter coefficients in the temporal direction 73
A.3 Analysis filter coefficients in the spatial direction 74
A.4 Synthesis filter coefficients in the spatial direction 75

List of Figures

2.1 Huffman trees 8
2.2 Layout of seismic data 10
2.3 Subband encoding 11
2.4 Subband decoding 11
2.5 Overview of a M-channel filter bank system 15
2.6 Separable filter over a 2-D image 16
2.7 Extended input and filtered output 18

3.1 Tesla architecture 22
3.2 The texture/processor cluster (TPC) 23
3.3 The streaming processor (SM) 25
3.4 Coalesced memory access, compute capability below 1.2 30
3.5 Coalesced memory access, compute capability above 1.2 31

4.1 Illustration of division based on a binary number 36
4.2 Illustration of a convolution step 39
4.3 Relationship between subband indices and coefficients 40
4.4 Mirroring schemes 43
4.5 2-D Transpose 48
4.6 3-D Transpose 51

5.1 Standard deviation over subbands 58

Abbreviations

1-D one-dimensional

2-D two-dimensional

3-D three-dimensional

API application programming interface

BLAS basic linear algebra subprograms

CPU central processing unit

CUDA compute unified device architecture

dB decibel

FFT fast Fourier transform

FLOP floating point operation

FLOPS floating point operations per second

GFLOP gigaFLOP

GFLOPS gigaFLOPS

GPU graphics processing unit

ISA instruction set architecture

JPEG joint photographic experts group

MAD multiply-add

MFLOP megaFLOP

PR perfect reconstruction

PTX parallel thread execution

RLE run-length encoding

ROP raster operation processor

RMS root mean square

SDK software development kit

SFU special-function unit

SIMT single-instruction, multiple-thread

SM streaming multiprocessor

SMC streaming multiprocessor controller

SNR signal-to-noise ratio

SP streaming-processor

SPA streaming processor array

SQ scalar quantization

TPC texture/processor cluster

TWT two-way traveltime

VQ vector quantization

Chapter 1

Introduction

Given todays increase in difference between computational performance and performance of memory, measures have to be taken to reduce this gap. Compression of data before a memory transfer has previously been explored at a low-level, that is between the last-level cache and memory [1]. Compression has also been used further away from cache and memory levels such as in sound, image and video compression, before it is transferred. Without compression many of todays media solutions would not be possible due to the amount of data that is involved and the limited bandwidth.

Current seismic datasets often become several terabytes. Compression of seismic data is hence desirable to improve the efficiency of both storage and transmission By reducing the size of these datasets through compression, they become more available given a limited amount of resources. In this thesis, we investigate the feasibility of using a lossy compression algorithm for seismic data, to determine if it is possible to mask the limiting bandwidth between CPU and GPU.

In this thesis we investigate the feasibility of using a lossy compression algorithm for seismic data, to determine if it is possible to mask the limiting bandwidth between CPU and GPU. Compression of seismic data is desirable to improve the efficiency of both storage and transmission. As seismic data sets can become several terabytes, reducing the size of the data makes it more available given a limited amount of resources.

The acquisition of seismic data offshore, is a process where signals are sent toward the seabed and the time differences received is dependent upon the sediments found below the seabed. A commonly used energy source is air guns firing highly compressed air generating what is know as a P-wave, Røsten [2]. This wave has particles moving in the same direction as the propagation, and is the type of wave recorded in conventional marine seismic exploration, according to Røsten [2]. These air guns are gathered in an array

on a suitable frame that is towed after a survey vessel. In addition to the source, the air guns, receivers are also found behind the vessel. The receivers are pressure-sensitive hydrophones that measures reflected waves that travels upward from the seabed. A pressure wave travels from the source, downward and into the seabed. When a wave hits the interface found between two geologic layers, some of the wave reflects while the rest is transmitted. The hydrophones records the pressure-wave amplitude reflections based on the two-way traveltime (TWT), as described by Røsten [2]. For a more detailed explanation of seismic data processing see the thesis of Røsten [2] and the book on the topic by Yilmaz [3].

Seismic data format

After the data has been recorded and been through preprocessing steps including migration and stacking, it results in a data set that can be interpreted as if the sampling was done in an ideal setting. That is, as if a beam is shot straight down into the seabed, and then received at the same location as the source of the beam. The layout in memory of the data set after all the preprocessing is as follows: Consecutive data represent values in the depth direction. Following the last value in the depth direction is the start of the next column. After processing all the columns in one direction the following column is placed behind the first column of the previous row of columns. If we represent the location of a sample by $f(i, j, k)$, where i denotes the ith plane in the depth direction, j the jth column in a plane and k the kth sample from the top of the column in the current plane. Then the function giving the position in memory of an element is given by $f(i, j, k) = i \times \text{ size(j) } \times \text{ size(k) } + j \times \text{ size(k) } + k$. Each element is represented by a floating point number, this has to be considered when calculating the position. This is the format of the seismic data used in this thesis.

1.1 Problem

This thesis investigates the possibility of using compression of 3-D seismic data as a means of reducing transfer time from CPU memory to the memory found on GPUs. It also opens for the possibility of storing compressed data in the GPU memory for later use. Doing the decompression on a GPU just before visualization is a benefit, making it possible to keep the data set in the GPU memory for a longer time, since it take less storage. Other benefits that come with a compressed 3-D seismic dataset such as reduced transfer time within a network, is discussed, but not the primary focus of our investigation.

Another motivation for compressing seismic data is because some oil companies consider their seismic data so valuable, they do not allow storage of their seismic data on the local machine, so it has to be transferred over a network each time it is used. Compression hence can give huge time savings when the dataset become big.

1.2 Goals

The primary goal of this work is to determine the feasibility of compression to reduce bandwidth requirements in addition to the need of storage space on GPUs (when compressed). To achieve this, we look at using the seismic data compression algorithm presented by Røsten in his dissertation, [2]. This algorithm uses subband coding targeted at seismic data and results in good compression ratios for seismic data.

As part of the compression algorithm presented by Røsten, entropy coding is used. Thus, decoding using the Huffman algorithm and run-length encoding on GPUs will be investigated along with the subband coding.

A poster by Leif C. Larsen et al., [4], shows that GPUs is favorable in speeding up transform algorithms for image compression. Therefore, subband transform on seismic data using GPUs can also be favorable.

1.3 Outline

This thesis is organized as follows:

Chapter 2 presents some technical background on basic compression methods, and subband coding theory.

Chapter 3 describes details of the NVIDIA Tesla GPU architecture.

Chapter 4 explains the details around the implementation, including the steps in the different parts of the decompression algorithm, and some of our design decisions. Our implemented GPU kernels are also presented.

Chapter 5 presents our results and discusses our finding along with several suggestions for improvements.

Chapter 6 concludes our findings and how they may be applied to seismic data, as well as presents some suggestions to future work that might improve our results.

Appendix A lists the coefficient tables used in the subband coding.

Appendix B presents our NOTUR2009 poster that summarizes some of this work.

Chapter 2

Technical background

This chapter will present different aspects to consider while solving the problem at hand. First an overview of the theory behind compression is presented, followed by the subband coding.

2.1 Compression

This section will try to give a concise description of different compression techniques applied in image compression. Starting out with lossless methods that are often used in combination with lossy image compression.

2.1.1 Terms used discussing compression

Before the different compression methods are explained, some terms used while describing compression are presented. These are the vocabulary words used by David Salomon in his book Data Compression [5].

The program responsible of compressing the input data stream and producing a compressed output stream, with low redundancy, is known as the *compressor* or *encoder*. The reverse process is done through a *decompressor* or *decoder*. It is not unusual to use the term *stream* when referring data input. A stream can be seen as a flow of data from a source to a sink. Therefore, while discussing the compressor and decompressor, saying data is streamed to the decompressor from the compressor does not imply a file as it can go directly. The original input *stream* to a compressor can be referenced to by the terms *unendoced*, *raw* or simply *original* data. As for compressed data, terms used are *encoded* or *compressed* and *bitstream*.

Other useful terms are *semiadaptive*, *adaptive* and *nonadaptive*. A nonadaptive compression method does not change its way of working based on

the data being compressed. An adaptive method on the other hand, is capable of changing its behavior based on the raw data. There are compression methods that do a two-pass processing of the data being compressed, where the first pass only collects statistics of the data and the last pass uses this data while doing the compression. This last method is known as semiadaptive.

Central terms in the literature of compression are *lossy* and *lossless* compression. Lossless compression preserves the original data, reproducing the exact original data after decompression. This type of compression is used on data that has to remain unchanged after decompression to be useful, examples are text files and source code, where changing only a bit can break its value. In contrast, lossy compression loses information and is commonly used on videos, images and sounds where loss is acceptable.

Finally, some terms describing the performance of the compression. The *compression ratio* is defined as [5]:

$$\text{Compression ratio} = \frac{\text{size of output stream}}{\text{size of input stream}}.$$

A value of 0.7 tells us that the data occupies 70% of its original size after compression. If the value is above 1 it tells us that the result is an expansion of the original data. The *compression factor* is the inverse of the the compression ratio, thus values greater than 1 indicate compression and values below 1 entail an expansion. The compression factor is defined as [5]:

$$\text{Compression factor} = \frac{\text{size of input stream}}{\text{size of output stream}}.$$

2.1.2 Run length encoding

A simple, yet sometimes efficient compression scheme is RLE. This scheme is most efficient when data elements occur in a contiguous order. RLE works as follows: Whenever an element e occurs n consecutive times, it is encoded with a repeat counter followed by the element such as ne. The repeat counter n is known as the run length, and the procedure just described is known as run-length encoding (RLE).

As an example use, this compression scheme is efficient for images with pixels that have same value in a contiguous pattern. The pattern to scan the image can of course be different than simply row by row, it is also possible to scan column by column, other scanning patterns are also possible. Which pattern is best suited is dependent on the data that is processed.

Earlier we presented the format of RLE as ne, but sometimes if consecutive values have different values, it is more efficient to mark the stream with an own counter for raw streams. Such that a stream of values that looks like $v_1v_2v_3v_4$ are encoded as $4_{raw}v_1v_2v_3v_4$, with one counter instead of a counter for each value. A counter for each value would often expand the data stream instead of compressing it, unless it has a majority of long consecutive values. This can easily be observed even for the tiny stream presented above, the result would be $1v_11v_21v_31v_4$. The size of the counter also has to be considered when deciding if is worth to code consecutive elements into the format ne. Let us say we have the string $aaabc$ and the size of the counter is four bytes, then it will be more space efficient to code this stream as $5_{raw}aaabc$ instead of $3a2_{raw}bc$.

2.1.3 Huffman coding

Huffman coding [6] is a widely used and known lossless compression method. It can be found as the only algorithm applied for compression, or as one of many methods used in combination to obtain a more compressed result. One such example is the use of Huffman to compress the result of the transformation done in the joint photographic experts group (JPEG) image compression.

The idea behind Huffman is to represent frequently occurring symbols with fewer bits than less frequently occurring symbols. The simplest way to describe the Huffman coding algorithm is by an example. If we consider five symbols s_1, s_2, s_3, s_4 and s_5 with occurrence probabilities: 0.46, 0.18, 0.18, 0.09 and 0.09. We can build a Huffman tree by merging the two smallest values, these can be chosen arbitrary if more than two values values fits the requirement. This will generate a new node with a value equal to the sum of its child nodes. In our example s_4 and s_5 are the two smallest values, they merge into a node with value $0.09 + 0.09 = 0.18$. This new node replaces the two smallest values it has merged. The next step in the building process now considers the remaining symbols and the newly created node. There are three instances of the minimum value at this point, we choose the newly created node with value 0.18 and a free node of value 0.18. At this point we have three nodes with the following values 0.36 (the newly created node), 0.18 and 0.46. There are no ambiguities at this point, and the two nodes with smallest values create a node with value 0.54. Then finally the root node is created out of two nodes with values 0.54 and 0.46. The resulting Huffman tree can be seen in Figure 2.1 to the left. Another possible Huffman tree of these probabilities is shown to the right in Figure 2.1. This version is built by selecting the nodes that are not in a subtree when it is possible to choose between free nodes and nodes of a subtree.

Decoding a Huffman encoded stream is also fairly simple, and is also best explained through an example. The method to generate the decoded stream encoded with the Huffman tree seen to the left in Figure 2.1 is as follows: First one simply reads the encoded stream from the beginning and traverses the Huffman tree with the values found in the bitstream, and whenever a leaf node is reached, start at the root again. Let us consider an encoded stream that looks like the following: 0101011011111110, with the first bit to the left. To decode this bitstream, one looks at one bit at a time while traversing the Huffman tree from the root down to the leaves. Which branch to take is given by the value of the bit. When a leaf is reached its symbol is emitted to the output stream. Since the encoded stream can only be interpreted in one way, there is no ambiguity of what symbol to emit.

For the given stream the first two symbols are found as follows: Starting at the root we follow the branch marked with a 0, which leads us directly to a leaf node with symbol s_1. Then the next bit in the bitstream is considered, starting at the root node, which leads us to the right, to node s_{2345}. This is not a leaf node, therefore the next bit in the bitstream is evaluated, and the result is a leaf node, s_2, thus this symbol is emitted to the output. If this is done with the whole bitstream the result is the following output: $s_1 s_2 s_2 s_3 s_5 s_4$.

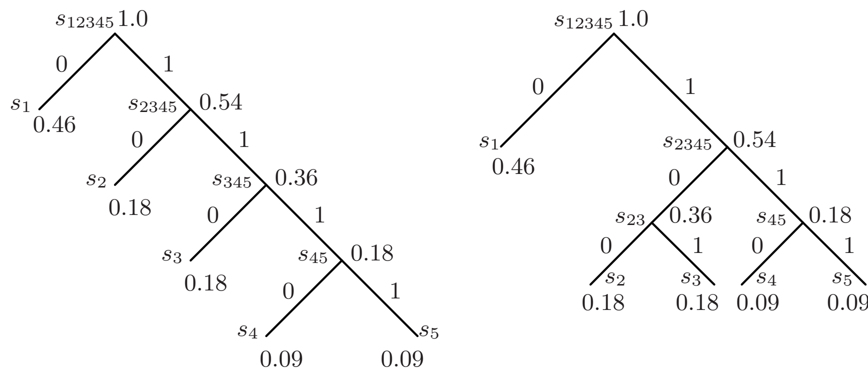


Figure 2.1: Two different Huffman trees for the probabilities s_1 to s_5

2.1.4 Parallel approaches

Huffman decoding and decoding of run-length encoded streams are algorithms that are sequential in their nature. While decoding a Huffman compressed bit string it is impossible to tell at which position the next symbol will occur without processing the previous bits. There are methods to parallelize decoding of Huffman bit strings, as will be discussed in Section **??**.

One simple method that does not scale well is to introduce an offset table before the bit string. This offset table could tell where the different blocks of output data can start their decoding within the encoded bit string. The down side is that this would also require storage and that it does not scale very well for many processing units. One reason is obviously the growth of the lookup table too support a given number of process, and too many entries would give an overhead so great that it would actually result in a growth in the data.

2.1.5 Image compression

The main idea of image compression is the same as for other compression scenarios, exploit the redundancy in the data to make it smaller. There are basically two ways of doing this on images, lossless and lossy. Given that noise to some degree is acceptable in images without perceptible visual artifacts, lossy compression is often used. The lossy compression schemes used on images uses a transformation to compact the information. Gathering the information into a small region due to decorrelation result in possibility to discard data with coefficients close to zero (Salomon [5]). As for lossless compression, it does not introduce any noise, but the compression ratio is not as good as the lossy compression.

The are three major steps in compressing an image with lossy compression:

1. Decorrelation (through a transformation)
2. Quantization
3. Entropy encoding

More details about these steps are given in Section 2.2.

The first step, decorrelation, is achieved through a transformation. If this transformation is separable it can be done in one dimension then in another, and produce the correct result. Separable filters are desirable due to their lower computational complexity. Details around separable filtering are given in Section 2.2.6.

The layout of the three-dimensional (3-D) seismic data is as depicted in Figure 2.2. It consists of two-dimensional (2-D) images stacked upon each other in the depth direction. Image compression for 3-D seismic data is based on the same methods as for 2-D images. Instead of using two passes with a separable filter, one for each dimension, there are three passes. First all the images in the stack are processed as the two dimensional case. Second

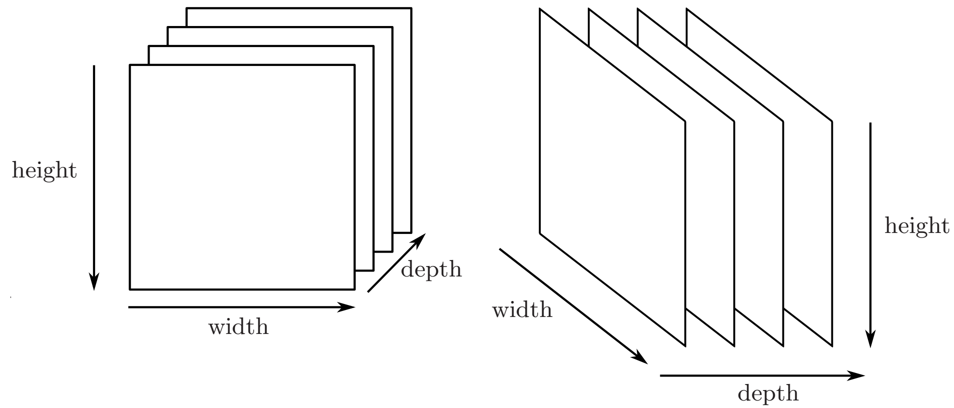


Figure 2.2: Arrangement of images in a stack for 3-D seismic data

the filter is applied on all the images in the depth direction. That is as interpreting the values in the depth and height directions as another 2-D image.

Compressing seismic data while considering three dimensions is of great advantage, because of the correlation that exists between the images in different directions. An example is the correlation that exists between values in the horizontal direction of seismic images both in depth and in width. As the changes in these directions are small and similar to each other, that is, slow varying.

2.2 Subband coding

The following sections will describe theory related to subband decomposition and coding. Starting out with decimation and interpolation in Section 2.2.2, followed by quantization in Section 2.2.3. Then in Section 2.2.4 and Section 2.2.5 representation of of these stages will be presented.

It can be shown that block transforms are a special case of filter banks that have filter length N, N channels, and a down-sampling by N.

2.2.1 Overview of subband coding

The encoding and decoding processes are presented in Figure 2.3 and 2.4. As can be seen both encoding and decoding is accomplished in three stages. Encoding starts with the analysis transformation, then quantization and finally the entropy encoding. As for the decoding this process is reversed, and the stages are entropy decoding followed by an inverse quantizer and at the final stage synthesis transformation.

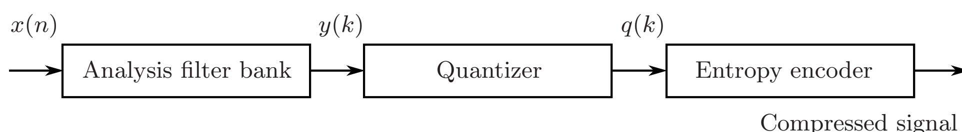


Figure 2.3: Subband encoding

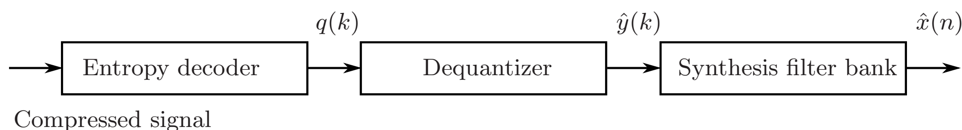


Figure 2.4: Subband decoding

2.2.2 Decimation and interpolation

Decimation and interpolation are operators that processes the samples of a signal. Decimation is the process of reducing the number of samples by an integer factor of M. This results in a reduced sample rate of the same factor. As for the interpolation operator, it does the opposite of the decimation operator, it increases the sampling rate by an integer factor of M. Both of these operations is done after a filter that usually is low-pass. The filter is followed either by a sub sampler, also called down-sample, for decimation, and for interpolation an up-sampler.

The up-sampler inserts zeros between the samples, such that $M-1$ zeroes are inserted between the samples. What the down-sampler does is to pick each Mth sample from the original stream, creating a new list of samples indexed 0 for the first sample and 1 for the Mth sample taken from the original stream, and so forth.

As we shall see later, these operators are used in the different stages of the transformation. The down-sampler in the analysis stage, and the up-sampler in the synthesis stage. They are depicted as $\downarrow M$ for the down-sampler and $\uparrow M$ for the up-sampler.

2.2.3 Quantization and inverse quantization

There are two well known quantization methods mentioned through out literature, scalar quantization (SQ) and vector quantization (VQ). We will only focus on scalar quantization in this text. SQ is a special case of VQ where the number of dimensions is equal to one. We will use scalar quantization and quantization interchangeably in the rest of the text, unless stated otherwise.

Quantization is a process that " '...restrict[s] a variable quantity to discrete values rather than to a continuous set of values' " as described by

Salomon [5] who refers to a dictionary. One way to quantize values is by finding the largest absolute value in the input, then use this when scaling the input data into the integer representation. Mapping into the signed integer range can be done with

$$\left\lfloor \frac{y(k)}{\max(\text{abs}(y(k)))} \times \frac{I}{2} \right\rfloor, \tag{2.1}$$

where I is a constant representing a value that is one greater than the maximal value of the integer representation. As examples of natural choices for the value of the constant I are byte (octet) representation[1] or word representation (here two bytes). This gives $I = 2^8 = 256$ for bytes and $I = 2^{16} = 65536$ for words.

Equation 2.1 will map the source data into the range of the integer representation[2], except at the positive end, where it will be possible to get an integer value 1 integer outside the range, as an example 128 is one outside the range of 8-bits signed values. If necessary this will be corrected in the quantization process when the index is calculated, as will soon be explained. This is the stage in the signal compression that introduces the loss of information by using coarser representation of the data values. Therefore it is important to set the quantization to adjust the desired compression ratio and thus the loss.

We use the mid-tread uniform threshold scalar quantizer as described in the PhD thesis of Røsten [2]. The quantizer is actually called a mid-tread uniform threshold scalar quantizer with dead-zone, but for convenience we will just present it as in the previous sentence. This scalar quantization has a dead-zone with a total width $T = 2\beta \times \Delta$ where $\beta > 0$, around zero. In the equation for T the Δ symbol represents the distance between quantizer decision levels, and is known as step-size. The variable β adjusts the size of the dead-zone, and for image compression $\beta = 0.5$ is often used, this gives no dead-zone. It is common to use $\beta = 0.6$ for compression of seismic data. Compression ratio below 1:10 requires a $\beta < 0.5$ to avoid too much quantization noise according to Røsten [2].

$$i = \begin{cases} I/2 + \lfloor (y(k) + T/2)/\Delta \rfloor, & y(k) \leq -T/2 \\ I/2, & -T/2 < y(k) < T/2 \\ I/2 + \lceil (y(k) - T/2)/\Delta \rceil, & y(k) \geq T/2 \end{cases} \tag{2.2a}$$

Equation (2.2a) gives the quantizer indices into the γ function of the quantizer. According to Røsten [2], SQ can be described as a non-linear

[1] An octet is 8 bites in size. The bytes in this text has the same size as an octet.

[2] Assuming two's complement.

mapping of $y_m(k) \equiv y(k) \in \mathbb{R}$ to a finite set $\gamma = \{\gamma(0), \gamma(1), \ldots, \gamma(I-1)\}$. Where the indices $i = 0, 1, \ldots, I-1$ of the $\gamma(i)$ function are called the quantizer indices and the results of the gamma function is the quantizer levels.

$$\gamma(i) = (i - I/2) \times \Delta \tag{2.2b}$$

In the subband coding system, quantization and inverse quantization is the stage just after the analysis filter bank and just before the synthesis filter bank, respectively (see Figures 2.3, 2.4 and 2.5). The output of the quantizer is denoted by $q(k)$ and the reconstructed signal by $\hat{y}(k)$. $q(k)$ can be seen in Figure 2.3 and $\hat{y}(k)$ in Figure 2.4 (the inverse quantizer is called "dequantizer" in this figure). The quantizer selects the index i according to Equation (2.2a), where I is even. The inverse quantizer finds the quantizer representation level by equation (2.2b). The dynamic range of the quantizer is given by

$$y(k) < -I/2 \times \Delta - T/2 \quad \text{and} \quad y(k) > (I/2 - 1) \times \Delta + T/2.$$

and is exceeded if any value is outside. If that happens the value of i should be replaced with $i = 0$ and $i = I - 1$ for the given equation, respectively.

2.2.4 Analysis stage

The analysis stage is where the input signal is decomposed into subbands. This results in a decorrelation of the signal as well as concentration of the energy into a minimum number of subbands [2]. After this stage quantization takes place as described above in Section 2.2.3. This introduces compression noise due to an approximation of the samples, since it is assumed that perfect reconstruction is possible. Consequently it is at the quantization stage that loss is introduced to the compressed signal.

Our subband coding scheme is based on the works of Røsten [2] and will now be described. It consists of M-channel parallel-structured uniform filter banks with non-unitary linear-phase near-perfect reconstruction (PR) properties for both analysis and synthesis filters. An illustration of such a system with M-channels can be seen in Figure 2.5. The number of subband filter banks, M, is equal to eight, and the number of taps (denoted L) is 32. If given an one-dimensional (1-D) input by $x(n)$ for $n = 0, 1, \ldots, N-1$ the uniform analysis filter bank will produce a decomposition into M subbands with K subband samples in each.

The analysis filter is denoted by $h_m(l)$ for $m = 0, 1, \ldots, M-1$ and $l = 0, 1, \ldots, L-1$ and the subband signals by $y_m(k)$ for $k \in \mathbb{N}$. The function

for reconstructed signals is denoted by $\hat{y}_m(k)$. For the synthesis filter the function is given by $g_m(l)$. The filters for analysis and synthesis is given by the following equations (which can be found in Røsten [2], and Ramstad et al. [7])

$$y_m(k) = \sum_{n=-\infty}^{\infty} h_m(kM - n)x(n) \tag{2.3a}$$

and

$$\hat{x}(n) = \sum_{m=0}^{M-1} \sum_{k=-\infty}^{\infty} g_m(n - kM)\hat{y}_m(k), \tag{2.3b}$$

respectively.

For 2-D and 3-D subband decomposition and reconstruction separate filtering in each dimension is performed. For the 2-D case this can be done by first filtering row-wise then column-wise, or the other way around, at the analysis stage, and for the synthesis stage the ordering of filtering is reversed. The 3-D case is similar to the 2-D case just with an expansion of one more dimension, see Section 2.1.5. For details concerning separable filters see Section 2.2.6.

When doing subband decomposition, expansion of the signal is prevented by adhering to three constraints. Firstly, the length of the input signal, N, divided by M must give K where K is the number of samples in a subband, ideally this should be equal for all the subbands. Secondly, extension of the input signal at the edges has to be considered. Thirdly, the subband samples has to be critically down-sampled by M. The result of following these constraints is a reconstructed signal $\hat{x}(n)$ that has the same length as the original $x(n)$. Furthermore, it gives a maximally decimated filter bank system that has the property $K \times M = N$. More details of the second constraint is given in Section 2.2.7.

2.2.5 Synthesis stage

In this stage signals are reconstructed from subbands into the original signal, if the signal from the analysis stage is used without modification near-PR is achieved. Loss of precision is mainly due to the quantization that introduces noise, as mentioned earlier.

Otherwise the synthesis stage is basically equal to the analysis stage, except for different filter and transformation function. The equation for the synthesis filter banks is given in Equation 2.3b.

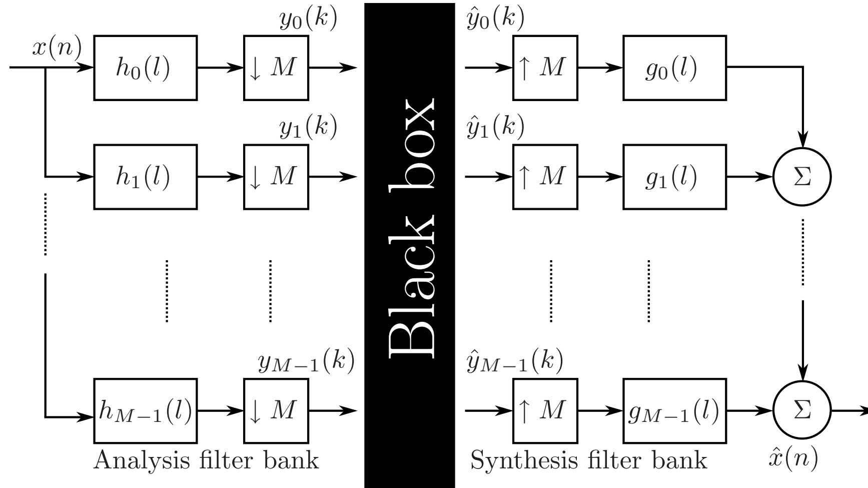


Figure 2.5: Figure adapted from Fig. 1.10 and Fig. 2.5 in [2]. Overview of a M-channel maximally decimated filter bank system with a black box in the middle.

2.2.6 Separable filters

Applying filter transformation to a 2-D image can be done in two ways, as a convolving mask or as two separate transformations one in the horizontal direction followed by one in the vertical direction, or vice versa. This last method works on filters that are separable and has great benefits with respect to amount of calculations performed. Let us consider the Sobel operator as described by Gonzalez and Woods [8]. The filter mask of the Sobel operator with size 3×3 is as shown in Table 2.1.

Table 2.1: Sobel operator of size 3×3.

-1	0	1
-2	0	2
-1	0	1

If we use the method of spatial filtering, described by Gonzalez and Woods [8], of an image of size $M \times N$ and a mask of size $m \times n$. The transformed image is given by,

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t) \tag{2.4}$$

where, $a = (m-1)/2$ and $b = (n-1)/2$. Equation 2.4 has to be applied for

all the values of x and y in the image, that is for $x = 0, 1, \ldots, M-2, M-1$ and $y = 0, 1, \ldots, N-2, N-1$. As can be seen the number of multiplications for each element is $m \times n$. This filter mask can also be written as a combination of two vectors multiplied together. If denoted as v for vertical and h for horizontal they may be represented as,

$$v = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \quad h = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \tag{2.5}$$

This equation shows how a separable 2-D filter can be decomposed into two vectors. Now, it is possible to transform an input image with the Sobel operator by first doing a vertical transformation then a horizontal transformation. Doing the transformation this way results in $m + n$ multiplications per transformed element. Thus, it is easy to see that the amount of calculations needed to do the transformation is drastically reduced with separable filters. The amount of calculation to filter an image without using separable filters is $MNmn$ versus $MNm + MNn = MN(m + n)$ for separable filters.

A figure illustrating the process of doing filtering in two separate steps, first horizontal then vertical is seen in Figure 2.6.

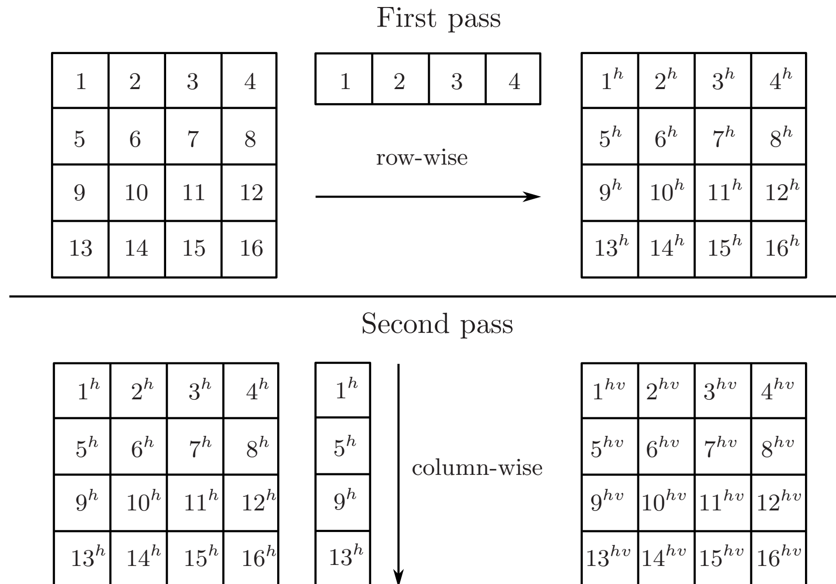


Figure 2.6: Separable filter over a 2-D image

2.2.7 Filter extension

To preserve the perfect reconstruction (PR) property of a signal in an analysis-synthesis filter bank, care has to be taken at the boundaries of a signal. This is due to the the overlapping of unit pulse responses of both the analysis and synthesis filter channels. The reason is that signal segments are reconstructed with an added influence from adjacent signal parts [7].

The solution is extension of the signal, according to Ramstad et al. [7], there are only two known methods of extending the finite length input signal while still preserving the PR property. This without generating additional information to be sent along with the signal. The methods are known as, circular extension and mirror extension.

Circular extension is achieved through repeating the finite input signal at its extremities. Given an input signal with a length of K samples, its extended signal will have a periodicity of K. As pointed out by Ramstad, et al. [7] it can be proved that the periodic property of an input signal is preserved after time-invariant linear filtering. Thus, each channel signal have a period of K before decimation. The period after after decimation is given by

$$K = pN, \tag{2.6}$$

where N is the decimation factor and p is the period for each of the subband signals. Furthermore, given that the decoder has to know each infinite subband signal for perfect reconstruction, which is fulfilled through the periodicity p of each subband, it is thus sufficient to transmit p samples for each subband [7].

Finally, we take at look at the mirror extension method. This method is similar to that of circular extension with a little twist, the signal is first mirror reflected at one endpoint, then periodic extensions are performed at the signal that now has double length. The benefits of mirror extension compared to circular extension is the avoidance of discontinuities present in circular extension [7].

Instead of taking advantage of periodicity, the mirror extension preserves the symmetry on both sides of the mirror points. As stated by Ramstad et al. [7], if a linear phase filter is applied to a symmetric signal $x(n)$ the output is symmetric. In general if an input signal have the same symmetry as the filter, symmetric or not, the result is symmetric, and if they differ the result is anti-symmetric. This same relation is valid for whole-sample symmetry and half-sample symmetry. Half-sample symmetry is the case when the symmetry is between two samples contrary to whole-samples where the symmetry is at a sample.

Let us consider an input signal of length 16, this should be filtered with

an even-length symmetric filter, h. Assume that the filter has half-sample symmetry and a length of four. Extension of the input signal can be done with either whole- or half-sample symmetry. Since we want an output that has whole-sample symmetry we should exploit the facts mentioned in the previous paragraph. Thus, we do a half-sample expansion of the input signal, giving us two half-sample sources which results in a whole-sample output. An illustration showing how this might look, is given in figure 2.7. The figure illustrates the extended input and the result before any decimation is performed.

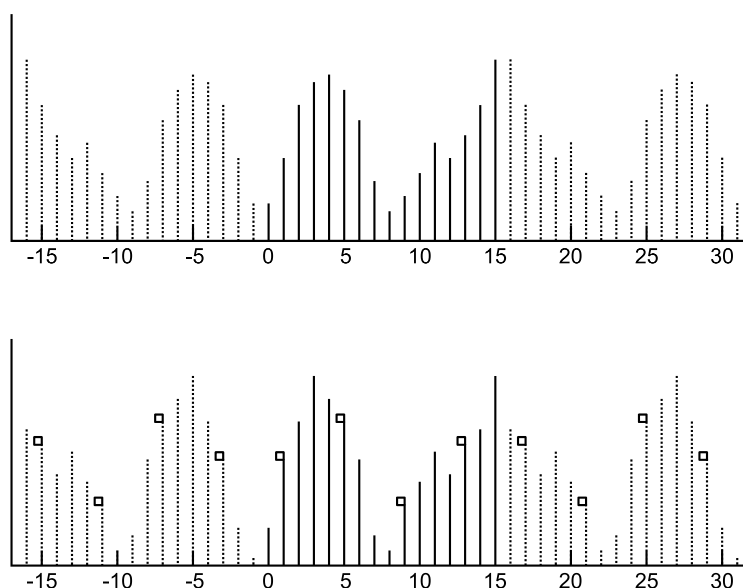


Figure 2.7: At the top the extended input, and at the bottom the filtered output.

The filtered signal now has 17 distinct values. Since the critical decimation with factor N result in a transfer of $16/N$ out of the 17 samples, care has to be taken while choosing the samples. Ramstad et al. [7], gives two criteria that has to be fulfilled: First, avoid picking whole-sample symmetry samples, that is -1 and 17. Second, it is important that the samples on the opposite side of the symmetry points have the correct distance. If the correct samples are chosen they have the same value at both sides.

2.2.8 The "black box" stage

Within the "black box" two sub-stages takes place, quantization (Section 2.2.3) and entropy coding. The entropy coding is to reduce the data amount

used to represent the information. It may be a single method such as Huffman coding, arithmetic coding or simply RLE, or a combination of several compression methods. The entropy encoding and decoding takes place after the quantization, and before the inverse quantization, respectively. Details about these compression methods was given in Section 2.1 and quantization was presented in Section 2.2.3.

Chapter 3

GPU programming

Since the the topic is decompression of seismic data on GPUs we will look at the architecture of modern GPUs, focusing on the NVIDIA Tesla architecture. Given that the application of interest is not designed with a CPU in mind, description of the CPU architecture is not described here. Section 3.1 and 3.2 and its subsections is taken from an earlier work of mine [9], and contains minor changes.

3.1 NVIDIA's Tesla architecture

To get a better understanding of how the GPU works, a presentation of the NVIDIA Tesla architecture will be given, based on [10] and [11].

Within Tesla based GPUs you will find groupings of texture/processor clusters (TPCs). Within a TPC you will find 2 streaming multiprocessors (SMs). Further, inside a SM there are 8 streaming-processors (SPs) cores. An overview figure of this architecture can be seen in Figure 3.1, and more detailed figures of the TPC (Figure 3.2) and SM (Figure 3.3). At the highest abstraction level we find the streaming processor array (SPA), which contains all from one TPC and up wards. As an example the NVIDIA QuadroFX 5800 has 240 SPs and 30 SMs.

The TPC contains the following elements: a geometry controller, a streaming multiprocessor controller (SMC), two streaming multiprocessors (SMs), and a texture unit (see Figure 3.2). The most interesting parts within a TPC for us is the SMs. Inside the SM you will find an instruction cache, a multithreaded instruction fetch and issue unit (MT issue), a read-only constant cache, 8 SP cores, 2 special-function units (SFUs), and a 16 kilobytes of read/write shared memory, shown in Figure 3.3).

A SP core contains a scalar multiply-add (MAD) unit, resulting in eight

MAD units for a SM. For transcendental functions and attribute interpolation the SFU is used. Each SFU contains four floating-point multipliers. The texture unit can be used as a third execution unit by the SM within the TPC. The SMC and raster operation processor (ROP) units implement external memory load, store as well as atomic access. Between the SPs and the shared-memory banks there is a low-latency interconnect network providing shared-memory access.

The SM is hardware multithreaded to be able to execute several hundreds of threads in parallel while running several programs. The number of threads that can be executed concurrently in hardware with zero overhead for a SM, varies from 768 to 1024 with compute capability 1.0 and 1.2 respectively [11].

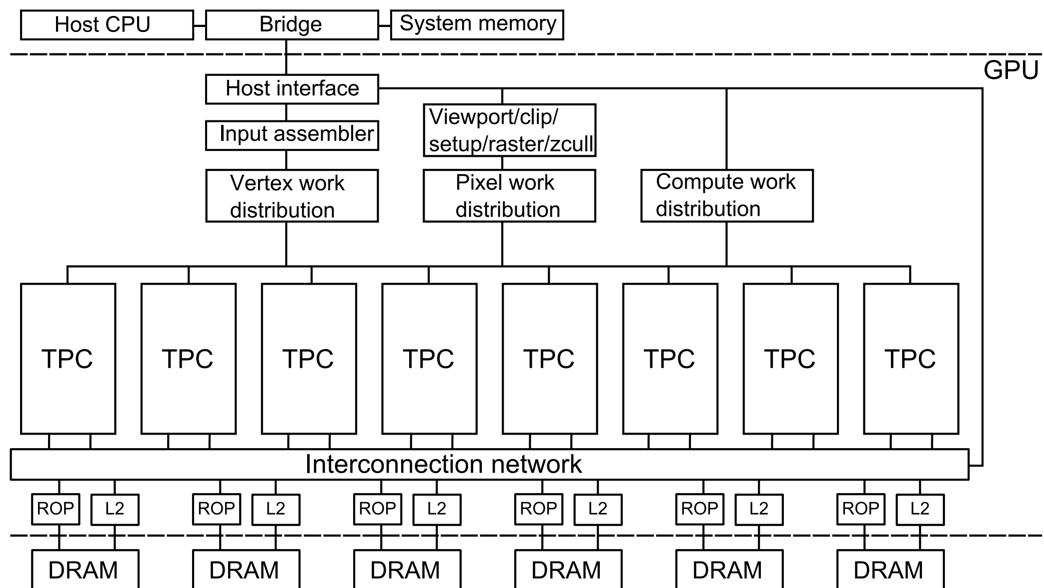


Figure 3.1: Figure of the Tesla architecture adapted from [10]

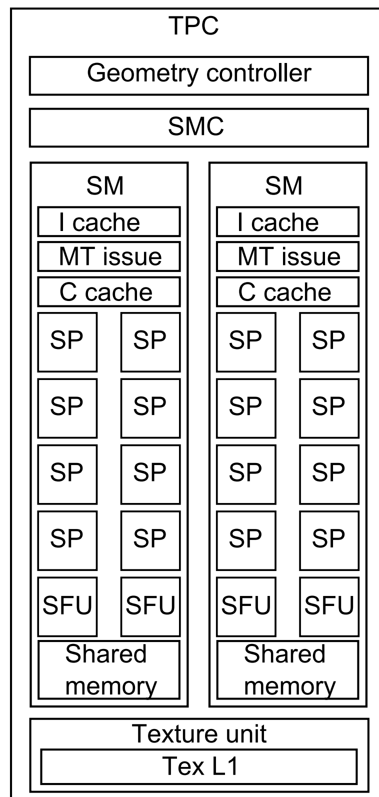


Figure 3.2: Figure of the TPC from [10]

The SM in the Tesla architecture uses what NVIDIA calls single-instruction, multiple-thread (SIMT). The SM's SIMT multi-threaded instruction unit's responsibility is creating, managing, scheduling and executing threads. Threads are executed in groups of 32 parallel threads known as warps. Creation of threads is lightweight, as is fast barrier synchronization between threads, which can be issued with an instruction. This gives a very efficient and fine-grained parallelism. Each SM manages a pool of 24 warps, with a total of 768 threads, or 32 warps with a total of 1024 threads for compute capability 1.2 or higher, an example is GeForce GTX 280 [11], we will assume 24 warps in a pool for the rest of the document, unless stated otherwise. The SM selects one of the warps, in the pool of 24, to execute a SIMT warp instruction, each cycle. A warp instruction issued is executed as two sets of 16 threads over a period of four processor cycles. It should be noted that the SP cores and the SFU units executes instructions independently, so by issuing instructions between them on alternate cycles, it is possible for the scheduler to keep both working. The choice of warp is based on a scoreboard that qualifies each warp every cycle. Warps that are ready is prioritized by the instruction scheduler, it then select the one with highest priority for issue. Prioritizing is based on warp type, instruction type, and "fairness" to executing warps within the SM.

Memory instructions provided by the Tesla architecture are of the type load/store. These instructions use integer byte addressing and registers with offsets through address arithmetic. There are three kinds of memory spaces accessible through these load/store instructions: local memory, shared memory and global memory. The properties of the different memory spaces gives varying performance, care has to be taken to utilize the correct memory space for optimal performance. We will consider this aspect in greater detail later, when we look at coalesced memory access. Each of the memory spaces have their own instructions for load and store, they are load-global, store-global, load-shared, store-shared and load-local, store-local. Memory bandwidth is improved by coalescing load/store instructions when accessing global and local memory.

3.2 NVIDIA CUDA

NVIDIA compute unified device architecture (CUDA) was introduced by NVIDIA to allow programmers access to the graphics hardware without going through a graphics application programming interface (API), such as OpenGL or DirectX. It is a programming model that extends the C programming language through the use of special declarations and an API. The ap-

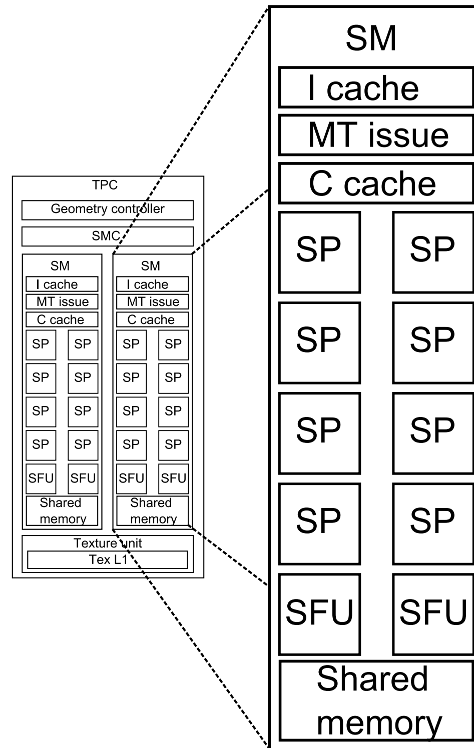


Figure 3.3: Figure of the SM from [10]

plication is built on top of a NVIDIA CUDA driver that communicates with the targeted device. Over this driver there are abstractions, such as NVIDIA CUDA runtime and NVIDIA CUDA libraries. NVIDIA CUDA runtime is an abstraction that simplifies the programming as is the NVIDIA CUDA libraries. The libraries include CUFFT and CUBLAS, that implements fast Fourier transform (FFT) and basic linear algebra subprograms (BLAS) respectively.

3.2.1 NVIDIA CUDA extensions to the C programming language

Programming C for CUDA provides some extension to the C language:

- Function type qualifiers
- Variable type qualifiers
- Kernel execution directive

- Built-in variables

Function type qualifiers specify if a function executes on the host, or on the device. It also specifies if it is callable from the host or the device. The qualifiers are **__device__,** **__global__** and **__host__**. Functions having **__device__** qualifier is only callable from the device, and executes on a device. In contrast to those having **__global__**, they are callable only from the host, but executes on the device. Finally, the code that are handled only by the host have **__host__** as a qualifier, or simply no qualifier. It is possible to combine **__host__** with **__device__**, in which case code for both host and device is compiled.

Variable type qualifiers specify where a variable is to reside in memory. They are **__device__,** **__constant__** and **__shared__**. For variable type qualifiers as with function qualifiers the **__device__** specifies that the variable shall reside on the device. In addition to this qualifier it is possible to specify which memory space on the device, being either **__constant__** or **__shared__**.

The execution configuration specifies how the kernel is executed on the device from the host. It is specified by the use of

```
¡¡¡DimGrid, DimBlock, NumSharedMem, Stream¿¿¿
```

Both `DimGrid` and `DimBlock` are of type `dim3`, it has three members: `x, y` and `z`. `NumSharedMem` is of type `size_t` and `Stream` of type `cudaStream_t`. `DimGrid` specifies the dimension of the grid, that is the number of blocks. `DimBlock` specifies the dimension of each block in the grid, that is the number of threads per block. `NumSharedMem` specifies the number of bytes in shared memory that is dynamically allocated. Finally, `Stream` specifies the associated stream, default is 0. An example of calling a function is given in listing 3.1.

The built in variables are the following:

- `gridDim` of type `dim3`, holds the dimensions of the grid.
- `blockIdx` of type `uint3`, a vector type with the components accessed through `x, y` and `z` as with `dim3`. It has the block index within the grid while running a kernel.
- `blockDim` is of `dim3` and holds the dimensions of a block, and thus the number of threads.
- `threadIdx` of type `uint3` contains the thread index within a block.
- `warpSize` is an `int` type containing the size of the warp in number of threads.

Listing 3.1: Calling a NVIDIA CUDA kernel

```
__global__ void foo(float *arg); // prototype of foo
foo<<<DimGrid, DimBlock, NumSharedMem, Stream>>>(arg);
```

These built-in variables cannot be assign values, and it is not allowed to take the address of them.

3.2.2 NVIDIA CUDA's memory hierarchy

Knowing the memory hierarchy is of great importance to able to write efficient code with NVIDIA CUDA. Since there are no cache on the local memory or global memory, accessing these gives a penalty between 400 and 600 clock cycles of memory latency.

The hierarchy is as follows [11]. Each thread has a per-thread local memory, each block contains a shared memory seen by all threads in the block, having a lifetime as long as the block. Then there is global memory accessible by all threads. In addition to these, there are special type of memory, known as texture and constant memory, both are actually constant. All of the mentioned memory spaces are optimized for different purposes. Texture memory for instance, offers different addressing modes, and it also has data filtering support for some specific data formats.

To maximize memory bandwidth, it is crucial to access the underlying memory hierarchy in the correct manner. If possible, for global memory what is called coalesced memory access should be used. Shared memory access should be done without bank conflicts to avoid reduced bandwidth [11]. Details around how this is done follows in the subsequent sections.

3.2.3 Shared memory

Because of the limited number of registers, 8192 for devices with compute capability below 1.2 and 16384 for devices supporting 1.2. This is the number of registers for each multiprocessor, in addition to this there are 16 kilobytes of shared memory for each multiprocessor. This memory is organized into 16 banks for devices of compute capability 1.x. Accessing different banks can be done simultaneously, therefore accessing n different addresses falling into n different banks yields bandwidth that is n times that of one single memory module (bank).

If bank conflicts occur, those addresses that map to same bank are serialized. This is done by the hardware, and results in as many separate conflict-free requests as necessary. The number of separate memory requests, if there

are n of them, is called a n-way bank conflict. Consecutive 32-bit words in shared memory goes into subsequent banks, and each bank has a bandwidth of 32-bits per two clock cycles.

Further, devices having compute capability 1.x have warp size of 32, and the bank count is 16. When a warp issues a memory request for shared memory, is it split into two request, one for each half-warp. Handling the first half-warp then the second, thus there are no bank conflicts between threads in the two half-warps.

3.2.4 Global memory

Due to the importance of utilizing the memory when doing high performance computation on GPUs, the coalescing of memory access will be described [11]. There are differences of the first NVIDIA CUDA capable devices and the new ones, classified by what is called *compute capability*. Devices with compute capability 1.0 and 1.1 are more restricted than that of 1.2 or higher when it comes to coalesced memory access.

The implementation was written with the strictest of the coalesced memory access patterns in mind, such that devices with compute capability below 1.2 and those compatible with 1.2 should be able to make use of coalesced memory access. Even if the kernels was designed to follow the strictest pattern as best as possible, both the access patterns are presented, that is for devices below and those including and above 1.2. The latter to show how it eases the way to get coalesced memory access on newer devices.

Now, coalesced memory access is presented based on NVIDIA's CUDA programming guide [11]. Coalesced memory access makes what could be several single memory transactions into one single memory transaction. First, devices with compute capability below 1.2 is described, followed by those including and above 1.2.

Compute capability below 1.2 Three conditions have to be satisfied for global memory access to be coalesced into one or two accesses. Coalescing is valid for all the threads within a half-warp if the following three conditions is fulfilled.

It is a requirement that the threads access either, 32-bit, 64-bit or 128-bit words. The latter case gives two memory transactions each of 128 bytes. Further all the 16 words that are accessed has to lie in the same segment or twice size for the 128-bit case. According to the programming guide for NVIDIA CUDA [11], the global memory is partitioned into segments that are of size 32, 64 or 128 bytes, and aligned to those sizes. The third condition

that has to be satisfied is that the threads accesses the words in sequence. Which means that the ith thread in a half-warp has to access the ith word.

If not all of the above conditions is satisfied, a memory access is issued for each of the threads. Accessing words of greater sizes reduces the bandwidth, for example accessing 64-bit words gives reduced bandwidth compared to 32-bit words, and so on. Figure 3.4 shows a coalesced memory access on the left side, and the right side shows a non-coalesced memory access.

Compute capability 1.2 and above Now, that the coalesced memory access conditions for compute capability 1.2 and below has been described, it is in place do describe that of compute capability 1.2 and above.

Coalesced memory access to global memory occurs for a half-warp whenever the words accessed by all the threads lie in the same segment. The segment has to be of size 32 bytes, 64 bytes and 128 bytes, for accesses to respectively 8-bit, 16-bit and for the last case 32-bit or 64-bit words, it is assumed that each thread accesses the the same word size.

The access pattern for addresses requested for a half-warp is not restricted, it is even possible for multiple threads to access the same address. Clearly this is not as strict as for devices of lower compute capabilities. An example of this can be given as follows: A half-warp addresses words in n different segments, this results in n memory transactions for devices of compute capabilities above 1.2. Now, devices with compute capabilities below that, issues 16 different transactions, which occurs as soon as n is above 1.

Even if not all words in a segment is used, all words are read. To reduce the waste of memory bandwidth, the smallest segment that contains the requested words is chosen. So if all the words lie in one half of a segment, and there exists a segment half of the original, the smaller one is chosen for transaction. Figure 3.5 shows different scenarios for devices of compute capabilities above 1.2.

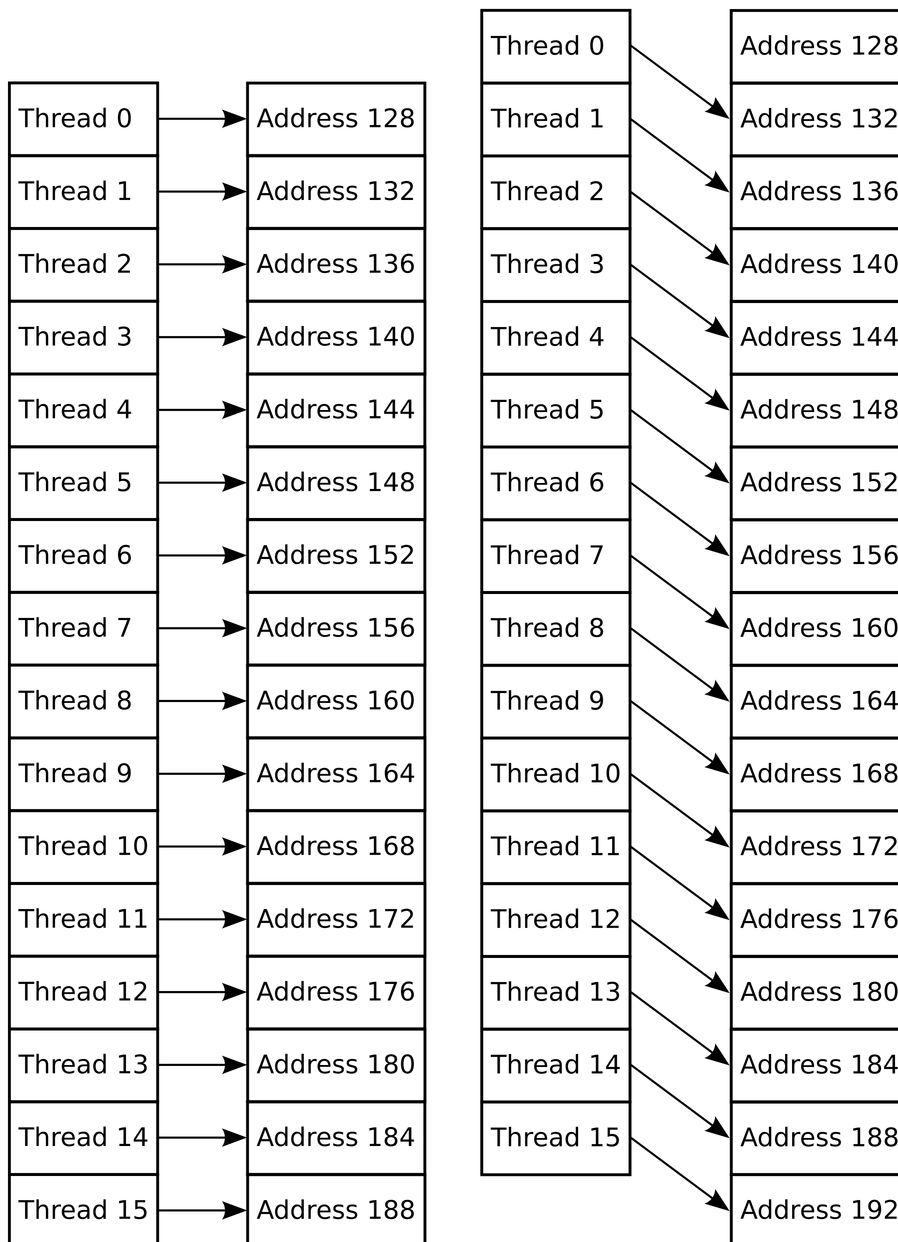


Figure 3.4: Coalesced memory access versus non-coalesced memory access for devices with compute capability 1.0 and 1.1, [11].

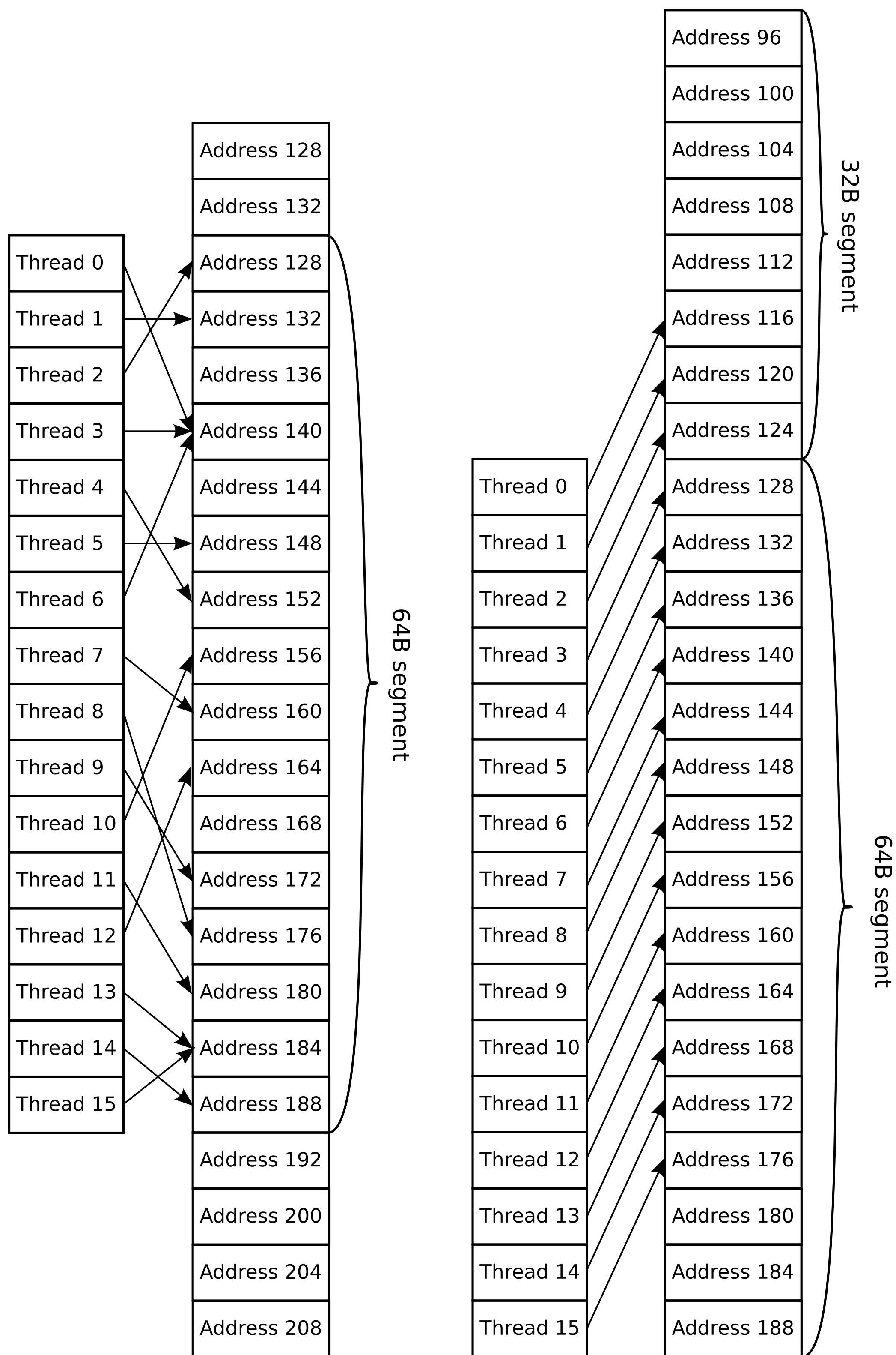


Figure 3.5: Coalesced memory access patterns for compute capability above 1.2

Chapter 4

Methodology

This chapter describes details concerning the implementation of different parts of the system. Starting with RLE in Section 4.1, then subband transformations in 4.2. Furthermore in Section 4.3 the Huffman implementation details are presented. Then the transpose functions are presented in Section 4.4.

4.1 Run-length encoding implementation

In this section we will present the GPU implementation of RLE decoding. It is a fairly straight forward implementation of RLE with minor modifications to speed it up on GPUs.

4.1.1 Layout of the RLE data

The format of the RLE is as described in Section 2.1.2. That is, a counter followed by a single value or a number of different values. The implemented RLE encoder uses a 32-bit type. The counter either gives the number of times to repeat a value or the number of following bytes that should be copied to the output stream. This is marked by the counter by having a positive value if the next byte is to be repeated, and a negative value if the following bytes are to be copied directly to the output, the absolute value gives the actual count.

Furthermore the modification done to the RLE-decoding on the GPU is the addition of a table with offsets into the input stream where the decoding can start. The motivation for this offset table is mainly to increase performance. A run-length encoded stream has to be decoded from the start because there are no way of telling where a counter starts without following

the counters from the start of the input.

To distribute the workload of decoding a RLE encoded stream among several processors the encoded stream is partitioned into smaller sections. This allows the different processors to work on their section of the encoded stream and produce their own output section. The decomposition of an encoded stream is such that the sections produced by each processors are about the same size. This is achieved by choosing the counters that are close to the given positions in the original stream. If we have a table with two offsets, we would start at the beginning of the encoded input stream and at a position in the encoded stream that would start writing close to the middle of the decoded output stream.

The offset table contains three variables for each entry: *input position*, *output position* and a *tag count*. The input position gives the offset in number of bytes from the beginning of the encoded stream. The output position gives the offset in number of bytes from the beginning of the decoded stream. Finally, tag count gives the number of the tag from the beginning of the encoded stream. This last variable is used to keep track of the extent of the section being decoded, by knowing the tag count of the next section, decoding can proceed until the tag count of the section being decoded equals the tag count of the next section. A pseudocode of the RLE-decoding can be seen in Algorithm 4.1.1.

4.1.2 The RLE decoding kernel

Instead of using branches to select which section of the encoded stream a group of threads should handle, the thread number is used to select the correct section. The kernel is designed to handle a RLE-stream that is divided into eight parts. To be able to fully utilize coalesced memory accesses it has 128 threads assigned to it. This way each section has 16 threads available to utilize coalesced memory access while reading or writing to global memory.

The partitioning is as follows: First, the thread ID is shifted to the right such that the three most significant bits of the maximum number of threads in a block, here 128, can be found as the three least significant bits. Then a mask is used to ensure that the only valid values are in the range 0 to 7. The result is that thread IDs in the range 0–15 belong to section 0, thread IDs in range 16–31 in section 1 and so on. An illustration of this scheme is given in Figure 4.1. Figure 4.1 illustrates how the binary number with range 000_2 to 111_2 maps to the different sections in the decoded stream[1].

[1] We denote the radix by subscript, e.g. 11_2 is 3 (decimal) in radix 2, and assume radix 10 as the natural radix (decimal).

Algorithm 4.1.1: RLE-DECODE($input, output, lenOut, threadID$)

```
local currentPos, posOut, startTag, stopTag, currentTag

(currentPos, posOut) ← GetPositions(input, threadID)
(startTag, stopTag) ← GetTagNumbers(threadID)

repeat
  comment: Read counter from input stream.
  count ← getCount(input[currentPos])
  if count ≥ 0
    then { symbol ← getNextSymbol()
           for i ← 1 to count
             do { write(output[posOut], symbol)
                  posOut ← posOut + 1
    else { comment: Copy count symbols from input to output.
           copy(output[posOut], input[currentPos], abs(count))
  currentPos ← currentPos + abs(count)
  currentTag ← currentTag + 1
until currentTag = stopTag
```

Starting with the initial length at the top, where binary numbers starting with a zero as the leftmost digit handles the first part of the output stream. Furthermore binary numbers starting with 00 handle the first quarter of the output stream. At the bottom of the figure, section 0 to 1 and section 1 to 2 are handled by binary numbers 000 and 001.

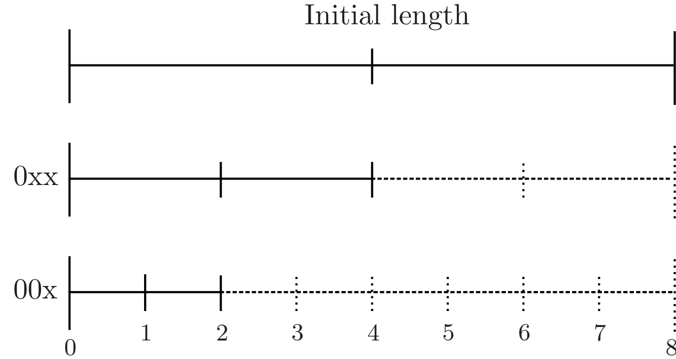


Figure 4.1: Illustration of division based on a binary number

4.2 Subband transform implementation

The kernel of the implementation doing the most compute intensive task, subband transformation, is presented in this section. There will be given a thorough description of how it was designed to gain the performance it has. First the serial implementation of the synthesis stage in the subband coding (CPU) is briefly described, then the conversion to a parallel version is given (GPU).

4.2.1 Description of the implementation

First, the serial implementation is described to give a natural transition for the parallel implementation, and because the serial version maps closer to Equation 2.3b than the parallel version.

A pseudocode of the serial version is given in Algorithm 4.2.1, this algorithm gives an overview of the synthesis stage. Pseudocode for the *upSample-AndFilter* function is given in Algorithm 4.2.2 The coefficients used is given in Appendix A, the algorithms presented use the coefficients for synthesis which can be see in Table A.2 and A.4.

The 1-D subband synthesis algorithm starts by padding (mirroring) the input signal at both ends, that is, at the start and at the end. The procedure

Algorithm 4.2.1: SERIAL-SB-SYNTHESIS(*subbands*, *dir*, *output*)

```
comment: Using 0-based indexing.
local paddedSB, coefs

comment: Process all the 8 subbands.
for i ← 0 to 7
  do
  { if (i mod 2 = 0)
  {   then evenFilter ← true
  {   else evenFilter ← false
  { if evenFilter
  {   then paddedSB ← doPadding(subbands[i], evenPadding)
  {   else paddedSB ← doPadding(subbands[i], oddPadding)
  {
  { comment: Get coefficients and do the up-sampling and filtering.
  { coefs ← getFilter(i, SYNTHESIS, dir)
  { output ← upSampleAndFilter(coefs, evenFilter, paddedSB)


procedure doPadding(SB, type)
 local padded

 if type = evenPadding
        { padded[0] ← SB[1]
   then { Copy(padded[1], SB)
        { padded[length(SB) + 1] ← SB[length(SB) − 2]
        { padded[length(SB) + 2] ← SB[length(SB) − 3]
        { padded[0] ← −SB[0]
        { padded[1] ← 0
   else { Copy(padded[2], SB)
        { padded[length(SB) + 2] ← 0
        { padded[length(SB) + 3] ← −SB[length(SB) − 2]
        { padded[length(SB) + 4] ← −SB[length(SB) − 3]

 return (padded)
```

Algorithm 4.2.2: UPSAMPLEANDFILTER($coefs, evenFilter, paddedSB$)

```
comment: The subband coding synthesis function.
comment: DS – down-sample size.
comment: FH – half the size of the filter.
comment: pSB – Alias of paddedSB.
comment: c – Alias of coefs.

local n, s, i, data, c, pSB, DS, FH
c ← coefs
pSB ← paddedSB
DS ← 8
FH ← 16
i ← 0

if evenFilter = true
  then {
    comment: Iterate over the samples in paddedSB.
    for n ← 0 to LENGTH(paddedSB) − 3
      do {
        comment: Loop down-sample number of times.
        for s ← 0 to 7
          do {
            data[i] ← data[i] + pSB[n + 3] × c[s]
            +pSB[n + 2] × c[s + DS]
            +pSB[n + 1] × c[FH − 1 − s]
            +pSB[n] × c[FH − 1 − s − DS]

            i ← i + 1
          }
      }
  }
  else {
    for n ← 0 to LENGTH(paddedSB) − 3
      do {
        for s ← 0 to 7
          do {
            data[i] ← data[i] + pSB[n + 3] × c[s]
            +pSB[n + 2] × c[s + DS]
            −pSB[n + 1] × c[FH − 1 − s]
            −pSB[n] × c[FH − 1 − s − DS]

            i ← i + 1
          }
      }
  }

return (data)
```

doPadding in Algorithm 4.2.1, presents pseudocode for how this is done, more details about the mirroring at the ends is given in Section 4.2.2. Then after the padding, the correct filter depending on the direction as well as the subband, is chosen. Finally, the up-sampling and filtering is done in the procedure upSampleAndFilter.

The output produced by the inner-loop of upSampleAndFilter is depicted in 4.2. Subband denotes the input signal starting at index 0, the old data and new data is the value in the output before and after the upSampleAndFilter function has been applied. This figure shows how the output is generated when $n = 0$ for the inner-loop variables $s = 0$ and $s = 1$. The values in indices 0 to 3 are multiplied by different coefficients for different values of s of the inner-loop. These products are then summed into one scalar that is added to a scalar from the data output producing the final value that is written back to the data output. Each iteration of the inner-loop increases the index of the data output, thus updating one index for each iteration. As an example, a subband with length 16 (without the padding) would produce $16 \times 8 = 128$ elements in the output. As can be observed upSampleAndFilter is called 8 times from the function Serial-SB-Synthesis, this entails that the output is a summation of the results from each subband. Thus each element in the output is a summation of 8 values.

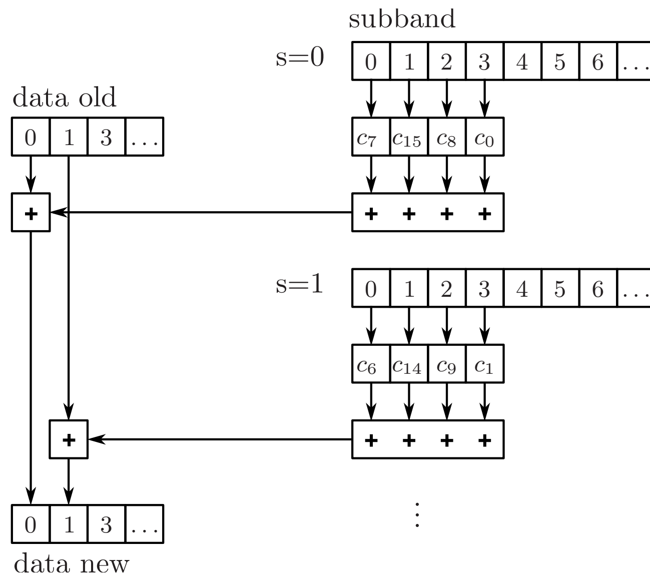


Figure 4.2: Illustration of the convolution step for $n = 0$ when $s = 0$ and $s = 1$.

Description of how the coefficients are connected to the respective indices in the inner-loop is now presented. Figure 4.3 illustrates how the subband

indices after up-sampling are related to the coefficient indices in the filter. Here, indices 0, 8, 16 and 24 in the subband has values, because up-sampling inserts zeros, thus these are the interesting indices. The first index uses the first 8 coefficients of the filter, and the next index of the subband uses the next 8, and so on. The last row illustrates in which order the filter coefficients are used with the subband value. By shifting the last row to the left, one index at the time, one can follow the horizontal arrow from the index at the top row through the middle row down to the value that will be used at that iteration of the inner-loop. Thus, if we say we are at the third round in the inner-loop of Algorithm 4.2.2, then the left-most value at the bottom row would be 5, and therefore index 0 in the subband would be multiplied by the coefficient at index 5 in the filter. Likewise index 8 in the subband by the coefficient at index 13 in the filter, and the same for the two other indices.

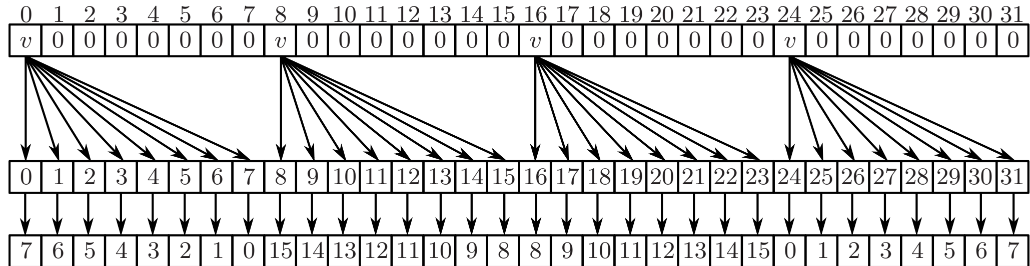


Figure 4.3: Relationship between subband indices and coefficients when doing synthesis filtering.

Now, as the pattern of the computation should be clear the parallel version is presented. At the GPU, utilizing threads to hide memory latency is of importance, therefore the GPU implementation has one thread for each output value generated. Furthermore, data fetched from global memory is reused by several threads in the block. Shared memory is used as a scratchpad memory for the subband data fetched from global memory, since accessing shared memory is almost as fast as accessing registers, or as fast as accessing registers under the right circumstances.

4.2.1.1 Walk-through of the synthesis kernel

A walk-through of the main part of the subband synthesis kernel, seen in Listing 4.1, is now presented.

The code in Listing 4.1 is executed by all the threads that produce output. In contrast to the serial version where the output of different elements are handled in a for-loop, the GPU version has one thread for each output element. Thus, the for-loop iterating over the samples in a subband is replaced

Listing 4.1: Inner-loop of subband synthesis kernel

```
  unsigned int DS = DOWN_SAMPLE;            // 8
  unsigned int F_HALF = FILTER_SIZE_HALF;   // 16
  unsigned int s = threadIdx.x & 0x7;
#pragma unroll
for (int sband = 0; sband < DS; sband += 2) {
  unsigned int filter_offset = sband * F_HALF;
  unsigned int delta = (8 * (tid >> 3)) + sband + 8; // plus 8 (adjustment)

  sum += SB_data[delta + 24] * filter[s + filter_offset]
    + SB_data[delta + 16] * filter[s + DS + filter_offset]
    + SB_data[delta + 8 ] * filter[F_HALF - 1 - s + filter_offset]
    + SB_data[delta     ] * filter[F_HALF - 1 - s - DS + filter_offset];
      }

#pragma unroll
for (int sband = 1; sband < DS; sband += 2) {
  unsigned int filter_offset = sband * F_HALF;
  unsigned int delta = (8 * (tid >> 3)) + sband;

  sum += SB_data[delta + 24] * filter[s + filter_offset]
    + SB_data[delta + 16] * filter[s + DS + filter_offset]
    - SB_data[delta +  8] * filter[F_HALF - 1 - s + filter_offset]
    - SB_data[delta     ] * filter[F_HALF - 1 - s - DS + filter_offset];
      }
```

by a thread for each element. To address the different subband and filter offsets used, each thread has to calculate its own offset based on its thread ID. Section 4.2.1 gives a description of which order the filter coefficients are accessed in the inner-loop of the subband synthesis function. In the kernel the variable `s` is assigned the value of the thread bitwise-anded with the value 7, this way the variable `s` will have a value ranging from 0 to 7. Therefore, the value of `s` will "count" modulo 8 over the increasing thread IDs. The variable `filter_offset` chooses the correct offset into the filter used based on the subband (`sband`).

The `delta` variable gives the start offset for the given thread. It is calculated such that a grouped of 8 threads belong to a given offset. This is done by dividing the thread ID by 8 (actually right shifting the value by 3) then multiplying by 8, since integer arithmetic is used the three least significant bits are lost. This leads to a grouping of the first 8 threads into one group and the next 8 threads (threads 8 to 15) into another group, and so on. This constitute the base address for first index, from which the next three indices are calculated. Given the interleaving of the subbands (see Section 4.2.4) the next index in the same subband is found 8 indices ahead in the interleaved stream. This can be seen by the addition of 8, 16 and 24 to reach the next indices used to calculate the current output. To access the same index for another subband a simple addition of the subband number is done, seen by

the addition of `sband` for the variable `delta`. There is also an addition of 8 as an adjustment because of the padding that is done in the kernel (to avoid using extra global memory for the padding).

A thread generates one output element which is a summation of the contribution from all the subbands. Therefore, the kernel has to loop over all the subbands. This looping is divided into even and odd subbands, because there is a different sign when utilizing the symmetry in the filter for odd subbands than for even subbands. This is observed by the two for-loops in Listing 4.1. The first loop handles even subbands and the second odd subbands.

4.2.2 Mirroring

Mirroring is used to avoid expansion of the filtered data and to reduce edge artifacts, in this section the mirroring technique used in the implementation is presented.

In Section 2.2.4, three constraints are mentioned to prevent expansion of the signal. In the implementation, the first constraint is fulfilled even though the length of each subband is not equal. The length of even subbands is given by K_{even} and odd subbands by K_{odd}. These are $K_{even} = \lfloor (N+M-1)/M+1 \rfloor$ and $K_{odd} = \lfloor (N+M-1)/M-1 \rfloor$. Where N is the length of the input signal and M denotes the down-sampling value. The second constraint focuses on signal extension through either of the following: zero, circular or mirror extension. The implementation uses a mirror extension. The last constraint, critical down-sampling, is also applied in the implementation.

The extension of the different subbands is such that even subbands uses symmetrical extension while odd subbands are extended asymmetrically. Figure 4.4 gives an illustration of this. In the figure the vertical dotted line marks the reflection axis, and is found at both ends of the signal. The circle found just outside the original signal for the odd scheme indicates a value that is always zero. The reflection axis is placed differently for symmetric and asymmetric extension, for the symmetric case it is on a sample and for the asymmetric case it is between two samples. Actually, the reflection axis for the asymmetric case is placed on the samples inserted at the ends, which are always zero. From this we can conclude that symmetric filters extend with samples that start with an offset of one from the ends of the input signal, and asymmetric filters extend without any offset into the input signal, but inserts an implicit zero between the input signal and the extended signal before doing the extension.

Extension as it is in the implementation can also be seen in Algorithm 4.2.1 in procedure doPadding. As zero-indexing is used, the length of a subband minus one gives the last element.

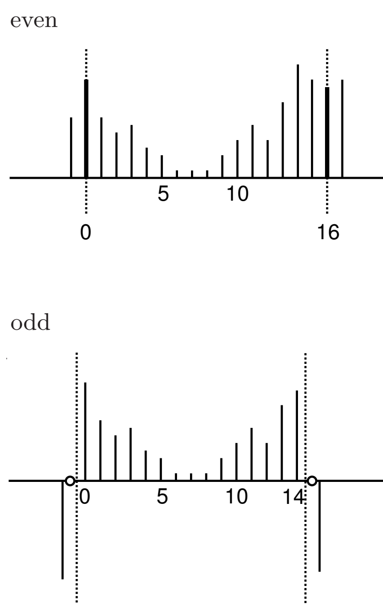


Figure 4.4: This figure illustrates the different mirroring schemes for even and odd subbands. At the top symmetric filtering is used, at the bottom asymmetric filtering is used.

4.2.3 Memory handling

Considerations to be taken while programming against the NVIDIA CUDA architecture is memory access patterns. The following subsection describes the access pattern used to utilize the memory bandwidth as much as possible.

4.2.4 Description of the interleaved format

Now, possible storage solutions for three-dimensional seismic data is explored. When designing high performance applications, details such as how data is stored is of importance. This is because of the way the memory hierarchy operates in current hardware solutions. If an application is memory bound, it is crucial to access the memory in a way that is as optimal as possible. Therefore, we now investigate some possible solutions that suits the hardware architecture of our implementation.

The program that is under investigation runs on a GPU, and since optimal performance is achieved when shared memory is used as scratchpad memory, it is a good idea to organize the data such that it is fetched in an optimal way. Details of how the memory architecture on GPUs work is described in Section 3.2.2.

After the analysis stage of the subband coding the data is partitioned into 8 subbands. To avoid passing eight different streams to the subband kernel or passing each subband consecutively in a single stream, interleaving is used. The subbands are interleaved into one stream by taking one index from each stream and gathering them into 8 consecutive places, this is done for all the indices. The result is that index 0 for all the 8 subbands are found at the 8 first indices of the interleaved stream. Further, index 1 of the subbands are found at the next 8 indices, and so on. As a demonstration, let us consider two indices from each of the 8 subbands, and denote each element as $s_n[i]$ where n is the subband and i is the index. Starting with two streams that looks like $s_n[0]s_n[1]$ for $n = 0, 1, .., 7$ we get: $s_0[0]s_1[0]s_2[0]s_3[0]s_4[0]s_5[0]s_6[0]s_7[0]s_0[1]s_1[1]s_2[1]s_3[1]s_4[1]s_5[1]s_6[1]s_7[1]$.

Organizing the stream this way keeps the same indices of the different subbands close, which is good when fetching the data, specially on GPUs see Section 3.2.4. Algorithm 4.2.2 shows the access pattern for the synthesis filter, as can be seen, only data three indices ahead is needed in the inner loop.

4.3 Huffman decoding implementation

This section will discuss challenges implementing Huffman decoding on GPUs. Only a simple Huffman decoder on the GPU has been implemented. It is able to decode Huffman encoded data from `libhuffman`. Given the sequential nature of Huffman coding the performance is limited to that of a single thread. A parallel approach has been investigated in [12], more details about this in Section 5.3.

The implementation of the Huffman decoder consists of two different logical parts, one builds the Huffman tree another decodes a Huffman encoded stream using the tree.

Table 4.1 gives the file format of data Huffman encoded with `libhuffman`. The number of codes, that is, leaf nodes in a Huffman tree, is given by the first four bytes in the encoded Huffman stream. The following four bytes gives the decoded size, which is the same at the original size. Then after these two values follows the actual definitions of the codes. The format of a code entry is: One byte gives the encoded symbol, followed by a byte giving the bit length. After these two bytes follows the necessary number of bytes to hold the bit string, the last byte might have additional unused bits. After all the code entries the actual encoded data is found.

The tree generation uses an array of a `struct` to represent nodes. This way pointers are not needed and a node can be referenced through an index

Offset	Length (in bytes)	Description
0	4	Number of codes (in big-endian)
4	4	Original size
8	variable	Code 1
⋮	⋮	⋮
varies	variable	Code N
varies	variable	Encoded data

Table 4.1: File format of Huffman encoded data from `libhuffman`.

into the array. This technique is described in the book *Introduction to Algorithms*, [13]. The reason this is done is because one early implementation using pointers resulted in incorrect use of shared memory, where global memory was accessed instead of the shared memory. This was found by analyzing the parallel thread execution (PTX) assembly code generated by the CUDA compiler. Where access to memory was done through the the instruction `st.global` for storing and `ld.global` for loading, that is, from global memory instead of shared memory, where the respective instructions should have been used: `st.shared` and `ld.shared`, see [14] for details. It might have been an erroneous implementation, but it is safer to use array indices when pointers are not needed. The NVIDIA CUDA programming guide also mentions the restrictions when using pointers with respect to addressing global and shared memory spaces:

> "Pointers in code that is executed on the device are supported as long as the compiler is able to resolve whether they point to either the shared memory space or the global memory space, otherwise they are restricted to only point to memory allocated or declared in the global memory space", [11].

The node structure consists of three members all of the same type, unsigned int, as observed in Table 4.2. The members *zero* and *one* is given the index of its child. As the tree is never traversed from a node up to the root, there is no need to keep track of a node's parent. Although the symbol can only have 256 different values, an unsigned int (32-bits) is used to represent the symbol, because a compaction of the data structure is not possible. The reason for this is because of a restriction in the instruction set architecture (ISA) of the PTX virtual machine used to generate device code for the GPU. The the document describing the PTX ISA mentions its alignment requirement, stating that all PTX instructions that access memory requires the address to be aligned to a multiple of the transfer size. It is

Type	Member Name	Description
unsigned int	zero	Index of the child at the edge marked 0
unsigned int	one	Index of the child at the edge marked 1
unsigned int	symbol	Value of the symbol

Table 4.2: Node structure

therefore little to gain by having the last member as an 8-bit type, even if it could result in less used storage, the cost would be lower performance. This is because data has to be transferred with lower transfer size, which result in lower bandwidth. Furthermore, to be able to transfer a 32-bit value it has to be aligned at an address multiple of 4. That is why the structure is organized as it is.

The nodes of the tree is allocated consecutively, starting with index 0 for the root of the tree. Then nodes are added to the tree referencing other nodes through indices. Index 0 is reserved as a special marker for child nodes, indicating that there is no child. This value was chosen since no node can have the root as a child. A node with value zero in both member *zero* and *one* is a leaf node.

Decoding a Huffman encoded stream on the GPU requires that the Huffman tree is built before the decoding can start. Then the encoded stream is parsed starting at its very beginning. The decoding is as described in Section 2.1.3. For each decoded symbol one starts at the root, the node at index 0, then follows the indices found in members *zero* and *one* depending on the current bit value found in the bit string. If a bit is zero then the index found in member *zero* is followed, and the same applies for bits with value one, except then the index found in member *one* is followed. When both the values found in member *zero* and member *one* are 0 then the node is a leaf node. At the point of reaching a leaf node the value in the member *symbol* is emitted (a byte) to the output stream.

4.4 Transpose

Transposing is done as means to improve performance and to simplify the implementation by only requiring the transform for one direction, either vertical or horizontal. Coalesced memory access is of great importance to gain high performance, thus data has to be organized in the correct manner for this to be exploited, which a transpose ensures. Two kinds of transposes is used, one for transposing a two-dimensional grid and one for a three-dimensional cube, a stack of two-dimensional grids. The 3-D transpose is actually a special case

of the 2-D transpose acting in three dimensions. They are described in the following subsections 4.4.1 and 4.4.2.

4.4.1 2-D transpose

The following subsection on 2-D transpose is a rewrite of an earlier work of mine [9].

The 2-D transpose kernel is now explained. First the logical partitioning is done, creating two-dimensional thread blocks of size $N \times N$. Coalesced memory access is possible if N is a multiple of 16. In the implementation shared memory is used as scratch pad memory, that is, as programmer-controlled cache. Each thread in a thread block handles an element each, which it reads into shared memory. For the transpose kernel each thread block is two-dimensional, so it has two-dimensional coordinates for its threads. Before writing a value to the shared memory the thread transposes its coordinates. Synchronization is needed before accessing data from the shared memory, either to write to or read back from global memory.

The kernel can be explained the following way: It reads data from global memory into shared memory, then synchronizes. After the synchronization, data from global memory is read into a temporary variable for each thread, from the part below the main diagonal of the logical partitioning. Then what is stored in shared memory is written to global memory, followed by another synchronization. After this synchronization the temporary variable is written to shared memory. Finally after the last synchronization, what is in shared memory is written to global memory.

Memory usage is kept at the same amount throughout the transpose, since it is an in situ algorithm. Thread blocks operate on the main diagonal or above, those below return at once. Thus the blocks not returning at once does the actual work, and they operate on both sides of the main diagonal. This is done because there are no natural mechanisms for synchronization between blocks. Figure 4.5 shows the idea behind the transpose function.

4.4.2 3-D transpose

To be able to do the subband coding efficiently a 3-D transpose has to be done. There are two good reasons for this: Implementation complexity and performance. The way the subband coding kernel is implemented it assumes a given layout of the data in memory. The layout of the data is organized such that elements that are needed to calculate a new value is close in memory. Not only does this ease the address calculation of the needed elements, but

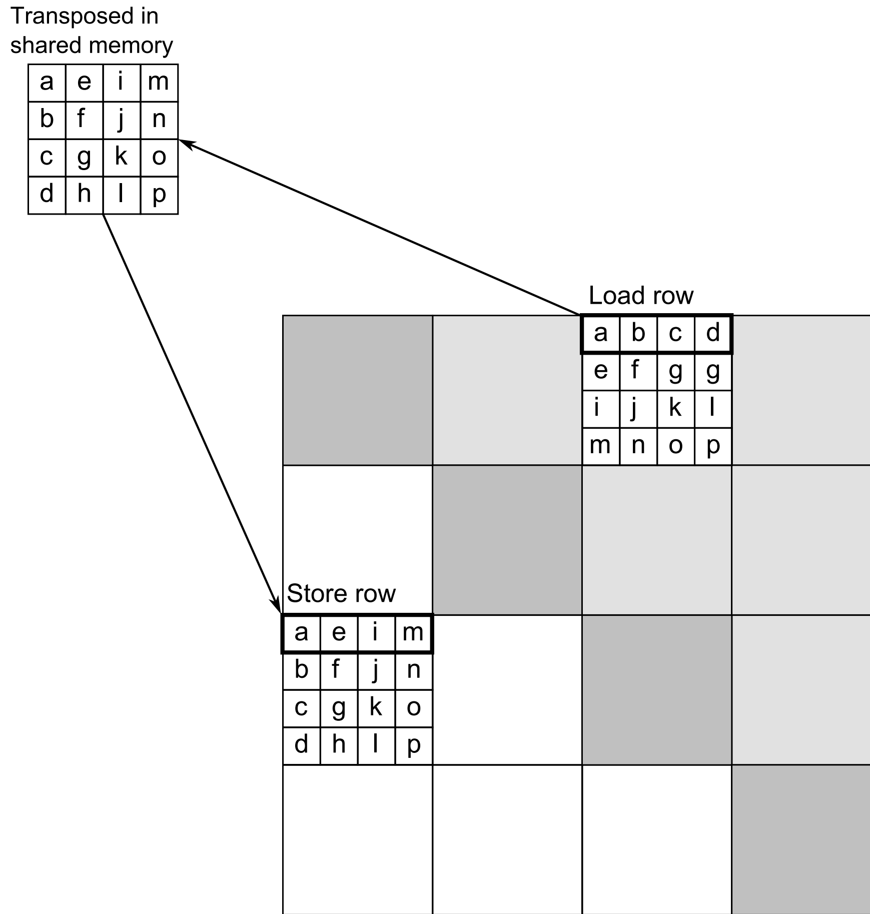


Figure 4.5: The 2-D transpose function's use of shared memory.

it also makes the data access more efficient since coalesced memory access is used.

The 3-D transpose used is actually an extension of a 2-D transpose, the difference being that it operates over a three-dimensional dataset. A stacked two-dimensional layout forms the three-dimensional dataset of seismic data. The layout of this three-dimensional dataset is not optimal with respect to performance when accessed in its "depth" direction, because of the memory access pattern that prevents efficient coalesced memory access. Since the subband coding kernel is designed to process data that is close in memory, a rotation or a transpose solves this layout problem. A in-situ transpose is chosen in favor of a 90 degrees rotation of the stack, mainly because of its ease of implementation. A 90 degrees rotation in-situ would require a more complex access pattern to avoid overwriting data, and would most likely not be that easy to parallelize.

After the transpose of the stacked data, what used to be on the Z-axis is now on the X-axis and vice versa. An analogy to this transpose is a book where each page represent a 2-D plane, turning the page is like traversing in the depth (Z-direction). Before the transpose, accessing the words in the Z-direction is like turning the page and finding the same row and column on the next page. After the transpose, the words are arranged on a line. Consider the first column on every page, after a transpose it is arranged as rows on page one, where the column on the first page is at row one, the column on the second page at row two, and so on. The same applies for the second column of every page where the columns are arranged into rows at the second page. Generally after a transpose, the ith column of every jth page is arranged into jth row of the ith page.

The 3-D transpose kernel is similar to the 2-D transpose kernel with only minor changes to support three indices. These changes are because the 3-D transpose kernel operates over a data set addressed through three indices. The shared memory is also expanded to three-dimensions, although two-dimensions is sufficient since the transpose is only working in a plane. This new dimension leads to a slightly different address calculation, as described below. Choosing the width (x-dimension) of the thread block to a multiple of 16 gives coalesced memory access. Given that there is a restriction of 512 threads per thread block and that a multiple of 16 is wanted for the x-dimension as well as the z-dimension, simply choosing the dimension of the thread block to be 16 in x-dimension 1 in y-dimension and 16 in z-dimension is natural, this results in 256 threads for the thread block. As each thread is to hold a float value requiring 4 bytes each, the amount of shared memory to hold these values is 4 bytes $\times$ 256 = 1024 bytes. Using the tool provided by NVIDIA for calculating occupancy on the GPU, a spreadsheet, suggest that this is optimal as it results in 100% occupancy. The occupancy gives an indication of how well the GPU resources is spent. Furthermore, the logical partitioning with a main diagonal is also present in the transpose kernel working on a 3-D data set, except that the y-dimension is now the z-dimension.

To illustrate the analogy with the book above and to show how the data is organized in memory, an illustration of the data in memory before and after the 3-D transpose is given in Figure 4.6. In the figure each rectangle is a slice, there are 4 slices in this cube of size 4^3. In the rectangles, the rows are the x-dimension and values along the rows are at consecutive addresses. For values along the columns, the y-dimension, consecutive addresses are calculated as: width $\times$ y-pos. On the GPU the actual width of a slice may differ from the logical width. This occurs when using an API call to allocate memory that do padding to meet the alignment requirements to allow coalescing when

going from one row to the next row. This ensures that coalescing is possible on each row. When allocating memory that can be padded, a variable giving the pitch (the actual width of a row in bytes) is returned. Addresses in the y-dimension can then be calculated through the following code (C code):

```
T* pElement = (T*)((char*)BaseAddress + Row * pitch) +
Column;
```

Where T is the type, and row and column is given in addition to the base address, as described by the NVIDIA CUDA reference manual [15]. The last direction the z-dimension, is calculated by: width $\times$ height $\times$ z-pos. All the address calculations mentioned should use the actual width when calculating the position of an element.

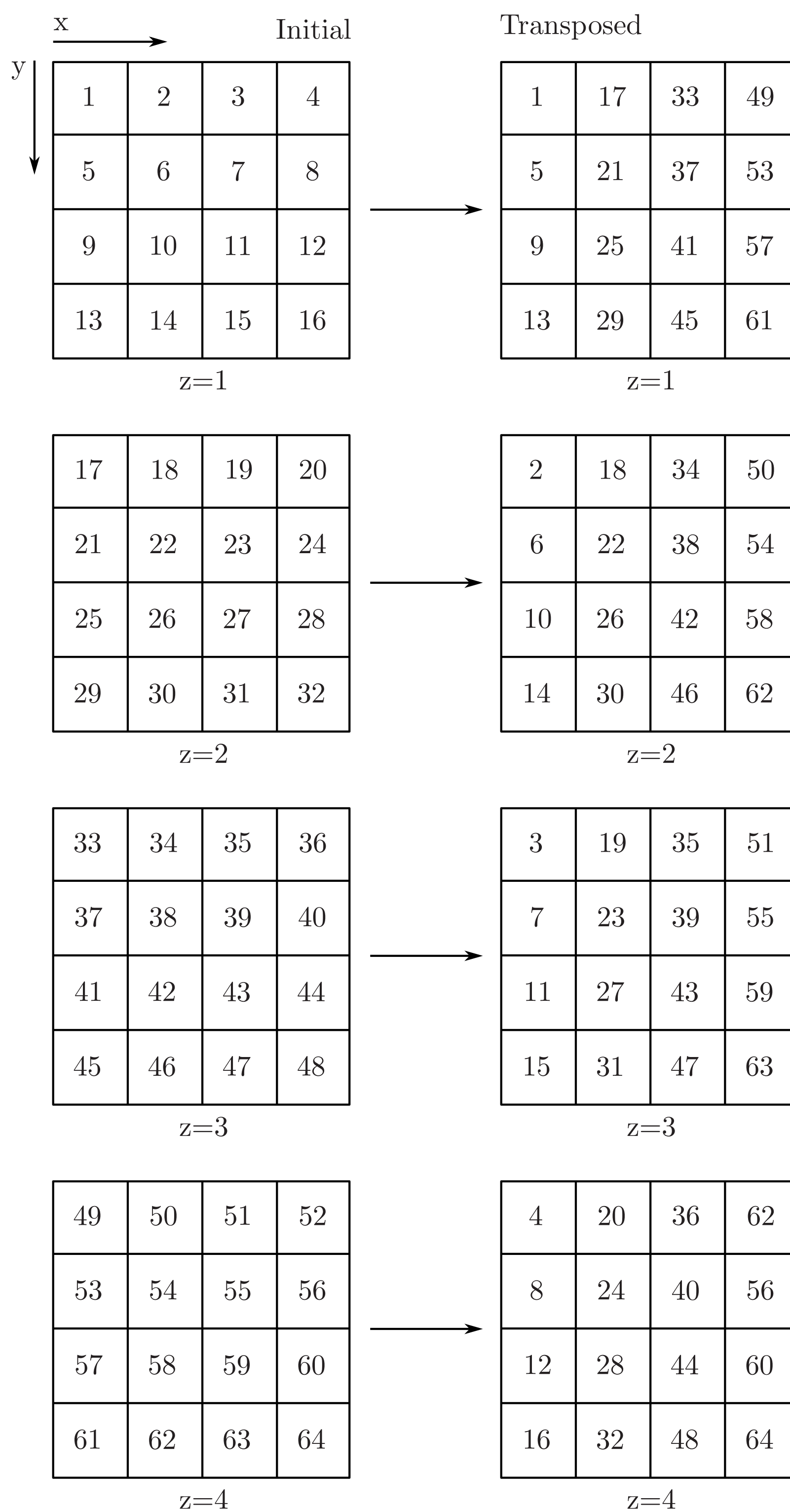


Figure 4.6: The layout of elements in memory before and after the 3-D transpose.

Chapter 5

Results

This chapter examines the results of different tests done on various selected components of our compression implementation. The following Section 5.1, gives and overview of the testing environment, followed by Section 5.1, presenting our benchmarking results, then Section 5.3 presents the compression efficiency, which is followed by Section 5.4 and Section 5.4 gives a description of our compression algorithms.

5.1 Testing environment

The timing benchmarks presented in this chapter was performed on a system with the following specifications: It ran on a system with Microsoft Windows XP Professional x64 Edition, version 2003 with service pack 2, as the operating system. The processor in the system was an Intel® Core™ 2 Quad processor Q9550 running at 2.83 GHz, with access to 8GiB[1]. The graphics card was a NVIDIA Quadro FX 5800 with 4GiB of memory, using driver version 182.50.

It should be noted that the program components have not been fully optimized as the primary concern was to get a working proof of concept giving correct results.

5.2 Benchmarking

To measure the performance of the implementation different setups was used. The goal was to see if transferring a compressed stream from the host mem-

[1] 1GiB = 2^{30} bytes = 1024MiB. 1MiB = 2^{20} bytes = 1024KiB. 1KiB = 2^{10} bytes = 1024 bytes.

ory to the GPU memory, and then decompress it on the GPU, would result in shorter transfer time. Because the decompression consists of different stages, which can be divided into three main stages as described in Section 2.2.1, only the most essential parts has been tested. That is, the parts that cannot be removed if compression is desired with the method suggested, subband coding. Therefore, the transformation (synthesis) and also the inverse quantization has been timed.

The timing was done by calling the functions a number of times, this is to reduce the inaccuracy of the measured time, because the resolution of the timer is unknown and might be to coarse if measuring is done over a small time interval. Before doing the measured iterations the kernels was called once, as a “warm up” as it is called in some of the examples of the CUDA SDK, such as the white paper by Podlozhnyuk [16]. This warm up was done because the first invocation of a kernel might give slower runtime than the following invocations. The result it then averaged over the number of iterations taken using the arithmetic mean given by: $\frac{1}{n}\sum_i^n a_i$, where a_i is the time for the i iteration and n is the number of iterations taken.

Four timing tests was done, with results given in Table 5.1:

1. A simple memory copy
   a) Allocated with Malloc3D on device
   b) Allocated with Malloc on device
2. Subband synthesis transform
   a) A memory copy followed by the subband synthesis transform
   b) Only subband synthesis transform
3. Inverse quantization
   a) A memory copy from host followed by inverse quantization and a copy to 3-D allocated memory on device.
   b) Inverse quantization and copying of result to 3-D allocated memory on device.
4. CPU version of the subband synthesis transformation

All the memory allocated on the host device that was copied to the device was allocated with `cudaMallocHost`, this gives what the CUDA reference manual call page-locked memory. This makes the driver track the

Table 5.1: Various timing results

Method	Description	Time (ms)
1a	Copy 8192 KiB to device (Malloc3D)	1.77
1b	Copy 8129 KiB to device (Malloc)	1.60
2a	Subband transformation (w/mem. copy)	813.76
2b	Subband transformation (no mem. copy)	811.97
3a	Inverse quantization (w/mem. copy)	1.82
3b	Inverse quantization (only copying on device)	1.42
4	Synthesis (CPU)	2029.00

virtual memory ranges of the allocated memory resulting in accelerated calls to `cudaMemcpy*()` functions[2].

In addition, timing using the NVIDIA CUDA Visual Profiler was done, with results summarized in Table 5.2. The resolution of the timings given by the Visual Profiler is higher than that of the other timing method used, giving values in microseconds (μsec). The results given in Table 5.2 was gathered by running a loop invoking these kernels ten times. The order of the kernels used is given below:

1. Transfer quantized data to GPU
2. Inverse quantize data on GPU
3. Copy result to data allocated with cudaMalloc3D
4. Pack the data (interleave indices from the subbands)
5. Apply the synthesis filter (horizontal coefficients)
6. Transpose data (X and Y axis)
7. Pack the data
8. Apply the synthesis filter (vertical coefficients)
9. Transpose data (X and Z axis)
10. Pack the data
11. Apply the synthesis filter (vertical coefficients)

[2] The star (*) represents all the possible endings to the function name.

Method	#Calls	GPU μsec	CPU μsec	% GPU time
Filter kernel	30	8.0615×10^6	8.0646×10^6	98.08
3-D transpose	20	1.4364×10^4	2.3636×10^4	0.17
2-D transpose	20	1.1600×10^4	1.7050×10^4	0.14
Shuffling	30	7.4556×10^3	1.2527×10^3	0.09
Inverse quantization	10	1.7301×10^3	1.8335×10^3	0.02
Memory copy	20	3.9404×10^4	3.2191×10^4	0.45

Table 5.2: NVIDIA CUDA Visual Profiler timing results.

12. Transpose data (X and Z axis)

13. Transpose data (X and Y axis)

These are the steps necessary to do the synthesis part of the subband coding over a 3-D data set. As can be seen from the times in Table 5.2 the subband coding is the most prominent kernel using most of the time of the synthesis process. All the tests was conducted on a cube with lengths of 128 elements for each dimension, each element of type float (4 bytes) which result in a memory footprint of 8 MiB for the cube.

The CUDA Visual Profiler documentations gives guidelines on how to interpret the results it produces. In addition to give accurate timing values it gives information of coalesced memory access, divergent warps and similar features. As the counters can only target one of the multiprocessors of the GPU, it does not give data on all the warps launched for a particular kernel. The documentation states that the values returned by the tool is not expected to match what can be found by inspected the kernel code. Thus, it is a tool best suited to compare optimized and unoptimized code.

Looking at the times given in Table 5.1 it is easy to conclude that the decompression of compressed data using subband coding with the current implementation on the GPU, is not more efficient than simply transferring the raw data. In this particular, implementation the synthesis filtering takes most of the time. More details around the results is given in Section 5.4.

5.3 Compression efficiency

The signal-to-noise ratio (SNR) of the seismic data and the seismic data after quantization and inverse quantization is measured over a cube of dimension 64. This gave a SNR of 24.67 decibel (dB), a value above 27 dB is desired to ensure that not too much visible noise appear. The SNR is defined by the following equation for dB:

$$\mathrm{SNR} = 20\log_{10}\left(\frac{A_{\mathrm{signal}}}{A_{\mathrm{noise}}}\right). \tag{5.1}$$

Where the amplitude A is in root mean square (RMS), RMS over a collection of values $\{x_1, x_2, \ldots, x_n\}$ is defined as:

$$x_{\mathrm{rms}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}. \tag{5.2}$$

In addition to measure the SNR, a measure of the standard deviation for each of the resulting subbands of the $64 \times 64 \times 64$ cube was calculated, this can be see in Figure 5.1. This figure shows how the energy of the input signal is gathered into few subbands, with the highest value found in one of the first subbands. The SNR given above can be improved by a more elaborate quantizer which gives a better result. As the focus of this thesis is not that of a signal processing perspective, this is not investigated.

To understand how efficient the compression of the implementation is, a simple test was conducted. It illustrates the compression achieved with simple RLE and with both RLE and Huffman applied to the quantized stream. The Huffman coding was done after the run-length encoding in the case where both RLE and Huffman was used. For the run-length encoded stream the RLE encoder written (CPU) was used, and for the Huffman coding `libhuffman` was used.

Using the run-length encoder implemented gave a compression ratio of 0.0739 for the cube of dimension 64^3 and 0.0655 for a cube of size 128^3. By using Huffman encoding these ratios was reduced further as can be observed in the column *Compression Ratio* in Table 5.3 that gives the ratio for RLE + Huffman.

Cube	Raw	RLE	RLE + Huffman	Compression Ratio
64^3	1024KiB	75.692KiB	36.702KiB	0.0358
128^3	8192KiB	536.529KiB	252.854KiB	0.0310

Table 5.3: Compression results with different cube sizes

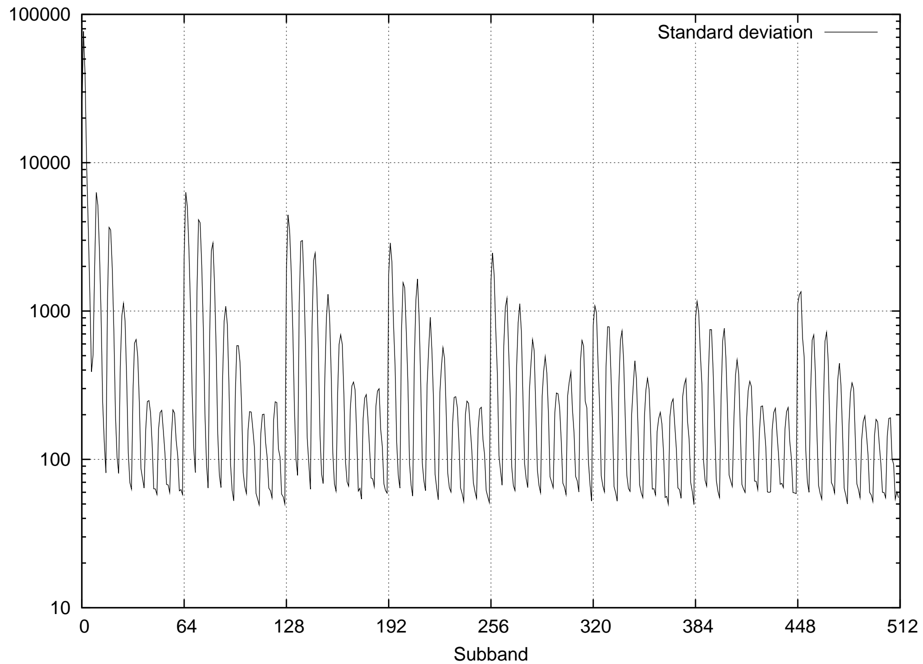


Figure 5.1: The standard deviation over 512 subbands of a 64^3 cube.

5.4 Discussing the results

The following subsections describes possible suboptimal design choices in our implementation

5.4.1 GPU memory accesses

The subband synthesis filter kernel is implemented using constant memory for filter coefficients, and shared memory for the filter data. This might not be the ideal solution due to the nature of constant memory and the way the kernel is designed to make use of it. Constant memory is cached according to the programming guide [11], and it states that reading from it is as fast as reading from a register, given that all the threads read from the same address. Otherwise it is serialized, assuming that it is still as fast as accessing a register for each access it is faster than global memory. The kernel does a lot of lookups on the filter coefficients, and each thread accesses it own location, which means that it probably is serialized and in worst case results in access to global memory. The shared memory is used to hold the filter data and is accessed as often as constant memory within the inner loops. Access pattern to the shared memory is not as strict as for constant memory and it is also

as fast as accessing a register given that there are no bank conflicts between the threads. If there are bank conflicts the access is serialized. There are 16 banks for devices with compute capability 1.x, and requests are split into one for each half warp (16 threads). Assuming that the shared memory allocated for each thread block is aligned in memory such that the first index starts at the first bank and the consecutive indices are at consecutive banks, the access pattern in the inner loop falls at 16 different banks for a half warp. Starting in the middle of the bank array, at offset 8 (offset 24 modulus 16) for the first lookup in the synthesis kernel, see Listing 4.1 (even subbands), this should avoid bank conflicts if the access does not have to be at an alignment modulus 16 for the first thread in the half warp. For the following iterations of the loop this constraint of accessing 16 different banks for a half warp is fulfilled. The same applies for the three other accesses to shared memory found in Listing 4.1.

5.4.2 Branching in GPU

Within the kernel there are many branches as well, this is to avoid having to store the padded values in global memory. Accessing global memory can take from 400 to 600 clock cycles of memory latency, this is why this choice was made. Branch divergence as described in the programming guide for CUDA states that divergence only occur within a warp, and that threads within a warp are serially executed for each branch path taken, disabling threads not on the path, this is done for each execution path and when complete the threads converge back to the same execution path [11]. Therefore, the conditional checking if a given thread block is the first or the last block processing the data should only execute its assigned code block if the conditional is true. Otherwise, the warps not fulfilling the conditional should not execute any of that code. There are some special cases handling the mirroring of the filter data that applies only to some threads, but they do not access global memory so they should not consume to much time. One possibility is also that all the branches are actually executed, which would lead to a great deal of unnecessary data processing and it can be the main reason for the poor performance.

5.4.3 Inverse quantization and alignment

Now, as another comparison basis of the GPU implementation, the inverse quantizer is interpreted. It is simple in its implementation, it only reads quantized data from one buffer, do some simple calculations, and write the result to another buffer. Because it does not write this data to memory

allocated with `cudaMalloc3D` it is not necessary aligned correct for the subsequent lines in a cube. Thus, after the processing the data written to a temporary buffer is copied into memory aware of padding and alignment requirements for data allocated for 3-D access. The time for the whole quantization step including memory copy from host till the complete result is placed in the target location is given in Table 5.1. The whole process takes 1.82ms, considering only the quantization including the final copy, it takes 1.42ms. Given that this is the fastest kernel in Table 5.2, the quantization timing-results presented in Table 5.1 might give the best times achievable out of these kernels. If this is the case, the time spent processing including copying from host memory is above that of a pure memory copy from host to device. The time doing the inverse quantization on the device including the copy between different buffers is marginally smaller than that of a pure copy from host to device. In addition, the quantization the way it is implemented only gives a compression ratio of 1:4. As this is only one part of the whole compression scheme under investigation, one cannot state that this is an improvement masking the bandwidth limitation between CPU and GPU using compression with subband coding. In addition the the inverse quantized data has to be processed by the subband synthesis filter before it can be used. This would add to the time achieved by the quantizer.

5.5 Proposing improvements to the implementation

Now that the possible bottlenecks have been discussed some suggestions for improvements are given. First, the use of constant memory has to be reconsidered. Second, the branch conditions removed or improved. Third, see if it is possible to use more registers and gain performance that way, and finally see if the use of more threads is a solution. The reason for choosing constant memory for the filter coefficients was because is has the lifetime of an application, so it is not lost during kernel launches. Furthermore, it is cached and has a peak performance equal to that of a register, with respect to access time. The problem is that is has a restricted access pattern to gain good performance. A solution is to copy the filter coefficients into shared memory for each block. This requires 16 (filter length) $\times$ 8 (subbands) $\times$ 4 (size of element) $=$ 512 bytes of shared memory in addition to that used for the filter data, and usually some uncontrollable use of shared memory inserted by the compiler, and by the parameters passed along to the kernel in shared memory. This expansion in use of shared memory will result in a

total shared memory footprint of 384 bytes (filter data) plus 512 bytes for filter coefficients in addition to the uncontrolled amount of shared memory, which gives a requirement of more than 896 bytes. Then some restructuring of the code reducing the number of branches taken can be tried, most likely that would require a step where the data is preprocessed extending the the signal ends in memory before calling the synthesis filter. If possible, one can try to use more threads to do the computation, this would probably require more shared memory for the filtered data and perhaps a change in how data is accessed by a block.

5.5.1 Planning tool

Along with the CUDA software development kit (SDK) a tool to calculate how much of the GPU resources is use is included, the CUDA GPU occupancy calculator, which is a spreadsheet containing data on the different compute architectures and how to compute the usage of different resources. It takes as input parameters information on the number of registers used per thread, the number of threads per block, as well as the amount of shared memory used per block. With this information and the compute capability of the target GPU it gives information on the occupancy of each multiprocessor, in number of threads, active warps and active thread blocks, and a percentage of resources used. It also presents information on the limiting resource or resources. An example with a device having resource capability 1.3 using 64 threads per block, 16 registers per thread and a usage of 1024 bytes of shared memory results in 512 active threads per block, 16 active warps 8 active thread blocks and a 50% occupancy of each multiprocessor. In addition to numerical values it gives graphs showing occupancy gained by changing different parameters.

Feeding this spreadsheet with the values of the current implementation of the subband synthesis kernel, which are: 64 threads per block, 16 registers per thread and if we round the shared memory up to 1024 bytes, it tells us that the limiting factor is the number of active warps per multiprocessor, there can only be 8 active thread blocks per multiprocessor and we use 2 warps per block using only 16 of the 32 warps available. Thus, there are 512 threads per multiprocessor (of 1024 on devices with compute capability 1.3). This suggest that more threads should be used, the problem is how to use them efficiently. With the current number of threads per block it is possible to use more shared memory without suffering according to the calculator. Comparing this to a device with compute capability 1.0 the story is a little different as the number of warps per multiprocessor is only 24 and not 32. In addition there are only 8192 registers and not 16384 as with compute capability 1.3. Thus the limiting factor is for devices of compute capability

1.0 both the available registers and warps utilized, so for these devices it uses 67% of the resources available.

From the information presented above, the expansion of shared memory to 1024 bytes and the copying of filter coefficients into shared memory should be a feasible expansion. If the number of registers stay the same, an expansion to 128 threads per block would result in 100% utilization for compute capability 1.3, the problem is do use the extra threads in a productive way, one solution might be to take all the 128 input element from the signal and process them in one kernel.

5.5.2 How fast is the subband synthesis filter?

The big question is: How fast does the subband synthesis filter have to be? Well, it is not a trivial answer as there are more to the decompression of the subband coded data than just the synthesis filter. In addition the decompression can be done in two different ways, one includes the transfer from the host memory to the GPU memory. The other neglects the transfer from host memory to GPU memory as it can be done once, and the compressed data can reside in GPU memory. An estimate on how fast the subband coding has to be if only memory copy and the synthesis is applied, is proposed in the following paragraphs.

Assuming that the user wants to access the data efficiently with coalesced memory access, the resulting data should be allocated with cudaMalloc3D to ensure correct alignment. Looking at the result of transferring data from page-locked memory on the host to the GPU suggest that this takes 1.77 ms for a 128^3 cube of floats (see Table 5.1). Therefore, if compression should be beneficial it should be able to transfer the compressed data and decompress it in a time that is lower that that of a pure transfer of the uncompressed data. Or, it should be able to decompress it faster than what a transfer of the raw data takes. In other words the whole process with quantization, decompression and filtering, should take no more than 1.77 ms.

A rough estimate of the number of floating point operations (FLOPs) required to do the subband synthesis based on Listing 4.1 is as follows: For even subbands $(4 \text{ mul } + 4 \text{ add }) \times 4 = 32$, four filter coefficients are multiplied by four signal elements, these are added together and stored in a variable. For odd subbands it is $(4 \text{ mul } + 2 \text{ add } + 2 \text{ sub }) \times 4 = 32$. The sum of these two are 64, therefore 64 FLOPs are done by each thread, this neglects other computations done by the thread such as calculating addresses. A cube of dimension 128 in each direction has 128^3 elements, and each element is processed by a thread. Therefore, the amount of calculations on a cube is at least $128^3 \times 64$ FLOPs ≈ 134 MFLOPs.

The number of floating point operations per second (FLOPS) of current NVIDIA GPUs is calculated as follows: number of SPs $\times$ 3 (FLOPs per cycle) $\times$ frequency. Thus, if we underestimate the GPU giving it only an 1GHz clock, and since the NVIDIA Quadro FX 5800 has 240 SPs, it would have a peak performance of $240 \times 3 \times (1 \times 10^9)$ which is 720 GFLOPs. This means that the GPU should be capable of calculating the synthesis filter in: 134 MFLOPs / 720 GFLOPS = 0.18 milliseconds. This totally ignores the time used to communicate with memory, nevertheless it illustrates that the problem with the current implementation probably is wrong use of different memory spaces.

5.5.3 Constant memory cache

One problem with the current implementation could be that the cache of the constant memory is not working as expected, it might be flushed such that memory access goes to global memory instead of using a cached value. If we estimate the time used accessing global memory to be approximately the same as for instance the 3-D transpose kernel plus a 50% overhead for computation and the extra global memory accessed. The synthesis filter read from global memory 192 values for a line of width 128 elements, then writes the 128 answers to global memory, so it has 320 accesses to global memory, where the 3-D transpose would have 256. The ratio 320/256 is 1.25 thus an estimate using 50% overhead should be reasonable. This would give us from the timing in Table 5.2 $(1.5 \times 1.4364 \times 10^4/20) \times 30 = 3.2317 \times 10^4$ μseconds which is about 250 times smaller than the current time of 8.0615×10^6 μseconds. This assumes that the values from the Visual Profiler over a multiprocessor is representable. Summarizing the result using the values of the other kernels in Table 5.2 as they are, excluding the memory copy, gives: $3.2317 \times 10^4 + 1.4364 \times 10^4 + 1.1600 \times 10^4 + 7.4556 \times 10^3 + 1.7301 \times 10^3 = 6.7467 \times 10^4$. This time is approximately 120 times smaller than that of the current implementation, even if the time for a new synthesis kernel was twice the suggested time, it would be a great improvement. If this new time scales for all multiprocessors, that is, the measurement on one multiprocessor is representable, the total time would be reduced by approximately 100 times. Although this is great, reducing the total time for the subband transform by 100 in Table 5.1 does not help, it would still be slower than the simple memory copy. Since this is just a theoretical estimate, it is difficult to say anything for sure on the performance actually achievable for the synthesis filter without investigating it in more detail. As this estimate does not consider the other stages in the decompression such as RLE and Huffman, the estimate does not tell the whole story. Both run-length encoding and Huffman decoding is

discussed below.

5.6 Our compression algorithms

Only coarse timing tests was done on both the Huffman and RLE implementations. As these were only implemented and tested for correctness they are not optimized and thus gave poor results, clearly favoring pure memory copying of data. These implementations do not utilize many threads and access memory in a suboptimal way so they are hardly representable, but they highlight the sequential nature of both these algorithms. Moreover a technical report from University of California at Berkeley [17], describes what they call dwarfs. A dwarf captures "... a pattern of computation and communication common to a class of important applications". One of these dwarfs are finite state machines, some of which can be "decomposed[ed] into multiple simultaneously active machines that act in parallel", [17]. The report summarizes results of their investigation of different applications, and it puts Huffman decoding under their 13th dwarf, finite state machines. This is the last dwarf and according to the report maybe the most challenging, and it might prove to be *embarrassingly sequential*. RLE is similar to Huffman it the way it is decoded and might fall under the state machine dwarf, different algorithms for text, picture and video compression are mentioned under this dwarf and some of these can use RLE as part of the compression.

Clearly, it is not a trivial task to gain high performance out of these compression algorithms. There is a paper by Klein and Wiseman [12] that investigates parallel decoding of Huffman encoded streams. Their experimental results shows a processing time about one-third that of a sequential Huffman decoder for a four processor system. This result suggests that it should be possible to achieve reasonable speeds on GPUs given their number of available processing units. Even if it did perform well on a CPU setup with four processors it does not necessarily scale to GPUs, so this should be investigated before any conclusions about the suitability of the algorithm to GPUs are made.

5.6.1 GPU versus CPU precision

A simple comparison between the CPU version of the subband synthesis and the GPU version with respect to precision was also performed. The comparison was done by calculating the distance between two values using

the following equation:

$$\max_i \left(d_i = \sum_1^n \sqrt{(x_i - y_i)^2} \right). \tag{5.3}$$

Where x and y denotes the input sequences, and d_i denote the distance between the two elements at the position i in the two different sequences. The max operator gives the greatest value in the distance set.

Applying Equation 5.3 on the result from the GPU and the result given by the CPU implementation gives the greatest distance between the two. This distance was measured to 0.007813. Even if the result differ by approximately 0.8% it is not the greatest source of error in the compression scheme, which is introduced by the quantizer. Thus this difference between the CPU and GPU implementation is acceptable.

Chapter 6

Conclusions

The gap between computational performance and memory access times, is only getting worse. This is particularly true when it comes to memory transfers between the CPU and GPU and the amazing computational power now available on modern GPUs. Since seismic data sets can become several terabytes, reducing the size of the data makes it more available given a limited amount of memory bandwidth. The goal of this thesis was hence to investigate the feasibility of compressing seismic data using subband coding as a means to reduce the effect of the limited bandwidth between the CPU and the GPU.

Our work included developing a proof-of-concept implementation of a system capable of decompressing seismic data on a GPU. An elaborate discussion of the validity of the implementation as a representable basis for a conclusion was then given. The main focus of our discussion was on the subband analysis kernel. Unfortunately our theoretical analysis gave it a half-open closure that needs further investigation. The compression methods we used, both Huffman and RLE, were briefly discussed as these implementations were implemented by porting from serial versions, but not fully optimized. In despite of the possibility of either the subband analysis transform, or one of the decompressors being able to give results with good performance, the combination of all these components seemed to result in a run-time exceeding that of a pure memory transfer without compression. Thus, the use of subband coding with the use of entropy coding to compress seismic data as a means to hide the limited memory bandwidth between CPU and GPU was not the success we had hoped for.

Another useful application of our work

Although not currently suited for the GPU-CPU compression, our GPU-based decompression implementations could improve transfer of seismic data over networks significantly. Seismic data sets can be enormous, so reducing these before a transfer over a network could reduce the retrieval time and the load on a network. Our compression results gave a compression factor between 27 and 32, and a SNR of 24.67dB for a cube of dimension 643. A speedup of 2.5 for the synthesis filter compared to the CPU implementation was achieved ($2029.00/813.76 \approx 2.5$). Given the compression ratio of roughly 0.04 presented in Section 5.3, our implementation would reduce the seismic data by a factor of 25. If we consider a network connection of 100 Mbit/s with 20% overhead, only 10MiB/s would be transferred. Using the proposed compression method reducing seismic data by 25 times would give a theoretical transfer rate of 250MiB/s.

6.1 Future work

Because the current implementation might be improved further, an effort could be made to see how much faster it is possible to get the synthesis kernel. The sequential nature of both Huffman and run-length encoding is probably the biggest challenge in parallelizing the decompression, and could be investigated further to see if they actually scale sufficiently to give acceptable decoding speeds on GPUs. Furthermore, other compression schemes better suited for the parallel architecture of GPUs could be investigated

Bibliography

[1] "Memory-link compression schemes: A value locality perspective," *IEEE Transactions on Computers*, vol. 57, no. 6, pp. 1–12, Jun. 2008.

[2] T. Røsten, "Seismic data compression using subband coding," Ph.D. dissertation, Norges teknisk-naturvitenskaplige universitet, Trondheim, Aug. 2000.

[3] ÖzdoğanYilmaz and S. M. Doherty, *Seismic Data Processing.* Tulsa, OK: Society of Exploration Geophysicists, 1987.

[4] "Speeding up transform algorithms for image compression using GPUs," *Stanford 50: State of the Art and Future Directions of Computational Mathematics and Numerical Computing*, March 29-31 2007, student poster.

[5] D. Salomon, *Data Compression, The Complete Reference.* London: Springer-Verlag London Limited, 2007.

[6] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the I.R.E.*, pp. 1098–1101, Sep. 1952.

[7] T. A. Ramstad, S. O. Aase, and J. H. Husøy, *Subband Compression of Images: Principles and Examples.* The Netherlands: Elsevier Science B.V., 1995.

[8] R. C. Gonzales and R. E. Woods, *Digital Image Processing.* New Jersey: Prentice-Hall, Inc., 2002.

[9] D. Haugen, "The lapped orthogonal transform using multiple GPUs," Computer Science Project Report, Norwegian University of Science and Technology, Trondheim, Norway, 2009.

[10] "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[11] *NVIDIA CUDA Compute Unified Device Architecture — Programming Guide*, NVIDIA, 2008, version 2.1.

[12] S. T. Klein and Y. Wiseman, "Parallel huffman decoding with applications to JPEG files," *The Computer Journal*, vol. 46, no. 5, pp. 487–497, 2003.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, Massachusetts and London, England: The MIT Press, 2003.

[14] *NVIDIA Compute — PTX: Parallel Thread Execution*, NVIDIA, 2008, ISA Version 1.3.

[15] *NVIDIA CUDA Compute Unified Device Architecture — Reference Manual*.

[16] *Image Convolution with CUDA*, NVIDIA, 2007, Victor Podlozhnyuk.

[17] K. Asanovic, R. Bodik, B. Catanzano, J. Gebis, P. Husbands, K. Keutzer, D.Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The landscape of parallel computing research: A view from berkeley," Electrical Engineering and Computer Sciences University of California at Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec. 2006.

Appendix A

Filter coefficients

The filter coefficients used by the subband coding can be seen in the four following tables Tables A.1, A.2, A.3 and A.4. Each filter has a length of 32, but only half the length of the filter is shown (16). The reason for this is that the other half i redundant and can be produced as seen in Algorihm 4.2.2. In addition to having a length of 32, the filters have 8 subbands marked by the columns h_x and g_x for analysis and synthesis filters respectively, where x is the subband number.

Table A.1: Analysis filter coefficients in the temporal direction, from Røsten [2].

	Analysis filter coefficients, $h_m(l)$							
	h_0	h_1	h_2	h_3	h_4	h_5	h_6	h_7
l								
0	0.006096	-0.015941	-0.023007	-0.008099	0.011024	-0.003653	-0.007104	-0.007900
1	-0.009134	0.000476	0.003330	0.005960	-0.021504	-0.004128	0.014532	0.016889
2	0.000601	0.017705	0.031131	0.007647	0.011982	0.013646	-0.008893	-0.010511
3	0.000165	-0.009817	-0.005237	-0.000981	0.010104	0.002960	-0.010908	-0.012643
4	-0.015298	-0.067306	-0.080253	-0.012672	-0.027305	-0.018986	0.029557	0.032331
5	-0.008286	-0.065844	-0.064885	-0.007899	0.030307	0.003832	-0.029121	-0.031880
6	-0.025789	-0.007358	0.015607	0.019643	0.000909	0.019207	0.005515	0.001630
7	-0.052737	0.001028	0.024532	0.033409	-0.065426	0.004166	0.035082	0.051029
8	-0.037663	-0.103766	-0.038451	0.118896	0.078424	-0.112463	-0.078470	-0.071138
9	0.033872	-0.139232	-0.178099	-0.044076	0.111467	0.231373	0.075029	0.031106
10	0.100825	-0.036716	-0.151063	-0.284211	-0.244996	-0.175210	0.020769	0.048119
11	0.152293	0.153597	0.124966	-0.059419	-0.039820	-0.096330	-0.189972	-0.138138
12	0.213064	0.332400	0.393746	0.364026	0.380032	0.354140	0.355229	0.228646
13	0.284517	0.421830	0.351936	0.166808	-0.124918	-0.306582	-0.426396	-0.312698
14	0.369063	0.360976	-0.018324	-0.397619	-0.416913	-0.059937	0.351563	0.379348
15	0.447614	0.145986	-0.387658	-0.322158	0.306708	0.419064	-0.136123	-0.418129

Table A.2: Synthesis filter coefficients in the temporal direction, from Røsten [2].

	Synthesis filter coefficients, $g_m(l)$							
	g_0	g_1	g_2	g_3	g_4	g_5	g_6	g_7
l								
0	-0.000160	0.003387	-0.004043	-0.003410	-0.000019	0.008453	-0.001746	0.000671
1	0.001048	-0.007547	0.008759	-0.012156	-0.014173	0.000314	0.016516	-0.022710
2	0.003908	-0.012832	0.013918	0.002350	0.009395	-0.017636	-0.014318	0.019351
3	0.006465	0.012661	-0.015430	0.018354	0.005434	0.005030	-0.008677	0.013456
4	-0.000750	0.040950	-0.045001	0.019606	-0.018034	0.032693	0.030401	-0.040645
5	-0.021309	0.029873	-0.024888	0.002297	0.024577	-0.000579	-0.030727	0.041260
6	-0.048609	0.001984	0.015180	-0.049628	0.007158	-0.020685	0.011331	-0.016950
7	-0.065727	0.024934	-0.002096	-0.093352	-0.018279	0.026760	0.033094	-0.055269
8	-0.043934	0.109218	-0.099950	-0.075215	0.101385	0.063607	-0.078469	0.099161
9	0.013576	0.147900	-0.188223	0.137132	0.042723	-0.221666	0.097492	-0.038866
10	0.078129	0.062944	-0.102608	0.277469	-0.244312	0.202597	-0.005522	-0.047053
11	0.136482	-0.124655	0.169837	-0.009494	0.029259	0.071738	-0.169810	0.135495
12	0.200938	-0.322868	0.396283	-0.354029	0.367703	-0.347160	0.346933	-0.224089
13	0.282134	-0.427550	0.321291	-0.109313	-0.168587	0.321836	-0.426406	0.308310
14	0.376453	-0.367232	-0.043690	0.400296	-0.404493	0.041150	0.360319	-0.373183
15	0.451957	-0.145624	-0.386818	0.298105	0.313810	-0.414986	-0.140746	0.419260

Table A.3: Analysis filter coefficients in the spatial direction, from Røsten [2].

Analysis filter coefficients, $h_m(l)$								
	h_0	h_1	h_2	h_3	h_4	h_5	h_6	h_7
l								
0	0.028521	0.016711	-0.015109	0.005772	0.009862	0.010154	0.000174	-0.001362
1	0.029985	0.008172	0.016105	0.016532	-0.000500	-0.006698	-0.000360	0.002941
2	0.014923	-0.003037	0.029721	0.013845	-0.011234	-0.018105	-0.001968	0.003341
3	-0.018072	-0.010483	0.003989	-0.008920	-0.012195	-0.010779	-0.002152	-0.002779
4	-0.051874	-0.018327	-0.024486	-0.019075	0.036217	0.040724	0.016813	0.014945
5	-0.063421	-0.025978	-0.021024	-0.026191	-0.017950	-0.004222	-0.030814	-0.037034
6	-0.047956	-0.025749	0.036362	-0.018587	-0.025978	-0.028487	0.033931	0.035760
7	-0.028350	-0.035764	0.071440	0.030709	0.011984	-0.008299	-0.013524	0.007132
8	-0.065276	-0.121619	-0.039981	0.147132	0.030011	-0.010965	-0.037661	-0.055654
9	-0.029414	-0.158878	-0.201710	-0.079363	0.117220	0.138888	0.060909	0.050566
10	0.041005	-0.086445	-0.156086	-0.282150	-0.224415	-0.138755	0.013513	-0.000556
11	0.139722	0.094500	0.111377	0.011158	-0.085653	-0.090311	-0.175982	-0.098496
12	0.243833	0.308314	0.361981	0.371695	0.385712	0.368322	0.350400	0.213524
13	0.326550	0.433381	0.315170	0.137313	-0.097106	-0.344411	-0.433517	-0.319338
14	0.380350	0.385782	-0.053417	-0.383178	-0.420351	-0.045989	0.361702	0.392374
15	0.409575	0.161938	-0.433829	-0.302472	0.303973	0.442656	-0.141355	-0.425021

Table A.4: Synthesis filter coefficients in the spatial direction, from Røsten [2].

	Synthesis filter coefficients, $g_m(l)$							
	g_0	g_1	g_2	g_3	g_4	g_5	g_6	g_7
l								
0	0.007463	-0.003982	0.003156	-0.017577	0.009358	-0.003249	-0.000355	-0.003806
1	0.005317	0.008598	0.018240	-0.024589	-0.002268	0.015302	-0.001474	-0.007345
2	-0.005030	0.021921	0.021601	-0.013073	-0.013063	0.024722	-0.002316	-0.005984
3	-0.019517	0.027365	0.002146	0.019659	-0.010059	0.005402	-0.000390	0.003258
4	-0.029959	0.031790	-0.017216	0.036616	0.032097	-0.050018	0.016001	-0.009592
5	-0.031347	0.040529	-0.010212	0.035013	-0.018171	0.004480	-0.029824	0.036694
6	-0.025098	0.049690	0.032128	0.007768	-0.027267	0.040574	0.034469	-0.042506
7	-0.013592	0.065095	0.042921	-0.052878	0.016627	0.010200	-0.019031	-0.004106
8	-0.004361	0.106162	-0.060483	-0.105525	0.054347	-0.020255	-0.038693	0.070838
9	0.035701	0.102718	-0.174097	0.121678	0.088697	-0.144998	0.070920	-0.051181
10	0.098056	0.011205	-0.100492	0.278828	-0.237640	0.150349	0.002003	0.009907
11	0.176120	-0.157290	0.157748	-0.032956	-0.063036	0.066600	-0.166805	0.095769
12	0.254916	-0.335560	0.371595	-0.369249	0.381299	-0.362242	0.347189	-0.210671
13	0.319950	-0.424899	0.292052	-0.115333	-0.113183	0.348068	-0.434200	0.321266
14	0.365755	-0.359145	-0.081206	0.389928	-0.415625	0.035775	0.365051	-0.391057
15	0.392123	-0.143210	-0.443834	0.294352	0.308292	-0.438996	-0.143558	0.423316

Appendix B

NOTUR2009 poster

The following page displays the poster presented at NOTUR2009, Trondheim.

Strategies for Handling Large Amounts of Data from Storage to GPUs

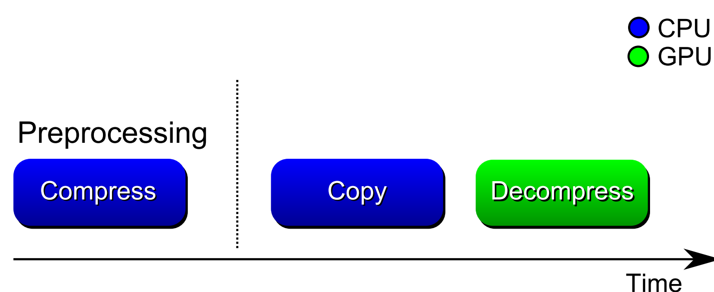
Daniel Haugen, IDI-NTNU
Supervisors:
Anne Cathrine Elster, IDI-NTNU
Tore Fevang, Schlumberger Ltd

Applications doing general-purpose computation on GPUs often suffer from bandwidth limitations due to the limited bandwidth of the communication channel, which typically is PCI Express.

Compression solves the bandwidth problem by reducing the amount of transfered data, without losing information or introducing noticeable noise. The amount of noise introduced is dependent on the compression algorithm used to do the compression, and can be totally avoided if lossless compression is used.

Subband coding is used to decorrelate the image data before doing quantization and entropy coding. This is done at the time of compression, and the inverse is done while decompressing. The coding involves convolution which is a compute intensive procedure that i well suited for the GPU architecture given its arithmetic complexity.

Overview of the decompression process



The compression under investigation utilize characteristics found in seismic data, that is, high correlation in horizontal direction, and not so high correlation in vertical direction.

To gain maximal compression the data is considered in three dimensions when compressing, not only two dimensions.

Stages in decompression on GPU

Examples of compression methods that could be used for entropy coding are Huffman coding and arithmetic coding.

Dequantization is the process of converting the quantized numbers back to, or close to, their original value.

The synthesis filter bank consists of a collection of filters that are applied to the dequantized data to recreate an approximation of the original input data.

Acknowledgements
We would like to thank NVIDIA for providing several of the graphics cards used in this project through Dr. Elster's membership in their Professor Afiliates Program. This project is done in collaboration with Schlumberger Limited, Trondheim, which also provides hardware and other resources.