Department of Informatics

Wavelet transforms and efficient implementation on the GPU

Master thesis

Hanne Moen

May 2, 2007

UNIVERSITAS OSLOENSIS

MDCCCXI

Contents

List of Figures

Abstract

Wavelets and wavelet transforms can be applied to various problems concerning signals. The ability to transform the signal into something representing frequencies and to see when the frequencies occurred, can be used in numerous fields. The calculation can be computationally expensive when applied to large datasets. By taking advantage of the computational power of a GPU when implementing a wavelet transform, the time of the computation can be substantially reduced. The goal is to make the application fast enough to solve a problem interactively. This thesis introduces the wavelet transform and addresses differences between some GPU toolkits, looking at development and code efficiency.

Preface

This thesis was written over a period of 18 months, reflecting one years worth of work. The work of the thesis has mainly been theoretical, learning about wavelets and wavelet transforms, and GPU background, but also about programming on the GPU with various toolkits. I would like to thank my supervisors Knut-Andreas Lie and Trond Runar Hagen who pretty much let me do whatever I found most interesting, answering all my questions and helped me stay focused on the task. I will also thank everyone at Hue AS, my friends and family who have helped me with optimistic suggestions whenever it was necessary. A special thanks to my partner Morgan, who always supported me and helped me throughout the project.

Chapter 1

Introduction

My thesis focuses on wavelets and wavelet transforms, and how to implement a wavelet transform with different GPU toolkits.

The idea behind using the GPU for general purpose computing is that the GPU is built in order to efficiently do parallel programming, meaning that it can perform the same computation on multiple data at the same time. When transferring this idea to a large computational problem you can get the result in a fraction of the time the same problem is computed on a CPU, where every problem has to be computed sequentially.

Even though I am not using seismic data when testing my implementations, that is the intensional use for the stationary wavelet transform, and is why I am explaining gathering seismic data in Section 1.2.

1.1 Research questions

Some questions I want to answer in this thesis are:

- What are wavelets and wavelet transforms?
- How can wavelets and wavelet transforms be applied to signals?
- How to use the GPU as a computational processor?
- What are the difference between some GPU toolkits (when implementing a specific wavelet transform)?

1.2 Gathering seismic data

Seismic data is often first represented as a gather of "‘shots"’, see Figure 1.2. Time is located on the y-axis starting on the top, while the offset of the shot goes from left to right. It is called shots because they can be produced by an air gun placed for example on a boat as in Figure 1.1, which shoots so that a sound signal is sent to the ocean floor, and the reflection is gathered by numerous microphones situated behind the boat. With the cannon on the boat shooting every 30 seconds, and the boat moving very slowly in a grid to cover all of the ocean floor, the datasets grow rapidly. Each shot is recorded by multiple microphones, and all the data is kept for future analysis. The gathered reflected signals is calculated with averaging algorithms to remove noise before the image in Figure 1.3 appears. It is not obvious to everyone what that is supposed to represent, and at this point the wavelet transform can be applied in order to easier see what the seismic data represents.

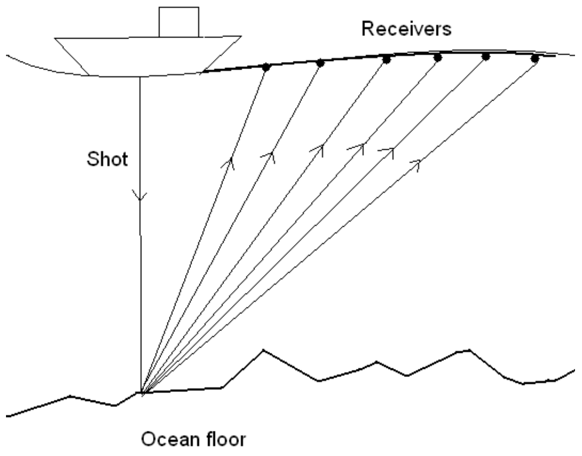


Figure 1.1: Boat gathering seismic data

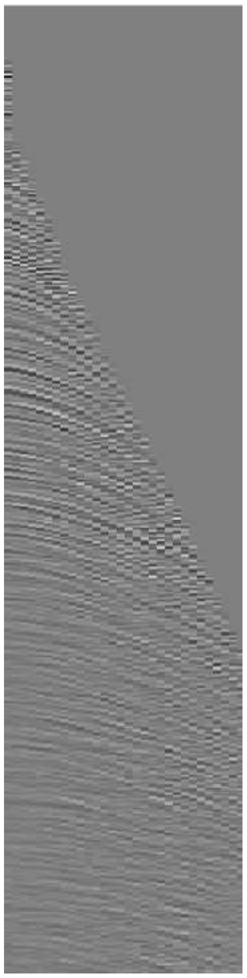


Figure 1.2: Seismic data gather

1.3 Thesis outline

The thesis has two main parts. The first part contains a literature study of wavelets and wavelet transforms, and the second parts contains a discussion of some GPU toolkits and implementation of the stationary wavelet transform. Chapter 2 introduces the wavelet and wavelet transforms. More about wavelets is found in Chapter 3, and Chapter 4 presents more details about some wavelet transforms. Chapter 5 contains some basics about the GPU and GPU toolkits, while I have written about my implementations in Chapter 6. A summary of the thesis together with a conclusion and suggestions for further work is presented in Chapter 7.

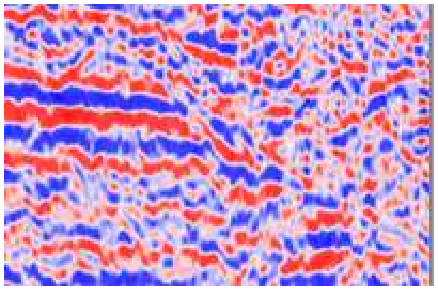


Figure 1.3: Seismic data example

Chapter 2

Introduction to Wavelets and Wavelet Transforms

> Wavelets are used to transform the signal under investigation into another representation which presents the signal information in a more useful form. [Add02, page 2]

When working with signals, the signal itself can be difficult to interpret. Therefore the signal must be decomposed or transformed in order to see what the signal actually represents. A common method here is to use the Fourier transform described in Section 2.3. The problem with the Fourier transform is that it can not give a precise estimate of when a frequency happens. Either you get the information about the frequencies of the signal or the time, not both simultaneously. When you want to know both what frequencies the signal consist of, and when the frequencies occurred, you should rather use the wavelet transform instead of the Fourier transform.

The continuous wavelet transform is the most general wavelet transform. The problem is that a continuous wavelet transform operates with a continuous signal, but since a computer is digital, it can only do computations on discrete signals. The discrete wavelet transform has been developed to accomplish a wavelet transform on a computer.

This thesis is about the difference between toolkits, but also about how they are used on a specific problem; in my case the stationary wavelet transform. There exist numerous different wavelet transforms, and why I chose to work with the SWT is a question that needs to be answered. I have briefly written about some of the wavelet transforms in Section 2.5, and more detail are given in Chapter 4; this in order to give a picture of what exists, but also to get an idea of the differences.

2.1 Wavelets and Wavelet Transforms

Wavelets and wavelet transforms are used to analyze signals. The transformed signal is a decomposed version of the original signal, and can be converted back to the original signal. No information is lost in the process.

When studying a musical tone, one of the features that is interesting is the frequency. The frequency for a clean A is 440Hz, see top plot in Figure 2.1. To determine the frequency of the signal one must measure the period of each wave, and calculate the frequency. The period of one wave is the time it takes from it is at one point in the wave, until it reaches the same position again. For example the time between two wave tops.

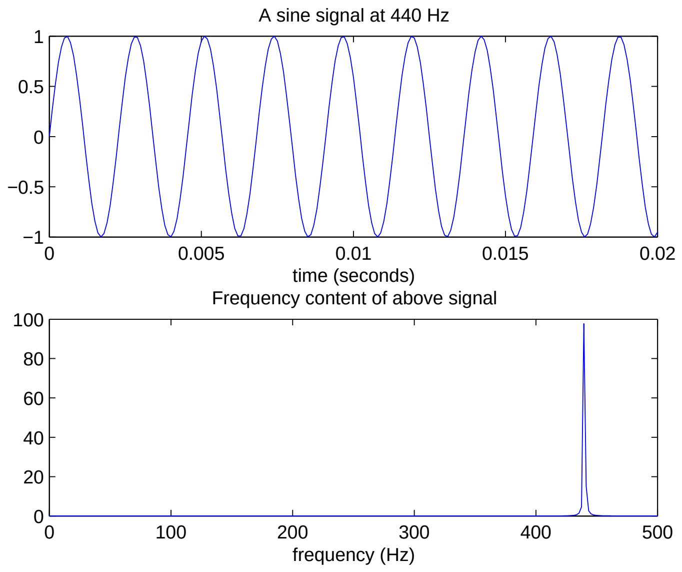


Figure 2.1: A sine wave at 440 Hz, and its Fourier transform.

Using different transforms, the signal can be transformed into other representations. For this example, instead of having amplitude as a function of time, it would be better to have the amplitude as a function of frequency. This can be done by using the Fourier transform. Once one knows what frequencies are present, one can easily determine which tones the signal consists of, in the case of a musical signal.

The bottom part of Figure 2.1 shows that it is easy to determine that the signal in the upper part of Figure 2.1 actually is an A when you perform the Fourier transform. Wavelet transforms can do the same, but they can also tell you when the tone A appeared in time, effectively giving you amplitude, time and frequency, all in one. More about this later.

2.2 Applications

Wavelets and wavelet transforms have many fields of application. In the case of music, the frequencies tell us what tones are represented. In the case of seismic data, the frequencies can tell us what the ground is made up of, what types of rock there are, and whether the rock contains oil or not.

It is appropriate to use wavelets and wavelet transform in all cases where you are looking for a given frequency/waveform and you also want to know what time it appears. Wavelet transforms are widely used in for example submarine sonars, to determine distances, speed, position and other information on other waterborne vessels and animals. Wavelets are also very good at removing noise from signals, detecting discontinuities breakdown points and self-similarity, and wavelet play an important part in compressing images. As an example [Gra95], the FBI in USA uses wavelet transforms to compress fingerprint images to 1/26 of the original size, thereby reducing the need for storage space from 200 Terabyte to just under 10 Terabyte. Wavelets are also used in fields like, but not limited to, astronomy, acoustics, nuclear engineering, sub-band coding, signal and image processing, neurophysiology, music, magnetic resonance imaging, speech discrimination, optics, fractals, turbulence, earthquake-prediction, radar, human vision, and pure mathematics applications.

2.3 The Fourier Transform

The wavelet transform is very similar to the Fourier transform, and knowing the Fourier transform can therefore be helpful when learning the wavelet transform.

The Fourier transform is a way of transforming a signal from time domain to frequency domain. If one can determine what frequencies a signal is composed of, and one knows the context of the signal, one can read much out of it.

Both the wavelet transform and the Fourier transform decomposes the signal into a sum of basis functions, but the basis functions are more compact with wavelets.

When computing a Fourier transform, coefficients are used to transform time-domain into frequency domain. The equation for a Fourier transform is written:

$$\hat{X}(f) = \int_{-\infty}^{\infty} x(t)e^{-i(2\pi f)t}\,dt. \tag{2.1}$$

The resemblance to the wavelet transform can be seen by comparing (2.1) with (2.4), where the frequencies also are placed in time providing the ability to know when the frequency occurred.

The Fourier transform (2.1) can also be described as the inner product of a signal $x(t)$ with a basis function $e^{-i\omega t}$:

$$\hat{X}(f) =< x(t), e^{-i(2\pi f)t} >= \int_{-\infty}^{\infty} x(t)e^{-i(2\pi f)t}dt. \tag{2.2}$$

The Fourier transform is used on many things outside the scope of this thesis. I will therefore only present an example to help establish the connection with wavelet transforms.

Let us say you are wondering if there are any whales in the sea close to where you live. You place an underwater microphone in the water and start recording. The received signal can be like the one in Figure 2.2. There is not very much you can tell about the signal by just looking at it. Nearby boats, waves hitting the shore, rocks rolling on the ocean floor, maybe even rain, will affect the signal, and you really have no clue what the signal actually represents. Then you perform a Fourier transform to see which components the signal is made of, as in the bottom part of Figure 2.2. Now we can see that there are two strong signals in all of this noise. One is 50 Hz, the other is 120 Hz. Knowing that toothless whales use low frequency sounds for communication, and toothed whales use high frequency sounds for echolocation and communication, this could very well indicate the presence of whales in the area.

The Fourier transform has the drawback that it does not place the frequencies in time. Therefore we do not know when the sounds in question happened. If they are continuous sounds, it would more probably come from a constantly rotating propeller from a ship or a nearby boat. With what we currently know, there is no way of telling. However, we are much closer to our goal than we where with the original signal. The next section describes a transform which tries to place frequencies in time, by using preset window sizes.

The Fourier transform maps a signal from time domain to frequency domain, but only knowing what frequencies a signal consist of is not enough when working with seismic data. You also need to know at what time the different frequencies occurred. That is why the wavelet transform is a more appropriate tool to use when working with data that needs to be located both in time and frequency. The Fourier transform needs a lot of

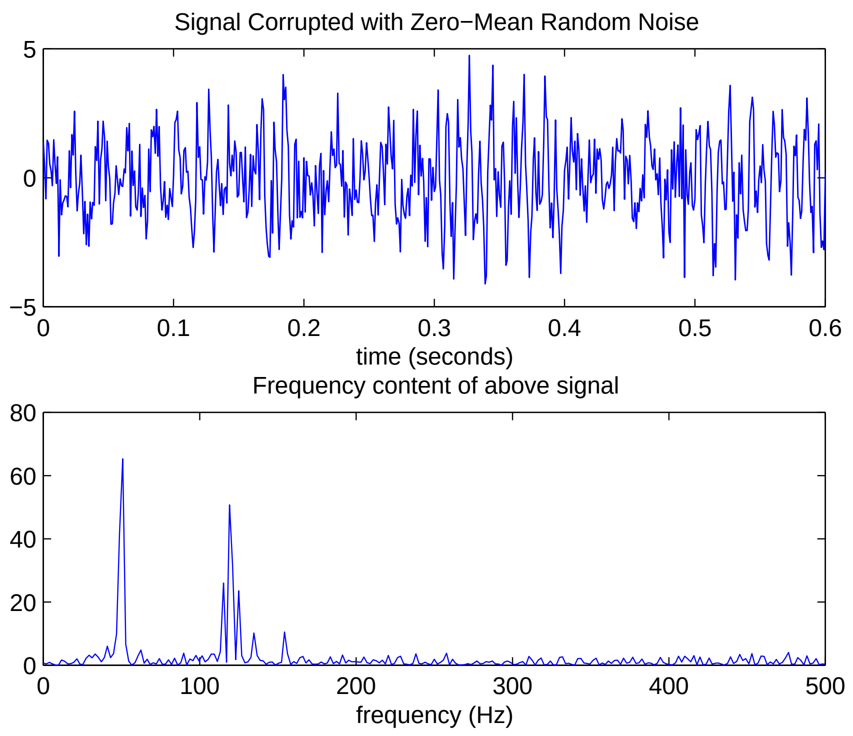


Figure 2.2: A noise input signal, and corresponding Fourier transform.

components in order to form a sharp corner as it uses sinusoids. When working with wavelets it can be seen that many have sharp corners themselves, and therefore do not need as many components to represent the same corner. Very briefly described, a wavelet is a wave that only oscillates for a finite period of time and is close to zero outside this period. Some examples of wavelets and more details can be found in Chapter 3.

2.4 The Short-Time Fourier Transform (STFT)

The short-time Fourier transform uses preset window sizes to better place frequency in time. Including time dependence can be done by taking short segments of the signal and then do the Fourier transform to get local frequency information. This method is called STFT, and the result is also called a spectrogram[1]:

$$STFT(\omega,\tau) = < x(t), \phi(t-\tau)e^{-i\omega t} > = \int x(t)\bar{\phi}(t-\tau)e^{-i\omega t}dt, \tag{2.3}$$

at time τ and frequency ω. Here $x(t)$ is the time-domain seismograph, $\phi(t-\tau)$ is the window function centered at time $t=\tau$, and $\bar{\phi}$ is the complex conjugate of ϕ. The Fourier kernel is written $e^{-i\omega t}$. If the window is a band-pass filter, small variations in frequency will be detected, while small changes in time will be washed out because of averaging over a long time duration. A window function over short time will not find rapid variations in frequency, but can detect short-lived changes in time.

The problem is that the window has the same size throughout the entire computation. And you have to choose whether you want a good time resolution or a good frequency resolution. Figure 2.3 illustrates the output after computing the STFT of the input signal from Figure 2.2. The color spectrum goes from blue to red, where red indicates a high output value and blue a low output value. A high output value means that the frequency is present. It can be observed that the signal is continuous at 50 Hz, and periodically a frequency at 120 Hz is also present. The time resolution is in this case quite good, but as you can see, the frequency resolution could be better. So still, this solution is not good enough when accuracy in both frequency and time is demanded.

2.5 The Wavelet Transform

When doing a wavelet transform, the signal is convolved with a wavelet.

$$x(t) = \frac{1}{C_g}\int_{-\infty}^{\infty}\int_{0}^{\infty} T(\sigma,\tau)\psi\left(\frac{t-\tau}{\sigma}\right)\frac{d\sigma d\tau}{\sigma^2}. \tag{2.4}$$

With convolution, the wavelet is shifted across the signal, and multiplied at each step. A large output at a step shows that the wavelet fits well, while a low output indicates that the wavelet is not similar with the signal at the current position. This process is computed with a wavelet which is scaled and translated, creating an output plot where every output is placed according to the current scale and translation of the wavelet. The continuous wavelet transform, Section 4.2.1, calculates the wavelet transform on an infinite signal. Wavelets at all scales and translations are convolved along

[1] A spectrogram is a graphic representation of a spectrum. In this case the result after calculating the frequency spectrum of the windowed frames of the signal.

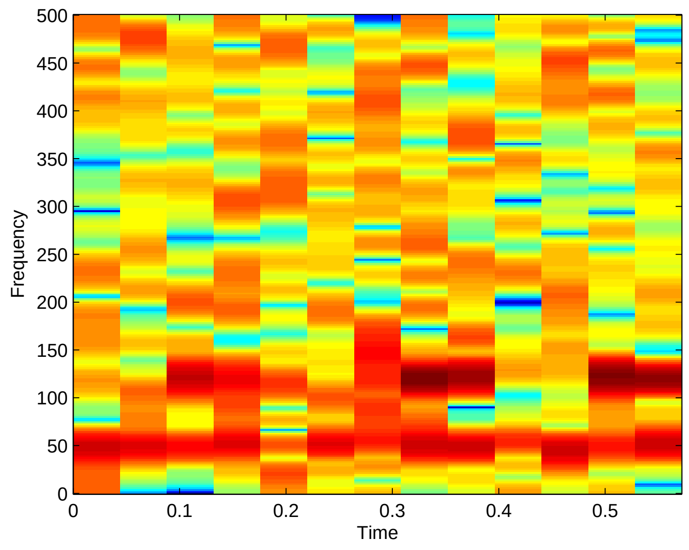


Figure 2.3: Spectrogram of STFT example.

the signal, creating a wavelet transform plot, Figure 2.4. With the example signal, the output looks like in Figure 2.5. The color scale is the same as for the STFT. Blue means that the frequency is not present at that time, while red means that the frequency is definitely in the signal at current time. The example input signal contains a lot of noise, which is why the output is a bit blurry.

Another wavelet transform is the discrete wavelet transform (DWT), which calculates the wavelet transform on a signal with finite length. The discrete wavelet transform performs one convolution with a high-pass filter, and one convolution with a low-pass filter at each step. Each step represents one line of the output plot. The result from the convolution with the low-pass filter is used in the next step, while the output from the convolution with the high-pass filter is saved. Section 4.2.3 describes how to find the filters and also explains more about the computation of the discrete wavelet transform. The output at each step in the discrete wavelet transform is down-sampled, so that the output from the two convolutions together have the same length as the input. The down-sampling process gives a result that is not accurate, as you would get different result if you keep every even or every odd value. This is where the stationary wavelet transform (SWT), Section 4.2.4, is presented.

The difference between the stationary wavelet transform and the discrete wavelet transform is that the stationary wavelet transform skips the

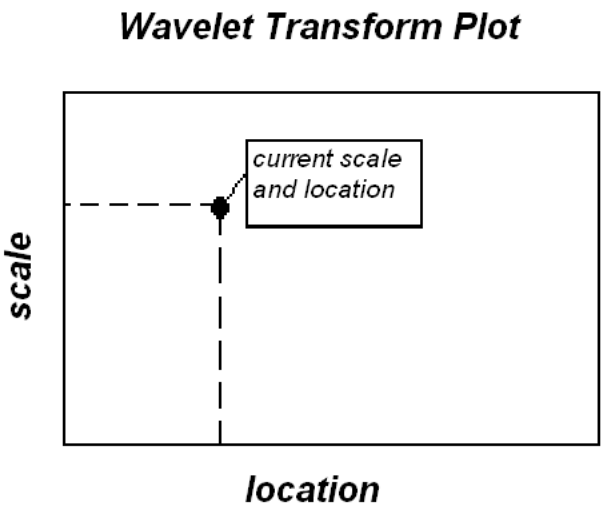


Figure 2.4: Wavelet Transform Plot

down-sampling. For every step, two outputs of the same length as the input are produced, providing an accurate but redundant result.

Section 4.3 describes a different method to decompose the signal. The matching pursuit method uses a dictionary of wavelets to one step at a time, reduce the signal with the best fitting wavelet until the signal is completely decomposed.

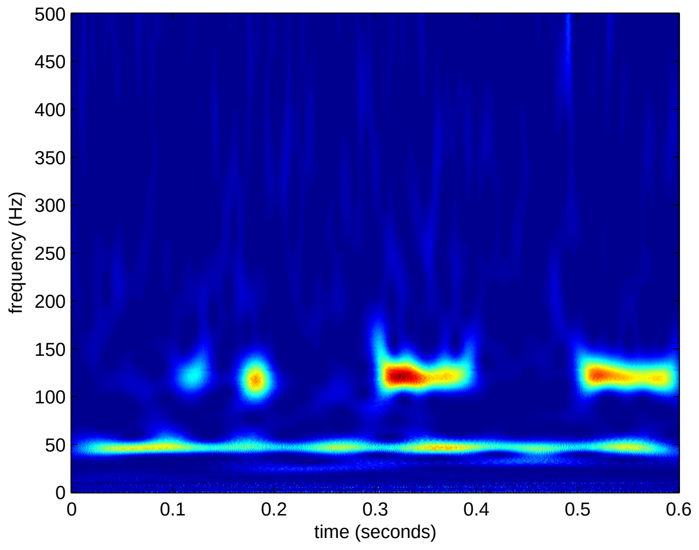


Figure 2.5: CWT of example signal.

Chapter 3

Wavelets

This chapter gives a short introduction to some of the most known wavelets, and Section 3.2 lists some of the requirements needed for a function to be a wavelet. The wavelet theory is a field in constant development, and the most useful wavelets were not seen until the late 1980's. But what exactly is a wavelet, and why use them?

A wavelet has a wave form concentrated in time, in other words a short wave. This is illustrated in Figure 3.1, where the sinusoid on the left extends infinitely in time, while the wavelet on the right is approximately zero outside the wave. A function which is continuous in time or space, like for example a sinusoid, can be described as a wave since it is oscillating. The word wavelet comes from the fact that small waves increase and decrease in size over short time periods. The idea that a small wave changes is transferred to the wavelet transform, see Section 4.1.1, as a wavelet easily is translated and dilated before applied to a problem.

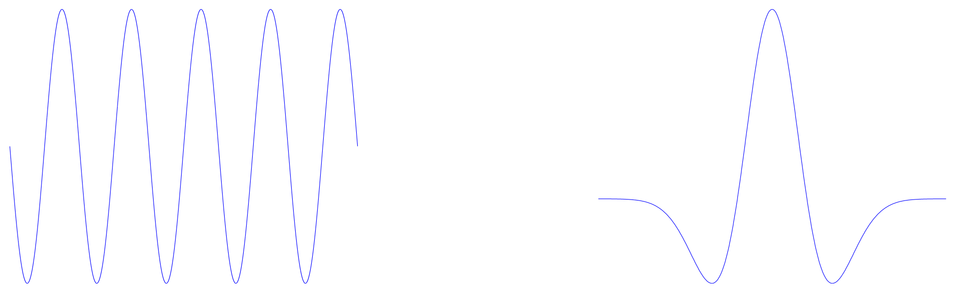


Figure 3.1: A sinusoid wave versus a Mexican hat wavelet.

Wavelets are very useful when it comes to representing functions. Not only because of their ability to place the signal properties both in time and frequency, but also because this can be done effectively and accurately when using wavelets. Almost any function can be approximated accurately with wavelets, because there exist many different wavelets and there usually is a wavelet that has some similar properties as the function. The sinusoids used in the Fourier transform are of infinite length, and it is therefore more complicated to approximate a function property like a sharp edge. The wavelets used for the wavelet transform are smaller and shorter, and can be started and stopped wherever or whenever you would prefer. A sharp corner can therefore more easily be matched.

A few examples of wavelets are discussed in Section 3.1 to give an idea of the differences between various wavelets.

More information about wavelets can be found in [BGG98] and [Add02].

3.1 Examples of wavelets

There are many different wavelets, like the Haar wavelet, the Mexican hat wavelet, the Morlet wavelet, and the Daubechies' wavelet [I. 92], among others. Wavelets can be a very powerful tool if used properly, as they are very effective when decomposing signals. The different wavelets have different properties. Some are good for signals with sharp edges, while others are better for smooth signals. Which wavelet you should use depends on the problem you are facing. This section gives an example of some of the wavelets that exist. More details can be found in [Add02].

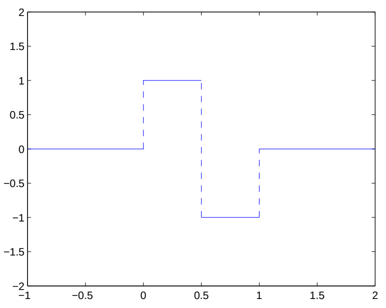
The Haar wavelet

The Haar wavelet in Figure 3.2 is the simplest orthonormal wavelet, and can be defined as a step function $\psi(t)$:

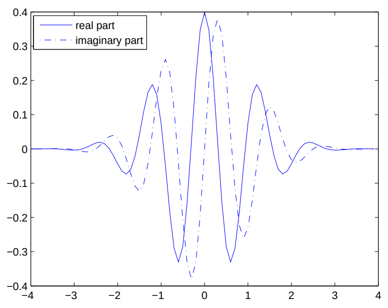
$$\psi(t) = \begin{cases} 1 & 0 \leq t < 1/2, \\ -1 & 1/2 \leq t < 1, \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

The Haar mother wavelet[1] can be described as two unit block pulses next to each other, where one of the blocks is inverted. The Haar wavelet has compact support, since it is zero outside the unit interval. This also means that it has a finite number of scaling coefficients. More about scaling functions in Section 4.2.3.

[1] A mother wavelet is the basis wavelet function, which can be translated and dilated to form a family of wavelets.



(a) Haar wavelet



(b) Morlet wavelet

Figure 3.2: Two example wavelets

The Mexican hat wavelet

All derivatives of the Gaussian distribution function $e^{-\frac{t^2}{2}}$ can be used as wavelets, but normally only the first and the second are used in practice. The Mexican hat wavelet seen in Figure 3.1, is Gauss' second derivative, and is the Gaussian derivative most commonly used as a wavelet. The equation for the Mexican hat wavelet is:

$$\psi(t) = (1 - t^2)e^{-\frac{t^2}{2}}. \tag{3.2}$$

The Morlet wavelet

The Morlet wavelet is the most frequently used complex wavelet, and is defined:

$$\psi(t) = \pi^{-\frac{1}{4}} e^{i2\pi f_0 t} e^{\frac{-t^2}{2}}, \tag{3.3}$$

where f_0 is the central frequency while the factor $\pi^{-\frac{1}{4}}$ ensures that the wavelet has unit energy. Using a complex wavelet makes it possible to separate the phase and amplitude in the signal, [Add02, page 35]. The complex transform values that result from performing the wavelet transform with the Morlet wavelet on a signal, show that the imaginary part is phase shifted[2] by one quarter of a cycle. In other words, the imaginary part has the best match with the signal one quarter of a cycle later because the imaginary part is inverted when doing the wavelet transform. This ability makes it easier to find discontinuities in the signal. (More about

[2]Phase is the current position in a cyclic changing signal, while phase shift is the constant difference between two existing phases.

the wavelet transform can be found in Section 4.2.) The Morlet wavelet has proved to work well with problems like audio and image enhancements [HRMS04]. The Morlet wavelet in Equation (3.3) can also be described as the real part

$$\psi(t) = \pi^{-\frac{1}{4}} e^{\frac{-t^2}{2}} \cos(2\pi f_0 t), \tag{3.4}$$

and the imaginary part

$$\psi(t) = \pi^{-\frac{1}{4}} e^{\frac{-t^2}{2}} \sin(2\pi f_0 t). \tag{3.5}$$

An example of the Morlet wavelet can be seen in Figure 3.2.

3.2 Requirements of a wavelet

Addison [Add02, page 9] writes that three requirements have to be met in order for a function to be a wavelet:

1. First of all, a wavelet needs to have finite energy:

$$E = \int_{-\infty}^{\infty} |\psi(t)|^2 dt < \infty, \tag{3.6}$$

   where E is the energy of a function equal to the integral of its squared magnitude and the vertical brackets $|.|$ represent the modulus operator which gives the magnitude of $\psi(t)$. If $\psi(t)$ is a complex function the magnitude must be found using both its real and complex parts.

2. The second criteria is that if $\psi(t)$ has the Fourier transform[3] $\hat{\psi}(f)$

$$\hat{\psi}(f) = \int_{-\infty}^{\infty} \psi(t) e^{-i(2\pi f)t} dt, \tag{3.7}$$

   then the following must hold:

$$C_g = \int_{0}^{\infty} \frac{|\hat{\psi}(f)|^2}{f} df < \infty. \tag{3.8}$$

   The wavelet has no zero frequency component, $\hat{\psi}(0) = 0$, which means that the wavelet $\psi(t)$ must have zero mean. Equation (3.8) is known as the admissibility condition and C_g is called the admissibility constant. The value of C_g depends on the chosen wavelet, and is equal to π for the Mexican hat wavelet given in Equation (3.2).

[3]See Section 2.3 for more about the Fourier transform.

3. An additional criterion that must hold for complex wavelets is that the Fourier transform must both be real and vanish for negative frequencies.

These criteria should be followed if an appropriate result is to be expected. A function is infinite in time if the first criterion is not followed, and hence not a wavelet. It is possible not to follow these criteria strictly, but in those cases extra caution is recommended as unpredicted results may appear. Wilson [Wil02] has an example where a proper result is computed even though the requirements are only followed loosely.

Wavelets satisfying (3.8) are bandpass filters. A bandpass filter lets through signal components within a finite range of frequencies, and tries to discard the components outside the range. The range is decided by an upper and a lower cutoff frequency value, where the bandwidth of the filter is the difference between the two cutoff frequencies. Figure 4.5 illustrates how a bandpass filter can be created using a high-pass and a low-pass filter. The low-pass lets low frequencies through, while the high-pass lets high frequencies through. Combining these two, results in a bandpass filter.

Chapter 4

Wavelet Transforms

This chapter describes some of the different wavelet transforms. The wavelet transform is similar to the Fourier transform in Section 2.3, but the wavelet transform uses a family of wavelets, described in Section 4.1, instead of sinusoids. A kind of approximation to the wavelet transform is explained in Section 2.4 with the short-time Fourier transform (STFT). The STFT decomposes the signal using a constant window size, but with better time resolution than the Fourier transform, and therefore provides a transform which lies between the Fourier transform and the wavelet transforms.

My thesis is about the stationary wavelet transform (SWT), but as SWT is an enhancement of other wavelet transforms, some background information on other wavelet transforms is required to get a proper understanding of the method. First comes the continuous wavelet transform (CWT), Section 4.2.1, which does the wavelet transform on a continuous signal. The time-frequency map from CWT in Section 4.2.2 is an improvement of the CWT.

Computing the continuous wavelet transform can not be done on a computer, and the discrete wavelet transform (DWT) provides a transform which can be computed on a discrete signal. The SWT is very similar to the DWT, but is said to be more accurate as it does not down-sample[1] the result as with the DWT.

Another wavelet-based method uses a selected dictionary of wavelets it convolves with the signal to find where they best match. Then the residue signal is convolved with another wavelet, and so on until the signal is decomposed. This method is called matching-pursuit. The matching pursuit method is used with different dictionaries like Gabor and Morlet,

[1]When down-sampling a signal, the signal is shortened by for example only keeping every second sample, leaving a result that may have lost important information.

and is described in Section 4.3.

When decomposing a signal, each part of the signal is divided into a selection of frequencies, which helps interpret the data. Section 4.4 gives an example of how this is done.

Some notation used in this section : The space $L^2(\mathcal{R})$ is the Hilbert space of complex-valued functions with a well defined integral of the square of the modulus of the function:

$$||x||^2 = \int_{-\infty}^{+\infty} |x(t)|^2 dt < +\infty. \tag{4.1}$$

The inner product of $< x, g >\in L^2(\mathcal{R})^2$ is defined by:

$$< x, g >= \int_{-\infty}^{+\infty} x(t)\bar{g}(t)dt, \tag{4.2}$$

where $\bar{g}(t)$ is the complex conjugate of $g(t)$. The Fourier transform of $x(t) \in L^2(\mathcal{R})$ is written $\hat{X}(\omega)$, and is defined as:

$$\hat{X}(\omega) = \int_{-\infty}^{+\infty} x(t)e^{-i\omega t}dt. \tag{4.3}$$

4.1 Wavelet systems

Before computing the wavelet transform, at least two decisions have to be made; which wavelet, and what kind of wavelet transform. Which wavelet to use depends on the signal, and on what you would like to accomplish with the transform. The requirement of a wavelet described in Section 3.2 should be followed when choosing the function to be used as the wavelet. Section 4.2 represents some of the properties for a handful of different wavelet transforms that can be considered when defining which wavelet transform to choose.

For a continuous signal, the wavelet transform is defined as

$$T(\sigma, \tau) = \omega(\sigma) \int_{-\infty}^{\infty} x(t)\bar{\psi}\left(\frac{t-\tau}{\sigma}\right) dt, \tag{4.4}$$

where the weighting function $\omega(\sigma)$ typically is set to $1/\sqrt{\sigma}$ for energy conservation reasons, and $\bar{\psi}$ denotes the complex conjugate. Doing a cross-correlation (4.5) of a signal with a set of wavelets with various widths, is another way to explain how to perform the wavelet transform. Cross-correlation is like convolution (4.6) without reversing the wavelet function

g. Instead the wavelet function is just shifted across the signal x generating an output at every step.

$$(x \star g)(i) \overset{\text{def}}{=} \int x(\bar{t})\, g(i+t)dt, \tag{4.5}$$

$$(x * g)(i) = \int x(t)g(i-t)dt. \tag{4.6}$$

The wavelet transform can be reversed by doing an inverse transform to get back to the original signal. The inverse wavelet transform is written:

$$x(t) = \frac{1}{C_g} \int_{-\infty}^{\infty} \int_{0}^{\infty} T(\sigma, \tau)\psi\left(\frac{t-\tau}{\sigma}\right) \frac{d\sigma d\tau}{\sigma^2}. \tag{4.7}$$

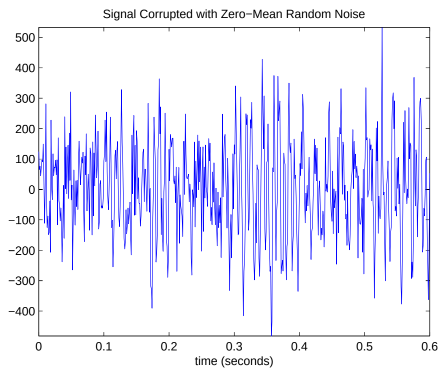
Where C_g is the admissibility constant from Equation 3.8. Another thing that should be noted in the inverse wavelet transform is $\frac{1}{\sigma^2}$. The coefficients which are multiplied with the wavelet functions to reconstruct the signal x, are the wavelet coefficients divided by the square root of σ. Each contribution from ψ in the reconstruction of x, are given by $T(\sigma, \tau)/|\sigma|^2$. In other words, $T(\sigma, \tau)/|\sigma|^2$ provides information on how much of each component ψ exist in the signal x. Integrating over all scales and locations, σ and τ, recreates the original signal. By limiting the scale over a range, the original signal gets filtered, which is illustrated in Figure 4.1. The coefficients above 150 Hz are set to zero, leaving an output where some of the noise is truncated from the original signal.

With wavelet analysis, the set of windows[2] used when decomposing the signal quickly decays to zero, because the windows have compact support[3] in time [CO95]. A broad time domain gives an overview of the signal structure, while a narrow analysis window shows more detailed characteristics. How the wavelet changes according to how the window size varies for some different methods is illustrated in Figure 4.4. As it can be seen, only the wavelet transform uses windows with various sizes, thereby creating good frequency resolution for low frequencies, and good time resolution for high frequencies. This helps detecting rapid changes, which are the fact when high frequencies are represented, and changes over time as with low frequencies. According to [BGG98, page 3] there are three general properties that can be used to identify a wavelet system:

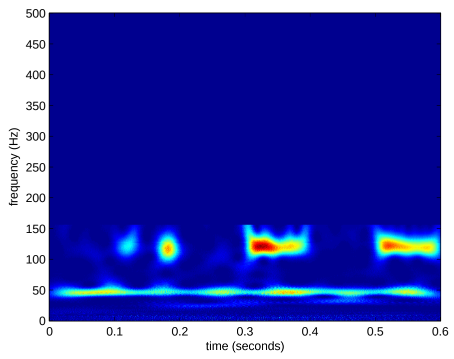
1. A wavelet system is a collection of basis functions that together can represent any signal or function. The set of wavelets is written $\psi_{j,k}(t)$

[2]A window can be represented by a wavelet, where a narrow window represents good time resolution, and a wide window gives good frequency resolution.

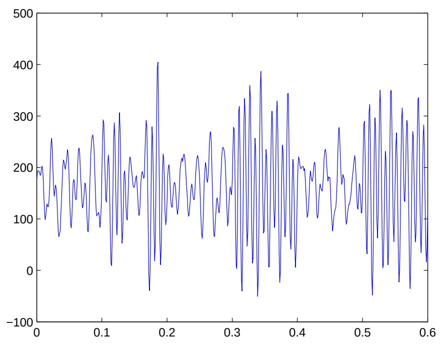
[3]Compact support means that the function is non-zero in a finite time space.



(a) Original signal with noise.



(b) Cut frequencies above 150 Hz.



(c) Reproduced signal.

Figure 4.1: Denoised signal

for $j, k = 1, 2, ...$, which for a set of coefficients $a_{j,k}$ has a linear expansion $x(t) = \sum_k \sum_j a_{j,k}\psi_{j,k}(t)$. For a class of one- (or higher) dimensional signals, the wavelet system is a two-dimensional expansion set.

2. The wavelet expansion provides a time-frequency localization of the signal, as a few coefficients $a_{j,k}$ can represent most of the signal energy.

3. The coefficients can be calculated efficiently since many wavelet transforms are calculated with $O(N)$ operations. The general wavelet transforms needs $O(N\log(N))$ operations, which is the same as what the Fast Fourier Transform uses.

A wavelet system is really just another word for a wavelet transform, but while the word transform usually is associated with only the function, the wavelet system includes the whole package with the function, wavelet and coefficients.

Mathematically, the wavelet transform is a convolution of the signal with the wavelet function. A large value is returned from the transform if the wavelet matches the signal, otherwise, a low value is produced. Figure 4.2 gives an example of how the wavelet transform works. A Mexican hat wavelet is convolved with the signal, which in this case is a sinusoid. It can be seen from the figure that the wavelet correlates well at location wletA, but very poorly at wletB. The '+' and '-' indicates if positive or negative values are produced.

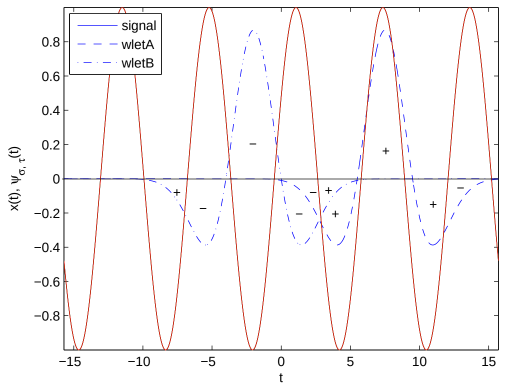


Figure 4.2: Example of a wavelet convolved with a sinusoid

Why wavelet analysis is effective

Burrus et al [BGG98, page 6] use the following properties to explain why wavelet analysis is effective.

1. Wavelets are very effective in signal and image compression, denoising, and detection, because the size of the wavelet expansion coefficients decreases quickly.

2. The wavelet expansion provides a more accurate local description and separation of signal characteristics than the Fourier coefficients. A Fourier coefficient is a component that does not change, and temporary events have to be described by a phase characteristic that allows cancellation or reinforcement over large time periods. A wavelet expansion coefficient component is local and easy to interpret, and

also allows a separation of components of a signal to overlap in both time and frequency.

3. Wavelets can be created to fit individual applications, since there exist many different wavelets, that are all adjustable and adaptable. Wavelets are therefore very useful for adaptive systems that adjust themselves to suit the signal.

4. When generating a wavelet and calculating the discrete wavelet transform, only multiplications and additions are used. This means that only operations that are basic to a digital computer are applied, which makes wavelets efficient for computer programs.

These properties are explained in the following sections.

4.1.1 A family of wavelets

In a wavelet transform, a family of wavelets is created in order to compute the wavelet transform. A function $\psi(t) \in L^2(\mathcal{R})$ in both time and frequency with a zero mean, is the definition of a wavelet. A family of wavelets is made by dilating (scaling) and translating a mother wavelet $\psi(t)$:

$$\psi_{\sigma,\tau}(t) = \frac{1}{\sqrt{\sigma}}\psi\left(\frac{t-\tau}{\sigma}\right), \tag{4.8}$$

where $\sigma, \tau \in \mathcal{R}$, $\sigma \neq 0$ is the dilation parameter and τ the translation parameter. When making a wavelet family, you first choose which mother wavelet to use, and then use (4.8) to create a family of wavelets. An example can be seen in Figure 4.3, where the Mexican hat mother wavelet (4.9) is dilated and translated in order to create a family of wavelets,

$$\psi\left(\frac{t-\tau}{\sigma}\right) = \left(1-\left(\frac{t-\tau}{\sigma}\right)^2\right)e^{-\frac{\left(\frac{t-\tau}{\sigma}\right)^2}{2}}. \tag{4.9}$$

4.2 The Wavelet Transform

Chakraborty et al [CO95] and Castagna et al [SRAC05] write about spectral decomposition of seismic data with the continuous wavelet transform.

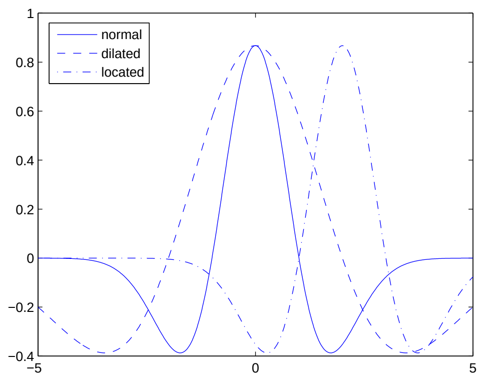


Figure 4.3: Example of dilation and translation

The continuous wavelet transform (CWT), Section 4.2.1, makes a time-scale map called a scalogram[4] instead of a spectrogram. Dilation and translation of wavelets, as with for example the CWT, produces the scalogram describing the time-scale map, while the spectrogram describes the time-frequency map calculated with a fixed time-frequency resolution. Both Abry et al[AGF93] and Hlawatsch et al [HBB92] explain methods that represent the scalogram as a time-frequency map by saying that scale is inversely proportional to the center frequency of the wavelet.

Another method to map the scalogram into a time-frequency map is called time-frequency CWT (TFCWT) is described in Section 4.2.2. The time-frequency continuous wavelet transform gives a high frequency resolution at low frequencies and high time resolution at high frequencies. The TFCWT can reconstruct the original signal as long as the inverse wavelet transform exists. It is also a fast computational process in Fourier domain, as usually only the forward transform is needed.

The discrete Fourier transform approximates the continuous computation by calculating with discrete functions. The same can be done in wavelet transformation using the discrete wavelet transform described in Section 4.2.3 to approximate the CWT. Section 4.2.4 introduces the stationary wavelet transform, which is an extension to the DWT.

[4]A plot of $E(\sigma, \tau) = |T(\sigma, \tau)|^2$, and highlights the dominant energetic features of the signal at the representative scale and dilation[Add02, page 29]

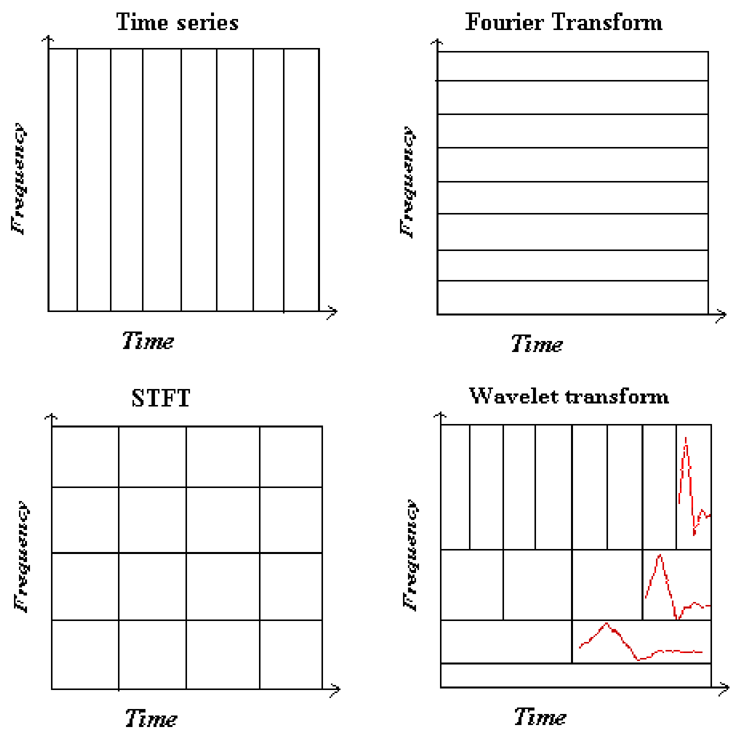


Figure 4.4: The idea behind windowing.

4.2.1 The Continuous Wavelet Transform (CWT).

The continuous wavelet transform is seen as the convolutions you get from:

$$T(\sigma,\tau) = \frac{1}{\sqrt{\sigma}} \int x(t)\bar{\psi}\left(\frac{t-\tau}{\sigma}\right) dt, \tag{4.10}$$

where σ is the scale and τ the translation. The bandwidth of the window is narrow when the scale index is low. When the scale index increases, the bandwidth of the window increases, and the time-domain width becomes narrow.

The CWT works similar to the STFT as they both make a 2D space from a 1D signal, but the CWT has better frequency resolution for low frequencies, and it provides better time resolution for higher frequencies as illustrated in Figure 4.4.

The modulated Gaussian defined in Morlet et al [JG82] is one example of a kernel wavelet:

$$\psi(t) = \int e^{ivt} e^{\frac{-t^2}{2}} dt < \infty, \tag{4.11}$$

where

$$v \geq 5,$$

and the requirements of the wavelet in Section 3.2 are met.

Step-by-step calculating the CWT

Calculation of the continuous wavelet transform can be described by the following steps:

1. A wavelet at scale $\sigma = 1$ is placed at the beginning of the signal.
2. The wavelet function at $\sigma = 1$ is multiplied by the signal and integrated over all times. Then multiplied by $1/\sqrt{\sigma}$.
3. Shift the wavelet to $t = \tau$, and get the transform value at $t = \tau$ and $\sigma = 1$.
4. Repeat the procedure of Steps 2 and 3 until the wavelet reaches the end of the signal.
5. Increase scale σ by a sufficiently small value, and repeat the above procedure for all σ.
6. Each computation for a given σ fills a single row of the time-scale map.
7. CWT is obtained when all values of σ are calculated.

These steps should not be too hard to follow, but as this is computed on a continuous signal, and therefore with an infinite number of steps, the computations can not be followed directly when calculating with a computer. Section 4.2.3 presents the discrete wavelet transform, which can be used when calculating the wavelet transform on a discrete signal.

4.2.2 Time-Frequency Map from CWT (TFCWT)

With the CWT, changes in frequency is supported in time because of the way the wavelets dilate. Time resolution increases while frequency resolution decreases and the other way around, as described in Castagna et al [SRAC05].

Recall Equation (4.8) for a wavelet family. The continuous wavelet transform is the inner product of a family of wavelets $\psi_{\sigma,\tau}(t)$ with the signal $x(t)$:

$$T(\sigma,\tau) =< x(t), \psi_{\sigma,\tau}(t) >= \int_{-\infty}^{\infty} x(t)\frac{1}{\sqrt{\sigma}}\bar{\psi}\left(\frac{t-\tau}{\sigma}\right)dt, \tag{4.12}$$

where $\bar{\psi}$ is the complex conjugated of ψ. We use Calderon's identity [I. 92] to reconstruct the signal $x(t)$ from the wavelet transform and get:

$$x(t) = \frac{1}{C_\psi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} T(\sigma,\tau)\psi\left(\frac{t-\tau}{\sigma}\right)\frac{d\sigma}{\sigma^2}\frac{d\tau}{\sqrt{\sigma}}. \tag{4.13}$$

To be able to find the inverse transform, the analyzing wavelet has to satisfy the admissibility condition in Equation (3.8).

A scale represents a frequency band, so some different approaches have to be used to interpret the time-scale map into a time-frequency map. The easiest approach is to just stretch the scale to fit the equivalent frequency, but a better way is to use the wavelet as an adaptive window to find the spectrum of a signal. We can look at the frequency content at different times, because of the translation characteristic. This provides a time-frequency map, which is adaptive to seismic signals, by computing the Fourier transform of the inverse continuous wavelet transform. Mathematically this can be described by first substituting $x(t)$ from (4.13) into (2.2):

$$\hat{X}(\omega) = \frac{1}{C_\psi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{1}{\sigma^2\sqrt{\sigma}}T(\sigma,\tau)\psi\left(\frac{t-\tau}{\sigma}\right)e^{-i\omega t}d\sigma d\tau dt. \tag{4.14}$$

Then use the scaling and shifting theorem of the Fourier transform:

$$\int_{-\infty}^{\infty}\psi\left(\frac{t-\tau}{\sigma}\right)e^{-i\omega t}dt = \sigma e^{-i\omega\tau}\hat{\psi}(\sigma\omega), \tag{4.15}$$

and interchange the integrals and substituting (4.15) into (4.14):

$$\hat{X}(\omega) = \frac{1}{C_\psi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{1}{\sigma^2\sqrt{\sigma}}T(\sigma,\tau)\sigma\hat{\psi}(\sigma\omega)e^{-i\omega\tau}d\sigma d\tau, \tag{4.16}$$

where $\hat{\psi}(\omega)$ is the Fourier transform of the mother wavelet. The last step is to remove the integration over τ and replace $\hat{X}(\omega)$ with $\hat{X}(\omega, \tau)$ to get a time-frequency map:

$$\hat{X}(\omega, \tau) = \frac{1}{C_\psi} \int_{-\infty}^{\infty} T(\sigma, \tau) \hat{\psi}(\sigma\omega) e^{-i\omega\tau} \frac{d\sigma}{\sigma^{\frac{3}{2}}}. \tag{4.17}$$

The time-frequency spectrum can be found from the continuous wavelet transform (TFCWT) of a signal. The time summation of (4.17) is the Fourier transform of the signal. There are two steps involved to reconstruct the signal. First time summation of the TFCWT, and then inverse Fourier transform of the resultant sum.

4.2.3 The Discrete Wavelet Transform (DWT).

To get an approximated result of the CWT when computing wavelet transforms, the discrete wavelet transform (DWT) can be used like the discrete Fourier transform is used when computing an approximation to the Fourier transform. The equation for a discrete approximation to the signal $x(t)$ is written

$$x(t) = \sum_{j,k} a_{j,k} \psi_{j,k}(t), \tag{4.18}$$

where the coefficients $a_{j,k}$ are called the DWT of the signal $x(t)$. The idea behind the DWT is the same as with the CWT, but the methods are different.

The CWT convolves the wavelet directly with the signal, while the DWT convolves the input signal simultaneously with a low-pass and a high-pass filter. The two filters are related and satisfies the criteria for the quadrature mirror filter (QMF) presented in the next paragraph. Figure 4.5 illustrates the idea behind the QMF. A low-pass filter is mirrored to make a high pass filter. Combining the two creates a bandpass filter to let through only certain frequencies.

The QMF [NS] is constructed by using a low pass filter, defined by a sequence g_n, where there is typically only a few non-zero values. Then a high-pass filter with the sequence h_n is built by using the low-pass values as

$$h_n = (-1)^n g_{1-n}. \tag{4.19}$$

Both filters satisfy the internal orthogonality

$$\sum_n h_n h_{n+2j} = 0, \tag{4.20}$$

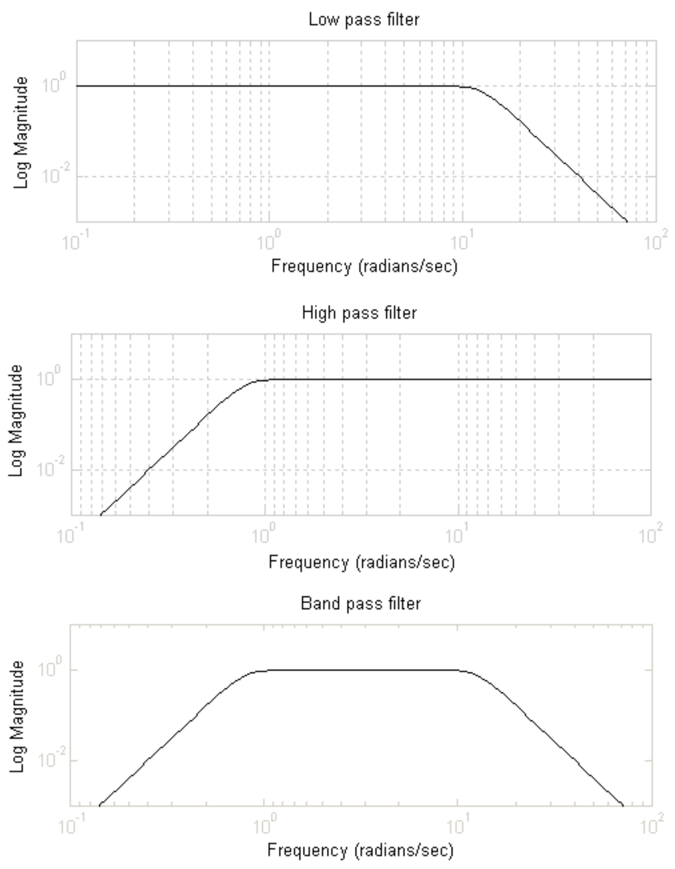


Figure 4.5: Quadrature mirror filter.

for all integers $j \neq 0$, and have the sum of squares

$$\sum_n h_n^2 = 1. \tag{4.21}$$

The mutual orthogonality relation

$$\sum_n h_n g_{n+2j} = 0 \tag{4.22}$$

for all integers j must also be satisfied.

The length of each filter is half the length of the signal. After convolving both filters with the signal, both outputs are down-sampled by a factor of two. The two outputs combined have the length of the input signal. The output after doing the high-pass filtering is called detail coefficients, and the output after the low-pass filtering is called approximation coefficients. The process is seen in Figure 4.6. The figure shows a filter bank, which is a tree-structured array of filters that separates the input signal into several components. The output components at each level can be filtered further, leading to the tree-structured figure. The decomposition can be repeated to increase the frequency resolution. The approximation coefficients is the input for the next decomposition level, and the calculations can be repeated until the output is of length one. The initial low-pass filter is constructed using the scaling function described later.

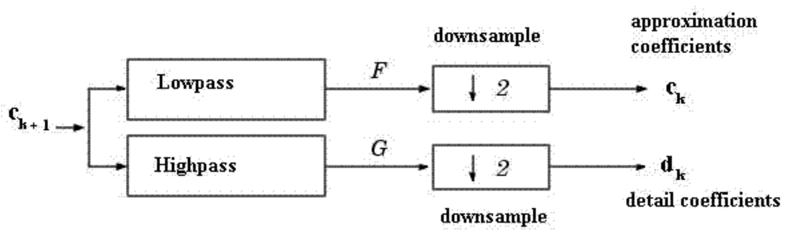


Figure 4.6: Filter bank for DWT

A wavelet function with dilation σ and translation τ is defined in Equation (4.8). Sample the parameters σ and τ with a logarithmic discretization of the dilation σ. Then link to the translation parameter τ, by moving in discrete steps to each τ, which is proportional to the dilation σ, to get a discretized wavelet function:

$$\psi_{m,n}(t) = \frac{1}{\sqrt{\sigma_0^m}}\psi\left(\frac{t - n\tau_0\sigma_0^m}{\sigma_0^m}\right), \tag{4.23}$$

where $\sigma_0 > 1$ and $\tau_0 > 0$, and dilation and translation are determined by m and n. Then the wavelet transform with discrete wavelets of a continuous signal $x(t)$ is defined:

$$T_{m,n} = \int_{-\infty}^{\infty} x(t)\frac{1}{\sqrt{\sigma_0^m}}\psi(\sigma_0^{-m}t - n\tau_0)dt. \tag{4.24}$$

The discrete wavelet transform values $T_{m,n}$, also called wavelet coefficients or detail coefficients, are given on a dilation-translation grid over m, n. The inverse discrete wavelet transform is formulated

$$x(t) = \sum_{m=-\infty}^{\infty}\sum_{n=-\infty}^{\infty} T_{m,n}\psi_{m,n}(t). \tag{4.25}$$

Dyadic grid scaling

The dyadic grid [Add02, page 67] is one of the simplest and most efficient discretization for practical cases, and it is therefore also the most commonly used method to construct an orthonormal wavelet basis. You get a dyadic grid by choosing the discrete wavelet parameters to be $\sigma_0 = 2$ and $\tau_0 = 1$. Equation (4.23) can then be written as the dyadic grid wavelet

$$\psi_{m,n}(t) = \frac{1}{\sqrt{2^m}} \psi\left(\frac{t - n2^m}{2^m}\right), \tag{4.26}$$

or more compact:

$$\psi_{m,n}(t) = 2^{\frac{-m}{2}} \psi(2^{-m}t - n). \tag{4.27}$$

Then we can write the Discrete Wavelet Transform with the dyadic grid wavelet (4.26) as

$$T_{m,n} = \int_{-\infty}^{\infty} x(t)\psi_{m,n}(t)dt. \tag{4.28}$$

Since we are now using an orthonormal wavelet basis, the inverse discrete wavelet transform with wavelet coefficients $T_{m,n}$ is defined:

$$x(t) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} T_{m,n}\psi_{m,n}(t). \tag{4.29}$$

The scaling function

Orthonormal dyadic discrete wavelets are linked with scaling functions and their dilating equations [Add02, page 69]. The DWT can be obtained by relating it to the scaling equation and the wavelet equation. A scaling function is built at one scale from a number of scaling equation from the previous scale. The scaling function is convolved with the signal to produce the approximation coefficients that are used when computing the next step of the discrete wavelet transform. This chapter presents the properties of the scaling function, while the scaling equation is described in more detail in the next section.

The scaling functions have two main properties. The first property is that the scaling function $\phi(t)$ and its integer translates $\phi(t + j)$ forms an orthonormal set in L^2 for all j. The second is that ϕ can be written as a linear combination of half-integer translates of itself at double scale. The smoothing of the signal associated with the scaling functions is written like the wavelet form:

$$\phi_{m,n}(t) = 2^{\frac{-m}{2}} \phi(2^{-m}t - n), \tag{4.30}$$

with the property

$$\int_{-\infty}^{\infty} \phi_{0,0}(t)dt = 1. \tag{4.31}$$

The function $\phi_{0,0}(t) = \phi(t)$ can sometimes be called the father scaling function. The scaling function is orthogonal to translations of itself, but not to dilations of itself. Convolving the scaling function with the signal produces approximation coefficients

$$S_{m,n} = \int_{-\infty}^{\infty} x(t)\phi_{m,n}(t)dt. \tag{4.32}$$

The signal can, with a smooth, scaling-dependent version of the signal $x(t)$ at scale m, have a continuous approximation:

$$x_m(t) = \sum_{n=-\infty}^{\infty} S_{m,n}\phi_{m,n}(t). \tag{4.33}$$

This can be used when representing $x(t)$ as a series expansion, where both the approximation coefficients and the wavelet coefficients are used at an arbitrary scale m_0 like

$$x(t) = \sum_{n=-\infty}^{\infty} S_{m_0,n}\phi_{m_0,n}(t) + \sum_{m=-\infty}^{m_0} \sum_{n=-\infty}^{\infty} T_{m,n}\psi_{m,n}(t), \tag{4.34}$$

which with Equation (4.33) can be shortened to

$$x(t) = x_{m_0}(t) + \sum_{m=-\infty}^{m_0} d_m(t), \tag{4.35}$$

where

$$d_m(t) = \sum_{n=-\infty}^{\infty} T_{m,n}\psi_{m,n}(t) \tag{4.36}$$

is the signal detail at scale m.

The scaling equation

To connect the scaling function to the wavelet equation, we write the scaling equation which describes the scaling function $\phi(t)$:

$$\phi(t) = \sum_k c_k\phi(2t-k). \tag{4.37}$$

The changed version $\phi(2t-k)$ of $\phi(t)$ is shifted by an integer along the time axis, and multiplied by a scaling coefficient c_k. The scaling coefficients must fulfill the constraint

$$\sum_k c_k = 2, \tag{4.38}$$

and following equation has to be satisfied to be able to create an orthogonal system

$$\sum_k c_k c_{k+2k'} = \begin{cases} 2 & \text{if } k' = 0, \\ 0 & \text{otherwise.} \end{cases} . \tag{4.39}$$

The scaling coefficients c_k are used in reverse with alternate signs, as explained with the QMF, when creating the associated wavelet equation

$$\psi(t) = \sum_k (-1)^k c_{N_k-1-k}\phi(2t-k). \tag{4.40}$$

The orthogonality property between the wavelet and scaling function is by this guaranteed. The coefficients used in the wavelet equation (4.40) can more compactly be written $b_k = (-1)^k c_{N_k-1-k}$. With a wavelet with compact support, the finite number of scaling coefficients is denoted N_k.

The DWT calculation

The Discrete Wavelet Transform is a method that convolves the signal with a low-pass filter and a high-pass filter according to certain criteria, to expand to a digital signal. Redundant coefficients are removed with down-sampling at each step to get the outputs, the scaling coefficients c_k and the detail coefficients d_k. The process is illustrated in Figure 4.6, where the scaling coefficients are the input for the next level. This process makes sure that the number of coefficients output at each level is the same as the number of coefficients used as input. Down-sampling the output at each level can in some cases remove important information, and this is where the stationary wavelet transform discussed in Section 4.2.4 differs from the DWT.

4.2.4 Stationary Wavelet Transform (SWT)

The stationary wavelet transform (SWT) [NS] is as already mentioned an improvement of the discrete wavelet transform. Figure 4.7, illustrating the process of the SWT is very similar to Figure 4.6 describing the DWT process. The only difference is that the SWT does not perform down-sampling after every filtering step, and instead up-samples the filters at every step. Since the outputs do not get down-sampled,the SWT produces two outputs with the same amount of coefficients as components in the input signal at each step. This gives an redundant result where no valuable information is lost, which can be necessary for sensitive data. The stationary wavelet transform has many different names, as many developed the same idea of not down-sampling the output. Some examples are the redundant wavelet transform [CW06], the translation invariant wavelet transform [BW98], the shift invariant wavelet transform [GLOB95], the overcomplete discrete wavelet transform [ZLBN96], and undecimated discrete wavelet transform [LGO+96].

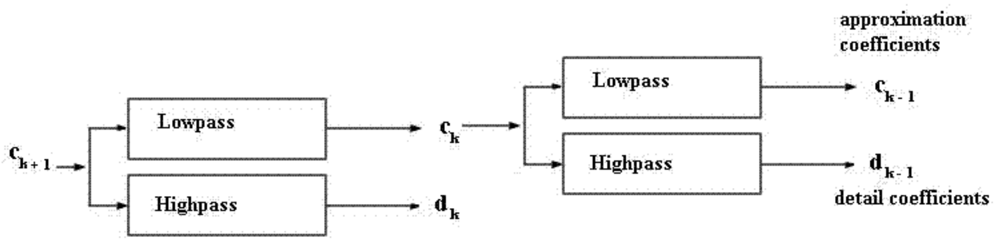


Figure 4.7: Filter bank for SWT

4.2.5 Transform overview

Various transforms have now briefly been described in previous sections. It may be hard to clearly visualize the differences between the different transforms. Like how the Fourier transform works compared to the wavelet transform, or how the short-term Fourier transform differ from the standard Fourier transform. All the transforms use some kind of a window to filter the signal. Figure 4.4 illustrates the window sizes of the different transforms. It can be seen that while the Fourier transform and the STFT use a constant window size, the wavelet transform change the window size to better place the signal properties. Interpreting what the signal represents is easier when the signal properties are placed properly according to frequencies and time.

4.3 Matching Pursuit with Time-Frequency Dictionaries

The matching-pursuit (MP) algorithm was first presented by Mallat and Zhang [MZ93]. When using the matching-pursuit method, any signal is decomposed to wavelets according to a given dictionary of wavelet functions. The signal has to be decomposed into something that is flexible enough for the signal to be rebuilt without any information loss.

Fourier bases are limited when it comes to representing a decomposed signal well localized in time, and wavelet bases have problems with Fourier transforms that support a narrow high frequency. The information is thinned out over the bases with both approaches, which makes it hard to find the signal patterns. High variation in time and frequency makes it especially important to have a flexible decomposition of the signal. To get proper results, the signal has to be decomposed into time-frequency atoms according to local structures. Matching pursuit selects waveforms from the given dictionary, as described in Section 4.3.2, that best match the structure of the signal. Convergence is guaranteed since it preserves the energy.

The matching-pursuit decomposition sub-decomposes the signal if necessary to get a good correlation with the dictionary at hand. The best adapted approximation is always chosen, making matching pursuit a greedy algorithm.

Section 4.3.1 contains some of the requirements when adapting the time-frequency decomposition to the signal structure, while the matching-pursuit algorithm is described in Section 4.3.2, with references to examples using Morlet and Gabor wavelets.

4.3.1 Time-Frequency Atomic Decomposition

Scaling, translating, and modulating a single window function $g(t) \in L^2(\mathcal{R})$ can produce a general family of time-frequency atoms[5]. Say that $g(t)$ is real and continuously differentiable. Also assume that $||g|| = 1$, that $\int g(t) \neq 0$, and $g(0) \neq 0$. For any scale $\sigma > 0$, frequency modulation ξ, and translation τ, set $\gamma = (\sigma, \tau, \xi)$ and define:

$$g_\gamma(t) = \frac{1}{\sqrt{\sigma}} g\left(\frac{t-\tau}{\sigma}\right) e^{i\xi t}. \tag{4.41}$$

[5]Each atom define one member of the dictionary.

By selecting a countable subset of atoms $(g_{\gamma n}(t))_{n \in N}$ with $\gamma_n = (\sigma_n, \tau_n, \xi_n)$, one is able to represent any function $x(t)$ as:

$$x(t) = \sum_{n=-\infty}^{+\infty} a_n g_{\gamma n}(t). \tag{4.42}$$

The window Fourier transform uses a constant scale $\sigma_n = \sigma_0$ for all the atoms $g_{\gamma n}(t)$, which means that it can only describe structures near the size σ_0.

Wavelets, on the other hand, decompose signals over time-frequency atoms with varying sizes, which is necessary to analyze structures of different forms. Frequency parameter $\xi_n = \frac{\xi_0}{\sigma_n}$, where ξ_0 is a constant, is used to build a wavelet family. Still this is not a very precise estimate of the frequency content, because it is not possible to define appropriate scale and modulation parameters a priori, but in this case it is good enough.

4.3.2 The Matching-Pursuit algorithm

A dictionary is defined as a family $D = (g_\gamma)_{\gamma \in \Gamma}$ of vectors in H (Hilbert space), with $||g_\gamma|| = 1$. The closed linear span of the dictionary vectors is called V, and is complete if $V = H$.

Theorem 4.1. *[FKK02] If D is a complete dictionary and if $x \in H$, then*

$$x = \sum_{k=0}^{\infty} < R^k x, g_{\gamma_k} > g_{\gamma_k} \tag{4.43}$$

and

$$||x||^2 = \sum_{k=0}^{\infty} | < R^k x, g_{\gamma_k} > |^2. \tag{4.44}$$

The linear expansion of x is approximated over a set of selected vectors from D with orthogonal projections on D's elements, to best match $x \in H$ structures. With $g_{\gamma 0} \in D$, the vector x can be decomposed to:

$$x =< x, g_{\gamma 0} > g_{\gamma 0} + Rx. \tag{4.45}$$

Here Rx is the vector that is left after locating x in the $g_{\gamma 0}$ direction. $g_{\gamma 0}$ is orthogonal to Rx, so

$$||x||^2 = | < x, g_{\gamma 0} > |^2 + ||Rx||^2, \tag{4.46}$$

where $| < x, g_{\gamma 0} > |$ has to be as large as possible to minimize $||Rx||$. The "best" vector g_{γ_0} can be found with:

$$| < x, g_{\gamma_0} > | = \max_{\gamma \in \Gamma_\alpha} | < x, g_\gamma > | \geq \alpha \sup_{\gamma \in \Gamma} | < x, g_\gamma > |. \tag{4.47}$$

The next step is to approximate Rx as was done with x, and so on, until a preset threshold is reached. The equation we get is:

$$x = \sum_{n=0}^{m-1} < R^n x, g_{\gamma n} > g_{\gamma n} + R^m x, \tag{4.48}$$

where $R^0 x = x$, which means that we do the decomposition up to order m. It is evident when looking at the above equation that reconstruction of the signal is not dependent of the order of elements. In finite space, Equation (4.48) can be written:

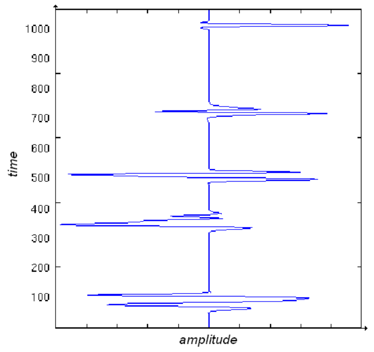
$$x = \sum_{n=0}^{m-1} < R^n x, g_{\gamma n} > g_{\gamma n}. \tag{4.49}$$

Examples of the Matching Pursuit algorithm used with Gabor dictionaries are described in [MZ93] and[FKK02]. The MP algorithm with Morlet wavelets can be found in [LM05] and [JG82].

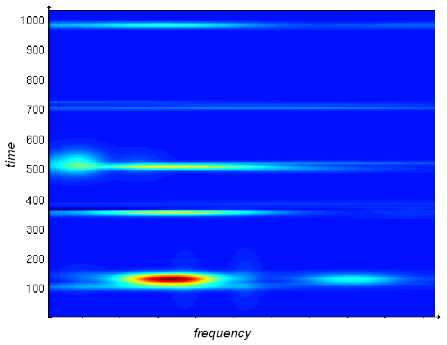
4.4 Instantaneous Spectral Analysis

Castagna et al [JPCS03] describe the instantaneous spectral analysis (ISA). ISA achieves excellent time and frequency localization by using a continuous time-frequency analysis technique that for each time-sample of a seismic trace provides a frequency spectrum. Castagna et al [JPCS03] have divided the ISA method into the three following steps:

1. Decompose the seismogram into constituent wavelets using wavelet transform methods such as Mallat's [MZ93][6] Matching Pursuit Decomposition.
2. Sum the Fourier spectra of the individual wavelets in the time-frequency domain to produce "frequency gathers".
3. Sort the frequency gathers to produce common (constant) frequency cubes, sections, time slices, and horizon slices.



(a) A synthetic input signal.



(b) Result when computing ISA.

Figure 4.8: An ISA example

There exists a number of different spectral decomposition methods. Most of the methods produce slightly different results, but none of the methods give a truly unique result. It is therefore important to use a method that captures the essential features. Castagna et al [JPCS03] found the most important criterions to be:

1. The sum of the time-frequency analysis over frequency should approximate the instantaneous amplitude of the seismic trace.

2. The sum of the time-frequency analysis over time should approximate the spectrum of the seismic trace.

3. Distinct seismic events should appear as distinct events on the time-frequency analysis. In other words, the vertical resolution of the time frequency analysis should be compared to the seismogram. The time duration of an event on the time-frequency analysis should not differ from the time duration on the seismogram.

4. Side lobes of events on the seismogram should not appear as separate events on the time-frequency analysis.

5. The amplitude spectrum of an isolated event should be undistorted. The spectrum should not be convolved with the spectrum of the window function.

6. There should be no spectral notches related to the time separation of resolvable events.

[6]Mallat's [MZ93] matching pursuit decomposition is described in Section 4.3.

The ISA technique is designed to meet Criteria (1) and (2). The ISA methods also meet Criteria (3) to (6) quite well, since the method does not involve windowing[7] of the seismogram. The best time-frequency representation is provided when using the most appropriate selection of the wavelet dictionary. Using an inappropriate selection of the wavelet dictionary will cause the method to fail meeting Criteria (3) and (4).

Figure 4.8 is an illustration on the result when computing the ISA. It can be seen in the figure how the frequencies of the input signal are placed according to when they occurred.

4.5 Overview

Many of the articles I considered concluded that matching pursuit is the most accurate method to represent time-frequency resolution. Methods like STFT and CWT are not capable of computing the same resolution as MP, since they are more restricted on choosing window size. Preset values for window size eliminate what parts of the input signal the method is able to represent properly. In CWT, high-frequency components are missing, while STFT is badly resolved in time. The matching-pursuit method finds the best approximation to the provided dictionary by finding the maximum $| < x, g_{\gamma 0} > |$ at each decomposition step. The articles presented two different dictionaries that were used with matching pursuit, Gabor and Morlet. Morlet can be used to find anomalies in the signal, while Gabor just decomposes the signal. The problem with the matching pursuit algorithm is that it can be computationally expensive, which is why the wavelet transform can be more popular to use when decomposing large datasets like seismic data.

Now that I have presented an overview of various wavelets and wavelet transforms the next step is to put some of it into practice. The next chapter explains some background information about the GPU and different toolkits that can be used. While Chapter 6 describes my implementation of the stationary wavelet transform.

[7] A window is convolved with the signal to find where the window matches the signal

Chapter 5

The GPU and programming tools

The graphics processing unit (GPU)[1] is computer hardware dedicated to graphics rendering. The GPU is either integrated on the motherboard or on the video card. The GPU is, as the name suggests, most commonly used to process graphics. A modern GPU has a parallel structure making it efficient for various complex algorithms. The central processing unit (CPU) has usually been used for computing algorithms, but now also the GPU can be used for the same purpose. Exerting the GPU's strengths, like its highly parallel structure, can solve complex problems in a fraction of the time the same problem can be solved using the CPU. For example, the peak computational performance of a high-end dual core Pentium IV processor is 25.6 GFLOPS[2], while the peak performance of a NVIDIA GeForce 7800 GTX (Already last generation.) is 313 GFLOPS [GGKM06].

The development of the GPU to simulate the physics of light for computer graphics, has resulted in the discovery that the GPU also can be used for general purpose programming. General-Purpose computation on GPUs (GPGPU) is a field which has recently been addressed [GPG, OLG$^+$05, DHH05] and means solving equations for other purposes than rendering computer graphics. The GPUs arithmetic ability is well suited for applications like signal processing, which I am addressing, image processing, partial differential equations (PDEs), visualization and geometry.

This chapter introduces the ideas behind GPU programming and a couple of toolkits: GLSL [Ros06], CUDA [NVI07b], and RapidMind [Rap], that can be used when writing a GPU application. I will give a short overview of the different toolkits, while further details can be found in

[1]ATI refers to their GPU as the visual processing unit (VPU).

[2]FLOPS is an acronym for floating operations per second, while GFLOPS is short for gigaFLOPS.

the referenced material.

5.1 Development of the CPU versus the GPU

The CPU is built for high performance on sequential code, and have transistors dedicated for tasks like caching and branch prediction instead of only computational power. The GPU is optimized for parallel computing, and can with the same amount of transistors perform higher number of arithmetic operations. Graphics rendering uses a compute-intensive and highly parallel computation like what the GPU is specialized for. On a GPU more transistors are used for data processing instead of data caching. However, the data flow between the GPU and other units can be slow.

The evolution of the GPU has gone considerably faster than for the CPU the last couple of years, making the GPU a very powerful computational tool. While the floating-point operations for the CPU only has increased according to Moores law, doubling the number of transistors every second year [Gee05], the GPU has evolved more rapidly driven by the gaming industry's goal to make as realistic graphics as possible. Gamers throughout the world have requested this development, which has driven the creation of very efficient and cheap GPUs to produce high-end pictures. The new games have to be played with a powerful GPU, and the popularity of these devices has made the prices low compared to performance. As a result, a GPU is cheaper and more efficient than an equivalent CPU, when the problem has a parallel solution model. The average guy can use this to his advantage, and make efficient GPU applications on his off the shelf graphics card. Earlier, GPU programming was very limited and complicated, but now this way of thinking has expanded into easier programming with tools like NVIDIA's CUDA [NVI07b].

5.2 GPU programming

Various programming methods for general purpose programming on the GPU have developed the last years. In the beginning, GPU programming could only be done through assembly, thereby making it hard to develop a program. Being able to develop a GPU program through graphics APIs has made the process easier. I will start with explaining the graphics pipeline to briefly explain the idea behind programming on the GPU.

5.2.1 Graphics pipeline.

The graphics pipeline in Figure 5.1 illustrates the traditional work-flow on the GPU. Input data and execution follows a preset path. The output of each stage cannot be sent to the next stage until that stage has finished its computations. The slowest stage is called a bottleneck, which can stall the other parts of the pipeline, thus determining the speed of the program.

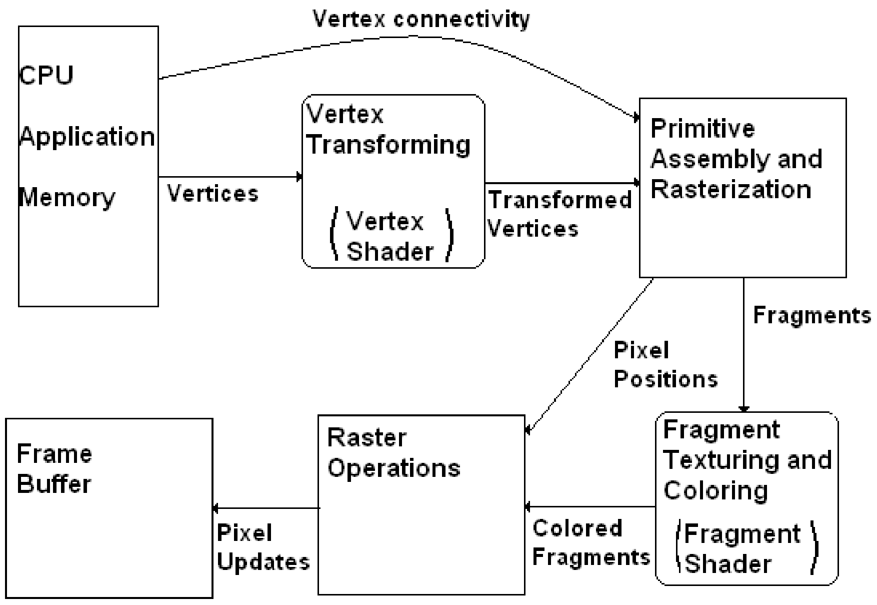


Figure 5.1: Simplified graphics pipeline.

The simplified pipeline in Figure 5.1 starts with an application stage. The application stage is purely software on the CPU giving the developer full control. This stage outputs the geometry of the points, lines or attributes the developer wants to render on the screen.

The vertex transformation stage sets the vertex attributes like location in space, color and texture coordinates amongst others. Vertex position transformation, per vertex lighting computations, generation and transformation of texture coordinates are some of the operations performed by the fixed functionality at this stage.

The inputs to primitive assembly and rasterization are the transformed vertex and connectivity information. The connectivity information tells the pipeline how the vertex connect to form each primitive[3]. This stage

[3]A primitive is a point, line, triangle, quad, etc.

is also responsible for clipping primitives against the view frustum[4]. The rasterization stage determines the fragments[5] and pixel positions of the primitives. A fragment defines the data that will be used to update a pixel at a specific location in the frame buffer. A fragment contains not only color, but also normals and texture coordinates that are used to find the pixel's color. This stage has two outputs. The position of the fragments in the frame-buffer and the interpolated values calculated in the vertex transformation stage.

Fragment texturing and coloring uses the interpolated fragment attributes as input. The color and texture coordinates were defined in previous stage and in the fragment texturing and coloring stage the color of the fragment can be combined with a texel[6]. If wanted, fog can be applied during this stage. Usually, the fragment texturing and coloring stage outputs a color value and depth for each fragment.

The raster operations receive the pixel locations, depth and color value of the fragments and then perform a series of tests on the fragments before they are written to the frame buffer. Some of the tests are the scissor test, the alpha test, the stencil test and the depth test. The pixel's value is updated with the fragment information according to the current blend mode if the fragment passes all the tests. Blending can only be done at this stage because only the raster operations have access to the frame buffer.

In newer graphic cards the vertex transformation stage can be replaced by vertex shaders, and the fixed fragment texturing and coloring stage can be replaced by fragment shaders. Both stages are then programmable and can be used for GPGPU programming. The vertex shader operates on the vertex, letting the developer do overall adjustments to the data. The fragment shader can define operations on each fragment. Most operations to be performed when performing on a general purpose calculation are defined in the fragment shader.

Textures

A texture can be seen as a picture. This picture can either be displayed as a normal square picture like in a picture frame, or it can be displayed by for example wrapping it around a ball. When wrapping the square picture around the round ball, the picture can look stretched out in some places, and compressed in others. Each texel is placed on the ball according to

[4]Removing everything outside the box defining what is visible to the viewer.

[5]A fragment is the name of the pixel before it is written to a frame buffer.

[6]Texture element.

the texture coordinates, making it fit perfectly. Realistic graphics can be produced by rendering a texture onto a surface. Textures are stored on the GPU reducing the cost of memory reads. With GLSL [Ros06], a texture can be processed with something called a shader.

Shaders

A shader is a program which can be run on the GPU. It makes it easier to create exactly what you like, or even calculate algorithms. Graphics of moving water can for example be calculated directly with algorithms on the GPU instead of generating a series of textures. One catch with using shaders is that when you write the code it looks like C++, but still it is restricted to certain operations. Another problem is that there are few well working debugging programs. In some cases when you write something wrong you will get an error, but you are not told where it is. Sometimes not even a warning is displayed, but you fail to get a proper result. As an example I can mention one thing I experienced: I wanted to have a for-loop in the shader. I did not get any error messages and the program seemed to run as it should, but the result was wrong. After a while I figured out that on my computer a for-loop in a shader had the maximum length 256, but if the for-loop was longer the GPU just exited the for-loop after 256 steps and continued the rest of the computations in the shader as if nothing had happened.

5.2.2 Before writing a program.

When programming on a GPU, a couple of things need to be kept in mind. First of all, the computations are performed in parallel and therefore extra caution should be exercised. That the computations are performed in parallel means that the current dataset is computed in a fashion where many values are computed at the same time. You should use at least two buffers while computing. One read only buffer holding the current dataset, and another buffer to store the computed values. Reading and writing using the same buffer can cause artifacts, because of data-dependencies between parallel operations.

Another thing is that the data should be represented as floating point values. The GPU can not represent integer values, and will in those cases use an approximated floating point value. Up until now, only single precision values have been supported, but even though this most likely will change within 2007 [NVI07a] along with the support for integers, some restrictions to the accuracy of the computations should be expected.

To write an efficient GPU program, detailed knowledge of how the GPU works is essential. More information about efficient GPU programming can be found in [Fer04] and [PF05].

The OpenGL shading language(GLSL) [Ros06] in Subsection 5.2.3 was developed as an extension to the OpenGL API [OSW$^+$05], which is used for graphics. The OpenGL shading language allows the user to write programmable shaders to more easily perform complex computations. Other GPU languages similar to GLSL that should be mentioned are the High Level Shading Language (HLSL) [SL05] and C for graphics (Cg) [FK03].

Developers familiar with OpenGL can quickly write a GPU program with GLSL. For others, the threshold for learning to use OpenGL was still quite large compared to developing a C++ application. Other programming models, which came after GLSL, provided programming more similar to C++. Development platforms like RapidMind, Section 5.2.5, and PeakStream (not discussed here) [Pea], lets the programmer write C++ code and then the platform generates GPU-specific code like GLSL or other multiprocessor languages.

Very recently, two new toolkits have been developed by ATI and NVIDIA. They have two very different approaches, but both toolkits can be used to directly write General-Purpose computation on GPUs (GPGPU). ATI developed an assembly-like programming language called Close To the Metal (CTM) [ATI06], which exposes the GPU hardware, letting you implement whatever you want, but the code can be very tedious to write. The Compute Unified Device Architecture (CUDA), further described in Section 5.2.4, from NVIDIA is a higher-level language similar to C++, and can therefore be easier to write. On the other hand, expertly written assembly will probably prove more efficient.

5.2.3 OpenGL Shading Language

OpenGL [OSW$^+$05] is one of the main languages used for graphic programming. First of all used as a tool to program the graphics to be displayed on the screen. An OpenGL program can be written in a C++ like programming language, where calls provided by the OpenGL package give the proper graphics commands. Other programming languages like Java and Python also have support for OpenGL. The OpenGL API was expanded with the OpenGL Shading Language (GLSL) [Ros06], which lets you do some of your computations directly on the GPU. This programmability has given more flexibility in what you can calculate on a GPU. With traditional OpenGL the developer had to follow the fixed pipeline, only giving calls in the application stage. With GLSL the developer is less restricted and is allowed to follow a partly fixed pipeline. The data follows the same pipeline, but the shaders makes it possible to program the vertex transform and fragment processing stages.

5.2.4 CUDA

CUDA [NVI07b] is the latest GPU programming toolkit provided by NVIDIA. It was released for developers in November 2006, and then for the public in February 2007. CUDA is a very C like language, where you can decide what parts of the code you would like to compute on the CPU and what parts you would like to do on the GPU, by using the CPU as a host, and the GPU as a device. It is a new way of thinking when it comes to GPGPU programming, and is therefore still partly under construction. Its purpose is to make it easier to use the GPU as a device for general purpose computing.

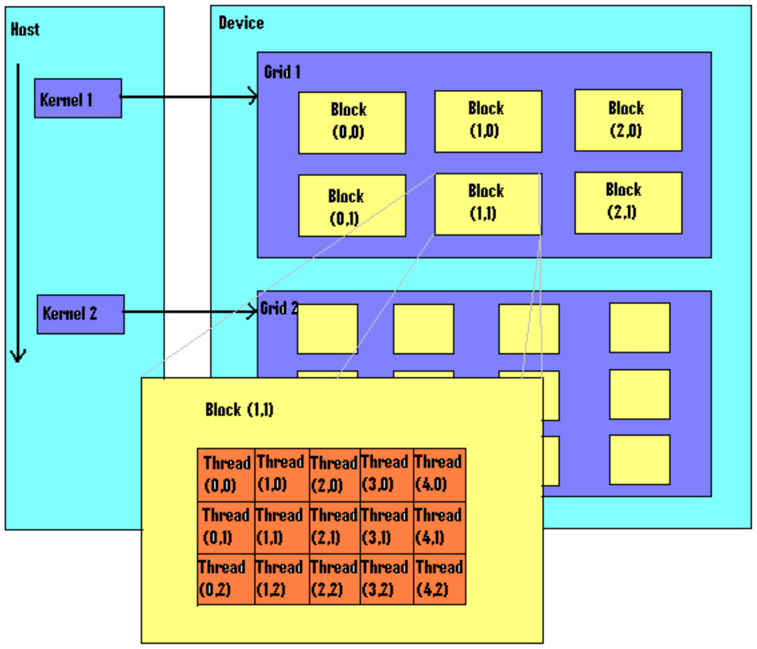


Figure 5.2: Organization in CUDA.

CUDA uses the GPU as a data-parallel computing device without mapping to graphics. Multitasking mechanisms in the operating system manage the access to the GPU by several CUDA and graphics applications running simultaneously. The CUDA API extends the C++ programming language making it easier for new developers.

CUDA can only be run on NVIDIA G80 cards and newer, but it is possible to run the program on computers with older graphic cards using Emulation mode. During Emulation mode the program will be processed on the CPU. The application will run quite slow when run on the CPU, but while debugging that is not a problem. It should be noted that you have to define that you want to run the application in Emulation mode, because if you do not, the application will fail when trying to run on something it expects to be a G80 card.

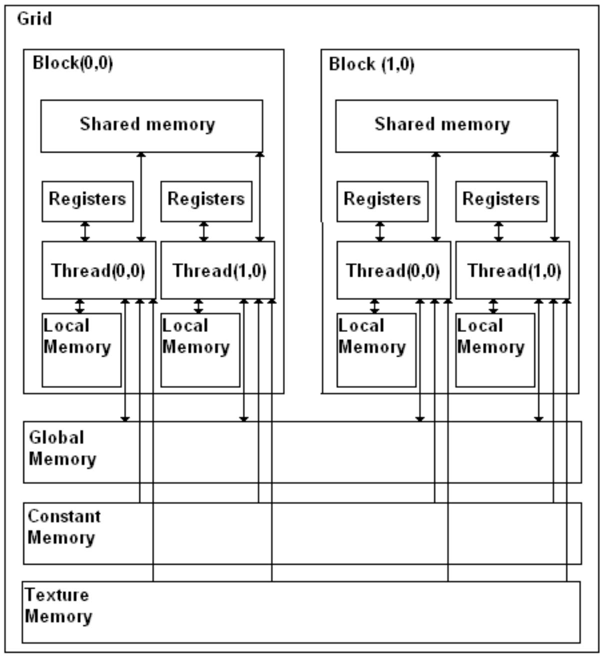


Figure 5.3: Cuda Memory Model.

The part of your program you would like to run on the GPU is written in a function called a kernel. A kernel roughly corresponds to a GLSL shader. Portions of data that can be computed independently are calculated simultaneously within the kernel. The part of the code you would like to run as a kernel is declared with CUDA keywords, but looks like any C code. A batch of threads is organized as a grid of thread blocks. Each grid is divided into blocks, and then threads within the blocks as illustrated in Figure 5.2. The amount of threads within one block is limited, but with a grid of blocks, this number can be larger. Letting more threads be run with the same kernel. The expense is less thread cooperation, because threads from different blocks can not communicate directly with each other. The data can be computed in parallel very efficiently using this structure on the GPU.

The structure of CUDA programs fits well with the new G80 GPUs from NVIDIA. The GPU is divided into multiprocessors where each multiprocessor is physically 8 ALUs[7] wide, and logically 16 ALUs wide working in the same manners as Intel's Hyper-Threading Technology[8]. The GPU contains 128 scalar ALUs as opposed to the last generation GPUs which uses fewer 4-vector units. This change is making it more efficient when working on scalar data, and easier to optimize.

Both device and host uses their own dynamic random access memory (DRAM) called device memory and host memory, respectively. Data can be copied between the two using optimized API calls that exploit the device's high-performance Direct Memory Access (DMA) engines. Threads share data with each other through parallel data cache or on-chip memory featured by CUDA with fast read and write access. Figure 5.3 describes the memory model for a grid. To get an efficient program the developer should minimize memory calls outside each block.

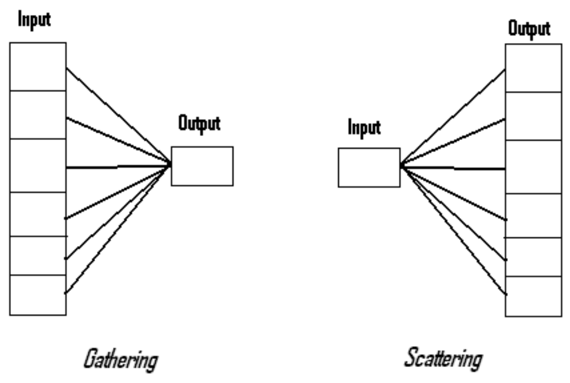


Figure 5.4: Gathering and scattering data.

When working on a dataset, two methods should be possible. Scattering and gathering. Figure 5.4 illustrates that with gathering, more than one data value is used to produce one output value. Or in other words,

[7]ALU is an arithmetic logic unit, and is the part of the processor performing the calculations.

[8]See Wikipedia for a good introduction on hyper-threading.

you collect data from different positions in the dataset to calculate one output that is written to current position. Addition of two points is just one example of gathering. Scattering means to write to one or more memory locations other than the one you are currently at. Some examples of scattering are quick-sort, hashing and histograms. Before CUDA, only gathering was possible on the GPU, but with CUDA also scattering is possible[9].

5.2.5 RapidMind

RapidMind is a software development platform for multi-core and stream processors like GPUs and the Cell Broadband Engine (Cell BE)[10]. It integrates with the existing C++ standard and therefore requires no new tools, compilers or preprocessors.

RapidMind differs from the other toolkits I have discussed with the fact that it is built on top of another language. That is done in order to maintain a known environment for the developer. As C++ is usually well known, implementing a program using RapidMind should not be too much trouble. RapidMind is still under evaluation by developers and not released to the public, but more information can be found at [Rap].

The programmer can write the code once and then RapidMind maps it to be run in parallel on any available computational resource supported by RapidMind.

RapidMind is presented as a library, but is used similar to a high-level programming language. Types and operations for data parallel programming are added to the existing C++ code using standard C++ features, making RapidMind easy to use within existing development projects.

The RapidMind platform can be used for both shaders and general purpose programming on the GPU. RapidMind enables the GPU to be used as a high-performance numerical co-processor when using general purpose mode. A shader is a special case of the general purpose mode and provides the full power of C++ for data abstraction.

What you do when developing a program in RapidMind, is that you write the usual C++, and then for the parts you would like to have computed on the GPU, you write a method in a manner that is recognized by RapidMind. These methods use RapidMind specific variables.

The developer can choose between three different backends: GLSL[11],

[9]Scattering was possible earlier, but only with complicated emulation by the use of vertex shaders.

[10]The Cell Broadband Engine supports nine processor cores on one chip, and is a specialized unit which for example is used in the Sony Playstation 3.

[11]When wanting to use the GPU.

Cell BE or CC[12]. If no specific backend is set, RapidMind will use the best backend available.

[12] CC can be used when you have multi-core CPU's.

Chapter 6

Implementation

So far, the thesis has given a survey of the wavelet transform, some GPU background and GPU programming languages. This chapter describes how I implemented the stationary wavelet transform with different GPU toolkits. I will give an overview of what I have done with some details. Experiences I gained and some difficulties I ran into along the way are also mentioned.

I chose to do an implementation of the stationary wavelet transform, not only because it was suggested, but also because I found wavelet transforms fascinating. For example, as I have mentioned earlier, wavelets and wavelet transforms are very useful when working on seismic data. Having a huge dataset also requires that the algorithm applied is efficient, so that you do not have to wait too long to get the result you want. On a GPU the data can be processed efficiently, giving you the result in near real-time. Of course, a cluster of CPU's could do the same, but when comparing one GPU with one CPU, the GPU has the advantage when it comes to efficiently processing the data in parallel.

In this part of my thesis I compare different programming toolkits for the GPU when applied to a specific problem. I decided to start with an implementation in C++ to have a reference dataset that could be used to verify that the calculations computed on the GPU were correct, and getting an overview of the problem at hand with a well known programming language. I thought an implementation using GLSL would be appropriate because that is one of the must common GPU languages. I implemented the algorithm in GLSL with the use of Shallows [cit], which is a library to make GPGPU programming with GLSL easier. Both CUDA and RapidMind are toolkits I heard about after I started on my thesis. The reasons to why I chose to do the implementation with those toolkits are that CUDA is a very promising toolkit introducing a new way of programming on the GPU, while RapidMind is intended to be a toolkit anyone with a C++ background should be able to use.

I will look at similarities and differences, and also how easy or difficult it is to implement SWT using the different toolkits. Then I will present a comparison of the efficiency of the implemented stationary wavelet transform and discuss how my way of thinking had to change when implementing with the different toolkits.

6.1 Implementation model

When implementing the stationary wavelet transform, two different approaches can be used. Either do the convolution directly between filters and signal or use the Fourier transform. Computing the convolution can be very expensive and does not collaborate well with a GPU application. A convolution involves many array multiplications and additions between the filter and the signal. The number of elements multiplied and added differs throughout the calculation, but for computing on a GPU all the passes and the changes in which elements to multiply and add are complicated and do not map well to the GPU hardware.

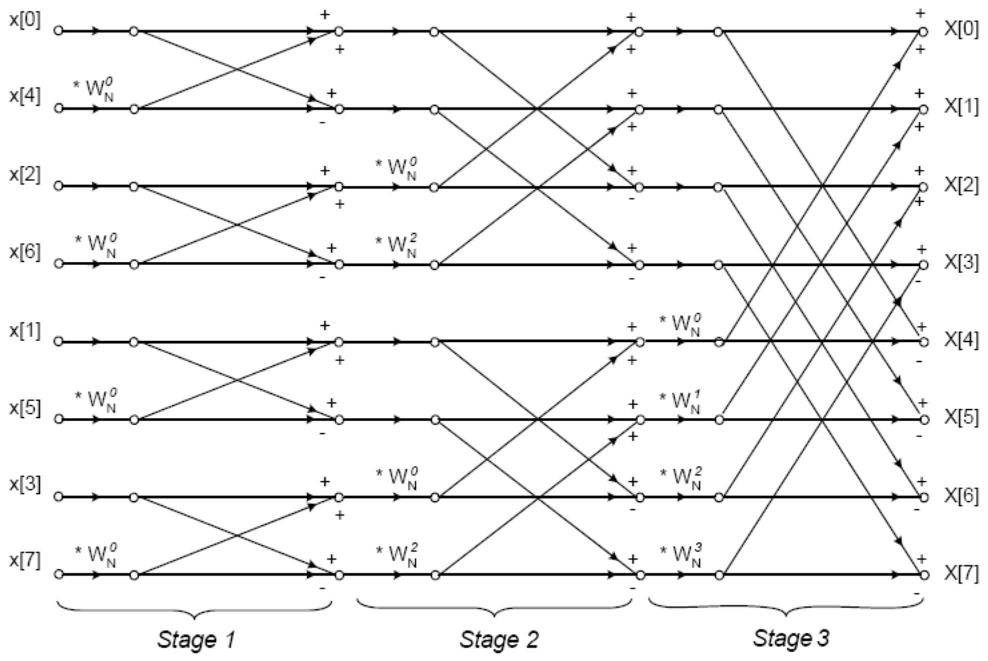


Figure 6.1: Butterfly for FFT.

```
calculate the two first filters
for all steps
  if not first step
     up-sample both filters with zeros

  compute Fourier transform of filters
  for all 1D signals
     if first step
        do Fourier transform of the signal

     multiply transformed filters and signal
     store result from the high-pass filtering in one array
     the low-pass filtering result is used in the next step
  end
end
compute the inverse Fourier transform
```

Listing 6.1: SWT pseudo-code

The fast Fourier transform (FFT) is the Fourier transform most commonly implemented on computers. The butterfly figure in Figure 6.1 illustrates how the FFT is calculated for a signal with length 8. The variable $W_N^R = e^{-i(\frac{2\pi R}{N})}$ is called the twiddle factor. The number of passes with the FFT is $\log_2 N$[1], where N is the length of the signal. In each pass, every element is added with one other element which has been multiplied with a value. The number of passes makes it a better choice to calculate the Fourier transform of the filter and the signal, and then multiply those outputs when calculating on the GPU. An inverse Fourier transform will then produce the result of the stationary wavelet transform.

When using the Fourier transform, both filters and the signal has to be Fourier transformed. Then the transformed filter and signal are multiplied element-wise and the output inverse Fourier transformed to get the result of the first step of the SWT. Listing 6.1 describes the pseudo code for the stationary wavelet transform with Fourier transform, and applies to all my implementations. When wanting to perform more than one step, the inputs are Fourier transformed before the first multiplication. The output from the high-pass filtering is written to memory, while the output from the low-pass filtering is used as input signal in the next stage. The filters used in previous stage are up-sampled to create two filters with zeros between every element. The calculations can continue until the filters contain only zeros.

[1] $N = 2^{\log_2 N}$

```cpp
for (int i=0; i < SIGNALLENGTH; ++i, ++tmp, ++dftSignal,
      ++dftFilter) {
 tmp->Re = (dftSignal->Re * dftFilter->Re)
          - (dftSignal->Im * dftFilter->Im);
 tmp->Im = (dftSignal->Re * dftFilter->Im)
          + (dftSignal->Im * dftFilter->Re);
}
```

Listing 6.2: Complex multiplication with C++

6.2 Implementation using C++

I implemented both methods in C++. First I implemented with convolution to get a result which I could use to verify the computations in the other implementations. The convolution method is the easiest to implement. Then I implemented with Fourier transform to get an application I could use as a base when implementing with the GPU toolkits.

My C++ implementation with convolution is pretty much straight forward. The signal is directly convolved with the filter, and I could then easily verify if the result was correct. Since this is the only implementation where I performed a convolution, I could not base my other implementations directly on the code. The code I used for the convolution is presented in Appendix A.

Implementing the stationary wavelet transform with C++, but with the Fourier transform was a bit more tricky. The signal and the filter is multiplied after being Fourier transformed, and then inverse Fourier transformed to finish the stationary wavelet transform. The Fourier transform produces a complex output, therefore complex multiplication. The code for the complex multiplication with C++ is written in Listing 6.2.

6.3 Implementation using GLSL

The most difficult part when implementing the SWT with GLSL was to implement the Fourier transform. I implemented the fast Fourier transform (FFT), which is said to be efficient when computed on the GPU. The general FFT implementation uses bit-shifting, which can not be calculated on the GPU. Instead I made a lookup table containing the bit-shift on the CPU and then used that table when calculating the transform. This operation also required that I used one shader for the first calculation in the butterfly, and then I could use another shader for the rest. The inverse

```
 // -*-C++-*-

[Vertex shader]

void main ()
{
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   gl_TexCoord[0]=gl_MultiTexCoord0;
}

[Fragment shader]
uniform sampler2D trace;
uniform sampler2D filter;

//Find the multiplication of a signal and a filter
//For given frequency-domain input data I and input filter data F,
//the output data D can be written:
//
// D(f) = F(f)*I(f);
//

void main ()
{
 vec4 traceFreq = texture2D(trace, vec2(gl_TexCoord[0].xy));
 vec4 filterFreq = texture2D(filter, vec2(gl_TexCoord[0].xy));
 vec4 result;
 result.x = (filterFreq.x*traceFreq.x)
             - (filterFreq.y*traceFreq.y);
 result.y = (filterFreq.x*traceFreq.y)
             + (filterFreq.y*traceFreq.x);
 gl_FragColor = result;
}
```

Listing 6.3: The shader for complex multiplication with GLSL

```
//Create a texture of the signal array
traceTex.reset(new Texture2D(LUMINANCE32F_ARB, SIGNALLENGTH,
                NUM_TRACES, GL_LUMINANCE, GL_FLOAT, signal, false));

//Initiate complexMult shader
complexMultProg.reset(new GLProgram);
complexMultProg->useNormalizedTexCoords();
complexMultProg->readFile("../convolution/complexMult.shader");
complexMultProg->setFrameBuffer(fb);
complexMultProg->setInputTexture("trace", rtTrace->getTexture())
complexMultProg->setOutputTarget(0, rt[in]);
complexMultProg->run();
```

Listing 6.4: Calls to initiate the shader for complex multiplication with GLSL

Fourier transform also required one shader for the first computation, and could then use the second FFT shader for the rest.

After implementing the FFT, I had to make a shader which multiplied two complex textures. The data in the texture stores the real value in the red-position and the imaginary value in the green-position, making it easy to access the data.

Listing 6.3 contains the shader code used to multiply the signal with the filter. In the implementation the shader code is stored in a separate file titled `complexMult.shader` and loaded within the application before it can be run. The filter and the signal is loaded into textures before being sent to the shader, so that the shader can calculate with data it understands.

The input data to the shader can be stored in a texture. The data is then first saved in an array, and then the call `traceTex.reset(new Texture2D(....));` creates a texture of the signal array. After the shader is initialized by the code in Listing 6.4, the call `setInputTarget` specifies where the shader can find the texture. An output target needs to be set with `setOutputTarget` and finally the shader is run with `complexMultProg->run();`.

The calls in Listing 6.4 are Shallows specific, and conceals a lot of what is actually going on. For example `run` hides that the program runs by rendering to a framebuffer.

One challenge when implementing with GLSL was to keep track of which buffer to use. I used a ping-pong technique when calculating the FFT. The ping-pong technique uses two buffers. The first buffer is used to read from and the second to write to. In the next calculation their read-write properties are switched. Using the same two buffers is efficient, because creating and deleting buffers can be computationally expensive.

```
// Allocate device memory for signal
Complex* d_signal;
CUDA_SAFE_CALL(cudaMalloc((void**)&d_signal, mem_size));
// Copy host memory to device
CUDA_SAFE_CALL(cudaMemcpy(d_signal, h_padded_signal, mem_size,
cudaMemcpyHostToDevice));
```

Listing 6.5: Copy data from host memory to device memory in CUDA.

```
//CUFFT plan
cufftHandle plan;
CUFFT_SAFE_CALL(cufftPlan1d(&plan, new_size, CUFFT_DATA_C2C,
                    ROW_SIZE));

//Transform the signal
CUFFT_SAFE_CALL(cufftExecute(plan, d_signal, d_signal,
                    CUFFT_FORWARD));
```

Listing 6.6: Use the FFT implemented in CUDA.

6.4 Implementation using CUDA

I based my CUDA implementation on one of the provided examples and rewrote it to fit the SWT algorithm. I chose to use the FFT algorithm provided with CUDA and then write the code to multiply the transformed outputs.

I created the signal and filters in arrays stored in host memory. For the CUDA program to interpret the arrays, the arrays are loaded from host memory into device memory with the commands in Listing 6.5. The call *CompMul<<<grid, threads>>>(d_signal, d_filter_kernel, 1.0f/SIGNAL_SIZE);* runs the CUDA environment code.

The calls in Listing 6.6 shows how to use the provided FFT implementation.

The complex multiplication which is run on the device is written in the kernel in Listing 6.7. The parameters of the current thread and block are used to calculate the current position in the dataset.

After all calculations are finished, the data is transferred back to host memory as in Listing 6.8 and can be output to the screen.

The challenge with CUDA was to understand how to divide the data into blocks, and then how to address them properly. You have to choose the size of the blocks, and try to maximize the utilization of the available computing resources. For example, the number of blocks should be at

```
// Complex multiplication
static __global__ void CompMul(Complex* a,
                               const Complex* b, float scale)
{
 int tidx = threadIdx.x;
 int tidy = threadIdx.y;
 int bidx = blockIdx.x;
 int bidy = blockIdx.y;
 int currentPos = (tidx + bidx*BLOCK_SIZE_X) +
                  (tidy + bidy*BLOCK_SIZE_Y)*SIGNAL_SIZE;
 Complex c;
 c.x = a[currentPos].x * b[tidx+ bidx*BLOCK_SIZE_X].x -
       a[currentPos].y * b[tidx+ bidx*BLOCK_SIZE_X].y;
 c.y = a[currentPos].x * b[tidx+ bidx*BLOCK_SIZE_X].y +
       a[currentPos].y * b[tidx+ bidx*BLOCK_SIZE_X].x;
 a[currentPos].x = scale*c.x;
 a[currentPos].y = scale*c.y;
}
```

Listing 6.7: Multiplication with CUDA

```
// Copy device memory to host
Complex* h_convolved_signal = (Complex*) malloc( mem_size);
CUDA_SAFE_CALL(cudaMemcpy(h_convolved_signal, d_signal, mem_size,
cudaMemcpyDeviceToHost));
```

Listing 6.8: Transfer the data back to host memory from device memory.

```
// Program for Complex Multiplication
Program ComplexMult = RM_BEGIN {
 In<Value2f> aInput;   // first input
 In<Value2f> bInput;   // second input
 Out<Value2f> output; // output

 output[0] = ((aInput[0] * bInput[0]) - (aInput[1] * bInput[1]));
 output[1] = ((aInput[0] * bInput[1]) + (aInput[1] * bInput[0]));
} RM_END;
```

Listing 6.9: Multiplication with RapidMind

least the same, but preferably more than there are multiprocessors on the device so that no multiprocessor will be left idle. The multiprocessors have a Single Instruction, Multiple Data architecture (SIMD)[2]. A grid of blocks is executed on the GPU with one or more blocks on each multiprocessor using time slicing. Each block is divided into SIMD groups containing the same number of threads, called warps. To have very fast memory access, a block is processed by only one multiprocessor keeping the shared memory space in the on-chip memory.

6.5 Implementation using RapidMind

The first look I had at some RapidMind examples, led me to believe that it should be easy to implement. It looked very much like C++ code, but with some RapidMind specific variables and calls. I decided to start with my C++ implementation, and then rewrite the parts where I could use RapidMind.

My first solution was therefore to do the FFT on the CPU, and then use RapidMind for the multiplication. The source code for the complex multiplication with RapidMind is presented in Listing 6.9. Before running the RapidMind specific code, the data to compute has to be in a RapidMind format. Then all there is to do is to run the RapidMind code with *rm_result = ComplexMult(rm_signal, rm_filter);*. To write the result on the screen, the data is converted back to C++ format with *const float* results = rm_result.read_data();*, which forces the completion and stores the result directly in the specified host memory.

I spent a lot of time trying to get the FFT working with RapidMind without having much success. I did find an implementation on their web

[2]Each processor executes the same instruction, but on different data.

page near the end of my thesis, which got me hoping that a working version of RapidMind would be possible to complete within the given timeframe. Unfortunately, using the FFT implementation proved to be a challenge. Looking at the source code for the FFT made me realize that implementing advanced algorithms with RapidMind was not quite as easy as I first expected. One of the challenges with this FFT implementation was that it used four values for each element in the array, effectively giving you room for two complex numbers in each element. This together with how the arrays in RapidMind work, and no real knowledge nor documentation of the inner workings of the FFT function stopped me from getting a working version in time. I did however manage to get it running, and my latest tests suggest that it is working and doing the amount of calculations it is supposed to. I still get wrong results from the calculation, but I believe that the problem is in the way that I input data. With this said, I have provided the timing results in the time tables in Section 6.6.

One of the features of RapidMind that got me confused was the way the RapidMind array works. Usually when working with arrays, in C++ or other languages, you can set a variable to be two-dimensional. With C++ you write something like "int foo[3][10]" to get an array with three rows and ten columns. In RapidMind it is done the opposite way, by using the first argument as width and the second as height. The reason for doing it like C++ is that one can easily access a whole row by using "foo[1]", giving you the second row. The data is organized sequential in the RAM, first by column, then by row. The element following the last element in a row is the first element on the next row. In RapidMind it seems that the arrays are organized in the same way, but you cannot access the data the same way. You can impose rules and set offsets and slices, as well as use something called stride and other functions to get the parts of the data you want. It is however very different from regular C++, and the fact that the syntax is so similar makes it easy to fall into old habits of C++ programming, resulting in compilations that fail.

Another issue I had with the RapidMind arrays, was that converting between a row from a two-dimensional array to an one-dimensional array produced all sorts of compilation errors. With the limited time I had, I was unable to figure out how to do that operation. The reason for the restrictions is of course that the underlying hardware cannot work on the data like a CPU can.

I figured that even though I was not able to implement the SWT as I wanted with RapidMind, I had already experienced a couple of useful things. One thing I had learned was that since you write in C++, the RapidMind specific code you write is transformed into shaders in runtime when you choose the GLSL backend. For example, letting the RapidMind specific code get a warm up run before actually calculating the complex multiplication, reduced the computational time from 600 to 60 milliseconds. That improvement shows the importance of doing a warm-up run when benchmarking. The warm-up run lets the program be compiled in memory and therefore run efficiently. The same can be done in other programming languages, but it was only with RapidMind the difference was significant.

6.6 Result

In this section, I will compare the run time of each implementation with different signal-lengths. I have timed the computations with CUDA both in emulation mode, and when running on a NVIDIA G80 card. The difference is just tremendous, and is related to the fact that the code is run in serial mode in emulation mode on the CPU, and in parallel when run properly on a G80 card.

I have changed the size of the dataset, both in signal length and how many signals there are in one dataset. I tested with one single input signal and then with 16, 64 and 512 to see how the applications behaved with more data to calculate.

Computing the stationary wavelet transform would demand the same amount of operations independently of the input. I have calculated with the same input signal multiple times in the cases with more than one input signal, because that made it easier to verify the result when testing the efficiency of the implementations. The length of the signal was specifically chosen to be power-of-two sizes to get a proper result with the GLSL and RapidMind's FFT implementations.

The computational efficiency of the RapidMind application seems to match what I was expecting, even though as I mentioned earlier the computations do not provide the correct results. Please keep in mind the notes I had regarding RapidMind in Section 6.5 when considering the following results. The calculation times are in the vicinity of the GLSL version, with a disadvantage on sizes except the largest datasets. RapidMind is more efficient at the larger datasets probably because of the fact that the RapidMind FFT code has doubled the memory efficiency of the GLSL code by using all four texture color values versus the GLSL version's two colors.

Tables 6.1 to 6.4 shows the difference in run-time with various sizes of the input signal. I let the application run 10 times before stopping the clock and calculated the average time to not risk getting only peak-time.

All applications are run on a computer with a NVIDIA GeForce 8800 GTX graphics card and an Intel Core 2 DUO E6600 CPU, and the timings are presented in milliseconds.

Signal length	64	256	1024	4096
C++ conv	0.012	0.040	0.154	0.596
C++ FFT	0.039	0.167	0.555	3.514
GLSL	8.850	10.652	10.805	13.106
CUDA G80	0.139	0.164	0.206	0.659
CUDA Emu	8.303	34.399	143.785	948.434
RapidMind	11.478	11.562	11.635	12.204

Table 6.1: Timing of the implementations with a single input signal. (milliseconds)

Signal length	64	256	1024	4096
C++ conv	0.159	0.615	2.374	9.564
C++ FFT	0.216	1.114	5.078	30.389
GLSL	7.808	9.678	11.280	17.047
CUDA G80	0.157	0.202	0.357	1.440
CUDA Emu	23.364	108.030	483.325	4374.110
RapidMind	23.362	23.959	24.753	27.074

Table 6.2: Timing of the implementations with 16 input signals. (milliseconds)

The run-time for the C++ application with convolution increases approximately linearly according to the amount of input data. That reflects that only the number of calculations influences the run-time. When running the FFT application in C++ the run-time increased more rapidly, but still linearly according to the amount of input signals.

The GLSL application represents a more complex run-time picture. With GLSL you have to add the cost of converting and transferring data to the GPU. Therefore a certain amount of data needs to be transferred and calculated before you can see an improvement. Compared to the C++ convolution you can not see improvement in run-time before the amount of data has reached 512x256, but above that size the speed of the GLSL implementation is up to twice as fast. Compared to the FFT implementation with C++, the GLSL application was more efficient for sizes from 256x256, which is half the size that had to be calculated compared to the convolution.

Signal length	64	256	1024	4096
C++ conv	0.609	2.605	9.727	40.062
C++ FFT	0.784	4.411	19.8	114.493
GLSL	8.569	10.193	14.585	28.953
CUDA G80	0.203	0.368	0.847	3.853
CUDA Emu	70.812	338.252	1590.721	15316.650
RapidMind	34.924	37.061	38.399	49.424

Table 6.3: Timing of the implementations with 64 input signals. (milliseconds)

Signal length	64	256	1024	2048	4096
C++ conv	4.892	19.778	80.184	159.053	318.488
C++ FFT	6.358	36.961	171.931	922.459	2431.53
GLSL	11.350	16.977	43.976	80.249	170.813
CUDA G80 1x1	0.817	2.329	8.663	19.550	39.468
CUDA G80	0.568	1.334	4.960	12.015	24.407
CUDA Emu	511.397	2527.385	11981.309	54429.312	116792.561
RapidMind	58.875	60.635	80.509	108.347	160.284

Table 6.4: Timing of the implementations with 512 input signals. (milliseconds)

Running SWT with CUDA on the G80 card resulted in generally lower timings. As with GLSL you have to take into account that the data needs to be transferred to the GPU before calculating. Only when calculating a single input signal the convolution on C++ was faster.

Compared to GLSL, CUDA on a G80 is always significantly more efficient, while compared to FFT with C++ slower only with a single signal of length 64.

CUDA Emulation mode is presented more as a curiosity. With the Emulation mode, the exact same code is used and run on the same computer, but still it could be up to many thousand times slower than running it as a proper CUDA application on the GPU. It is clearly meant as a debugging tool.

With CUDA I also tried with different block sizes, and experienced that in emulation mode, running with block-size one, was up to twice as fast as running with more threads and more blocks. That is most likely caused by it being run in serial mode, and having to work with many calculations simultaneously just slows down the process. Computing with CUDA on a G80 on the other hand needs many threads per block to utilize the GPU properly.

Table 6.4 includes computational times for CUDA both when using blocks of size one, and blocks with 64 threads. All the other tables uses blocks with 64 threads, to get the best timings. When only having one thread per block, the multiprocessors can not use their parallel power, and is forced to only do one computation at a time. The CUDA implementation is still quite efficient, because it can make use of that the different multiprocessors do not have to idle.

I had expected CUDA to be more efficient than GLSL, but I did not expect the speed-up I got. The reason why there should be a speed-up is that CUDA is written for the hardware on the G80, but I do also believe that the CUDA implementation of the FFT is more optimized for the GPU than my FFT implementation with GLSL.

The computational time of my RapidMind implementation was close to the timings for the GLSL implementation. Mostly, RapidMind was slower, which to some extent could be caused by the fact that RapidMind generates the shader code. A shader code written directly in GLSL could give a more efficient result. As mentioned, RapidMind's improved memory efficiency over my GLSL solution probably gave RapidMind an edge on large datasets. That RapidMind is still a beta version can also have an effect on the overall efficiency.

It is worth to point out that RapidMind generates the GLSL shaders on the fly, making the first round of computation slower than the following ones using the same code. This makes it unsuited for one-time calculations, and calculations that constantly changes the program code being computed. Also noteworthy is the fact that a real-life application should not implement warm-ups. The warm-up just computes all the functions with dummy data that are meant to run on the GPU anyway. It is better to just run it with the correct data, and save one calculation as opposed to do a warm-up.

I also want to note that I was unable to create a version that could multiply a single dimension array with all rows in a two dimensional array. This fact increased the times the FFT was run by about 50%, as I had to do the FFT on the same number of rows for the filter as for the signal, instead of calculating the FFT only once for the filter. This means that the efficiency of the RapidMind code is not optimal. Getting it to work properly would save a lot of computations. I will not be speculating too much in how much the improvement would be, other than saying that the FFT is the most computationally demanding part of the application, and that increasing the number of runs with 50% will have an meassurable impact on the performance, although not significantly enough to alter the timing tables noteworthy. The closest opponent to RapidMind performance vice seems to be the GLSL implementation, and in most places the computation time differs with more than 50%.

It should also be noted that calculating the FFT for the filters does not usually increase the times as much as 50%. Especially on small datasets where transferring data takes most of the time.

6.6.1 Summary of the implementations.

The different implementations all had different challenges. With C++ the challenge was to figure out how to implement the stationary wavelet transform for the first time. Knowing that the SWT can be solved with the fast Fourier transform meant learning the FFT in detail. That implied using time on something I then would know for the next implementation. On the other hand, since this is more about comparing the different GPU toolkits, the time I used to implement with C++ is not entirely relevant, but the C++ implementation made it easier to implement with the GPU toolkits.

Table 6.5 implies my opinion of my experience when implementing the various toolkits. All three toolkits I tried were more efficient than the CPU when calculating the stationary wavelet transform. At least when computing enough data to make use of the GPUs parallel computational power.

First time implementing something with GLSL can be very troublesome if you do not have experience with OpenGL. CUDA is easy to start with if you begin by rewriting some of the sample code. It has a lot of similarities with C++, and should therefore be fairly familiar. RapidMind is easy if you want to compute something very simple. Like adding two arrays. When facing more complicated calculations, the threshold to implement is high. The implementation time varied for the different toolkits. GLSL took a long time to get a correct implementation. Implementing with CUDA went quite fast when rewriting the code, but to get a understanding of how to divide the data into threads and blocks required some time. That understanding was not required to get a correct result, but only to make the computations more efficient. With RapidMind it was difficult to figure out exactly how to work with the data. It seemed easy at first, but required a lot of insight before I knew how to use the RapidMind variables and calls correctly. The three toolkits I implemented all had documentation, but GLSL also has books to easier understand the ideas. Both GLSL and CUDA can be discussed in on-line forums, while RapidMind still only provides support by e-mailing to the developers.

	GLSL	CUDA	RapidMind
Efficiency vs. CPU	Good	Very good	Good
Beginner threshold	High	Low	Medium
Implementation time	Long	Short	Medium
Documentation	Good	Good	Poor

Table 6.5: Table of GPU toolkit comparison.

Implementing with GLSL required some creative ideas, but as I have done some implementations with GLSL before, my experience probably helped me implement faster than if it had been my first GLSL implementation. With GLSL you have to know when to use the different buffers, and you have to control most things yourself. It can therefore be complicated and take time to implement.

CUDA had provided many examples which helped me understand the idea behind implementing code with CUDA. Also the fact that I did not have to implement the FFT reduced the implementation time considerably. I found CUDA easiest to understand.

RapidMind, which I expected to be easiest to apply, proved to be the most difficult. The reasons for that can be hard to find, but I think that the most significant reason is that it was hard to figure out how to implement what I wanted, giving RapidMind a high beginner threshold. An advantage with RapidMind is that the same application can be used on various platforms. That implies that you only have to implement once when wanting to test on the different backends supported by RapidMind.

What I am saying is that it is hard to say exactly how much time I used implementing with the different toolkits, since I faced different difficulties and learned more about how to implement the SWT after completing each implementation. I do believe that GLSL can be the most complicated to implement even though you have some experience. RapidMind is probably a lot easier to implement once you have learned how to use it and get proper documentation, but until then RapidMind can be tedious to work with.

Chapter 7

Conclusion and further work

Wavelets and wavelet transforms is a field with a lot of recent development making it difficult to sort out the most useful information. You can use wavelets to approximate a signal, and a wavelet transform exploits that ability. It is the user's responsibility to choose the wavelet and wavelet transform that best suit the problem.

The properties of the wavelet transform can be used in many fields, for example on very large datasets that can be computational expensive when the computing resources are limited. Using the GPU as a tool to calculate parallel problems can improve the computational power of your computer substantially.

Programming on the GPU is still in the starting phase. A lot of companies have opened their eyes for it, seeing how efficiently a GPU can compute a parallel problem compared to a CPU. Many toolkits have recently been developed and which toolkit to use depends on your intension. Some toolkits demand that you learn a new programming language. These languages are closer to the GPUs way of processing, and could prove very efficient under many circumstances. On the other hand, the toolkits that resemble C++, which you might already know, will probably reduce the time it takes to develop the application, and the result will in some cases be just as quick as the GPU specific languages. It all comes down to what you need, and as history has shown, someone will always try to squeeze out the last bit of potential performance where it really matters, whilst others just want a quick and easy increase in speed.

The OpenGL extension GLSL can be complicated because you have to control much of the work-flow yourself. A shader program has a shortage of some things that can easily be done with C++. When working around those problems, you have to be creative and should know about how the GPU works to create an efficient result. Much of the required skill and knowledge comes from experience, as GPU programming still is a very new technology with limited documentation and Internet resources.

CUDA presents a new way of structuring the implementation, but it is quite easy to get a result which works as expected since it is similar to C++. The most significant difference is how you have to structure the chunks of data you want to calculate.

RapidMind is presented as I toolkit anyone with a C++ experience should be able to understand. My problem was that I found it difficult to figure out how RapidMind wanted to have the data represented when implementing a complex problem.

Unfortunately I was not capable of implementing everything I wanted. I was therefore unable to do much more than to get the different toolkits to work correctly. The CUDA and the RapidMind implementations are the first applications I have written with those toolkits, so my experience was very limited. I found CUDA to be easier to use probably because I had more than one example program to look at when figuring out how to implement the stationary wavelet transform. I thought that RapidMind was a lot more troublesome. Not only because I found their documentation harder to read, but also because I only had a couple of poorly documented examples to lean on.

It is difficult to find wanted information about RapidMind and CUDA since they both are pretty new. CUDA was presented publicly in November and released in January, while RapidMind is still in evaluation phase. CUDA has an online forum which you can use when running into problems. With RapidMind you currently have to send an email, which makes the development slower.

The main advantage of using toolkits must be the abstraction that removes the need for OpenGL specific commands. The OpenGL commands have been a major source of errors and without proper error messages, also very difficult to locate and solve. Both RapidMind and CUDA have good error reporting systems, which simplifies the debugging process.

7.1 Further work

I learned the hard way that implementing on the GPU can take a lot more time then expected. A problem that might seem simple is not as straightforward to do when there are calculations you cannot implement directly. My implementations therefore did not get as far as I wanted. For example, I only implemented a single pass of the stationary wavelet transform. It should not be very difficult to expand the implementations to do the full transform, but I had to make some choices and thought that it was more important to get the implementations to work correctly with the different toolkits. Still my implementation of the stationary wavelet transform with RapidMind does not output a correct result.

Another thing I was thinking of implementing, was the FFT with CUDA. Of course that would not be necessary because NVIDIA has already provided an implementation for CUDA, but having my own implementation would make the comparison more reliable since I have my own in the C++ and GLSL implementations. Implementing my own FFT with RapidMind would also be interesting.

Finally it would have been nice to further improve the implementations. Trying out different ideas is an essential part of getting an efficient program, so continuing with experimenting would be useful. Getting the implementations to be even more efficient would therefore be something that should be explored.

Appendix A

Convolution in C++

```
void convolution(float *signal_in, int signalLength,
                 float *lowFilter, float *highFilter,
                 int filterLength, float *signal_outlow,
                 float *signal_outhigh)
{
 int iSignal, jFilter;
 float signalLow, signalHigh;

 for (iSignal=signalLength;
      iSignal < signalLength+filterLength-1; ++iSignal){
  signal_in[iSignal] = signal_in[iSignal-signalLength];
 }
 for (iSignal=0; iSignal<signalLength; ++iSignal){
  signalLow = 0;
  signalHigh = 0;
  for (jFilter=0; jFilter<filterLength; ++jFilter){
   signalLow = signalLow +
   signal_in[jFilter+iSignal]*lowFilter[filterLength-1-jFilter];

   signalHigh = signalHigh +
   signal_in[jFilter+iSignal]*highFilter[filterLength-1-jFilter];
  }
  signal_outlow[iSignal] = signalLow;
  signal_outhigh[iSignal] = signalHigh;
 }
}
```

Bibliography

[Add02] P. S. Addison. *The Illustrated Wavelet Transform Handbook*. Taylor & Francis, July 2002.

[AGF93] P. Abry, P. Gonçalvès, and P. Flandrin. Wavelet-based spectral analysis of $1/f$ processes. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 237–240, 1993. Minneapolis, MN, USA.

[ATI06] ATI. Ati ctm guide, 2006. Available at http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf.

[BGG98] C. S. Burrus, R. A. Gopinath, and H. Guo. *Introduction to wavelets and wavelet transforms: a primer*. Prentice-Hall, pub-PH:adr, 1998. With additional material and programs by Jan E. Odegard and Ivan W. Selesnick.

[BW98] K. Berkner and R. Wells. Smoothness estimates for soft-threshold denoising via translation invariant wavelet transforms, 1998.

[Cas01] J. P. Castagna. Recent advances in seismic lithologic analysis. *Geophysics*, 66(1):42–46, 2001.

[cit] Shallows homepage. More information available at http://shallows.sourceforge.net/.

[CO95] A. Chakraborty and D. Okaya. Frequency-time decomposition of seismic data using wavelet-based methods. *Geophysics*, 60(6):1906–1916, 1995.

[CW06] S. Cui and Y. Wang. Redundant wavelet transform in video signal processing. In *IPCV*, pages 191–196, 2006.

[DHH05] T. Dokken, T. R. Hagen, and J. M. Hjelmervik. The gpu as a high performance computational resource. In *SCCG '05: Proceedings of the 21st spring conference on Computer graphics*, pages 21–26, New York, NY, USA, 2005. ACM Press.

[Fer04] R. Fernando. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley Professional, March 2004.

[FK03] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[FKK02] S. E. Ferrando, L. A. Kolasa, and N. Kova&#269;evi&#263;. Algorithm 820: A flexible implementation of matching pursuit for gabor functions on the interval. *ACM Trans. Math. Softw.*, 28(3):337–353, 2002.

[Gee05] D. Geer. Taking the graphics processor beyond graphics. *Computer*, 38(9):14–16, 2005.

[GGKM06] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, New York, NY, USA, 2006. ACM Press.

[GGM84] P. Goupillaud, A. Grossmann, and J. Morlet. Cycle-Octave and related transforms in seismic signal analysis. *Geoexploration*, 23:85–102, 1984.

[GLOB95] H. Guo, M. Lang, J. E. Odegard, and C. S. Burrus. Nonlinear processing of a shift-invariant DWT for noise reduction and compression. In *Proceedings of the International Conference on Digital Signal Processing*, pages 332–337, Limassol, Cyprus, 26–28 1995.

[GPG] GPGPU. gpgpu.org. Available at http://www.gpgpu.org.

[Gra95] A. Graps. An introduction to wavelets. *IEEE Computational Sciences and Engineering*, 2(2):50–61, 1995.

[HBB92] F. Hlawatsch and G. F. Boudreaux-Bartels. Linear and quadratic time-frequency signals representations. *IEEE Signal Processing Magazine*, ?:21–67, April 1992.

[HRMS04] G. Hernandez, B. Reusch, M. Mendoza, and L. Salinas. Shiftability and filter bank design using morlet wavelet. In *QEST '04: Proceedings of the The Quantitative Evaluation of Systems, First International Conference on (QEST'04)*, pages 141–148, Washington, DC, USA, 2004. IEEE Computer Society.

[I. 92] I. Daubechies. *Ten Lectures on Wavelets*. SIAM Publications, 1992.

[JG82] J. Morlet, G. Arens, I. Fourgeau and D. Giard. Wave propagation and sampling theory. *Geophysics*, 47:203–236, 1982.

[JPCS03] S. Sun J. P. Castagna and R. W. Siegfried. Instantaneous spectral analysis: Detection of low-frequency shadows associated with hydrocarbons. *The Leading Edge*, 22(2):120–127, 2003.

[Kai94] G. Kaiser. *A Friendly Guide to Wavelets*. Birkhauser, August 1994.

[KFG97] P. Kumar and E. Foufoula-Georgiou. Wavelet analysis for geophysical applications. *Review of Geophysics*, 35(4):385–412, 1997.

[LGO$^+$96] M. Lang, H. Guo, J. E. Odegard, C. S. Burrus, and R. O. Wells. Noise reduction using an undecimated discrete wavelet transform. *IEEE Signal Processing Letters*, 3(1), 1996.

[LM05] J. Liu and K. J. Marfurt. Matching pursuit decomposition using morlet wavelets. *SEG Technical Program Expanded Abstracts*, 24(1):786–789, 2005.

[LZC$^+$06] H. Li, W. Zhao, H. Cao, F. Yao, and L. Shao. Measures of scale based on the wavelet scalogram with applications to seismic attenuation. *Geophysics*, 71(5):V111–V118, 2006.

[Mal99] S. Mallat. *A Wavelet Tour of Signal Processing, Second Edition (Wavelet Analysis & Its Applications)*. Academic Press, September 1999.

[MZ93] S. Mallat and Z. Zhang. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing*, 41(12):3397–3415, 1993.

[NS] G. P. Nason and B. W. Silverman. The stationary wavelet transform and some statistical applications. pages 281–300.

[NVI07a] NVIDIA. Cuda release notes version 0.8, 2007. Available at http://developer.download.nvidia.com/compute/cuda/0_8/NVIDIA_CUDA_SDK_releasenotes_readme_win32_linux.zip.

[NVI07b] NVIDIA. Nvidia cuda guide, 2007. More information available at http://developer.nvidia.com/object/cuda.html.

[OLG+05] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[OSW+05] Opengl, D. Shreiner, M. Woo, J. Neider, and T.Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[Pea] PeakStream. Peakstream inc homepage. More information available at http://www.peakstreaminc.com/.

[PF05] M. Pharr and R. Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, March 2005.

[Rap] RapidMind. Rapidmind inc homepage. More information available at http://www.rapidmind.net/.

[Ros06] R. J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, January 2006.

[She92] M. Shensa. The Discrete Wavelet Transform: Wedding the À Trous and Mallat Algorithms. In *IEEE Transactions on Signal Processing*, volume 40, pages 2464–2482, 1992.

[SL05] S. St-Laurent. *The COMPLETE Effect and HLSL Guide*. Paradoxal Press, 2005.

[SRAC05] S. Sinha, P. S. Routh, P. D. Anno, and J. P. Castagna. Spectral decomposition of seismic data with continuous-wavelet transform. *Geophysics*, 70(6):P19–P25, 2005.

[Wil02] S.S Wilson. Using a pseudo-random binary sequence as a mother wavelet in thewavelet-correlation system identification method. *SoutheastCon, 2002. Proceedings IEEE*, pages 58–61, 2002.

[ZLBN96] R. Zaciu, C. Lamba, C. Burlacu, and G. Nicula. Motion estimation and motion compensation using an overcomplete discrete wavelet transform. In *International Conference on Image Processing*, pages I: 973–976, 1996.