NTNU
Innovation and Creativity

# Real-Time Wavelet Filtering on the GPU

**Erik Axel Rønnevig Nielsen**

Master of Science in Computer Science
Submission date: May 2007
Supervisor: Anne Cathrine Elster, IDI

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

Problem Description

Specialiced graphics hardware, GPUs, has in recent years had a rapid increase in both performance and programmability. This has made it interesting as a platform for general purpose computations.

Ultrasound imaging has for many years been one of the most popular medical diagnostic tools. Compared to X-rays, computed tomography(CT) and magnetic resonance imaging (MRI), ultrasound has the advantages of safety, low cost and interactivity. With the recent introduction of 3D ultrasound, the requirement for data processing has increased tremendously.

In this thesis, we want to study the intersection of these two fields. The objectives are to:
* Evaluate image enchancement techniques that are currently implemented on the CPU to see if it is possible to implement them on the GPU. An important subgoal is to conduct performance comparison of the two solutions.

* Evaluate image enchancement techniques that are currently not in use, but are made possible with the use of GPU technology.

If the first two objectives are met, other related tasks may be explored.

Assignment given: 15. January 2007
Supervisor: Anne Cathrine Elster, IDI

# Abstract

The wavelet transform is used for several applications including signal enhancement, compression (e.g. JPEG2000), and content analysis (e.g. FBI fingerprinting). Its popularity is due to fast access to high pass details at various levels of granularity.

In this thesis, we present a novel algorithm for computing the discrete wavelet transform using consumer-level graphics hardware (GPUs). Our motivation for looking at the wavelet transform is to speed up the image enhancement calculation used in ultrasound processing. Ultrasound imaging has for many years been one of the most popular medical diagnostic tools. However, with the recent introduction of 3D ultrasound, the combination of a huge increase in data and a real-time requirement have made fast image enchancement techniques very important.

Our new methods achieve a speedup of up to 30 compared to SIMD-optimised CPU-based implementations. It is also up to three times faster than earlier proposed GPU implementations. The speedup was made possible by analysing the underlying hardware and tailoring the algorithms to better fit the GPU than what has been done earlier. E.g. we avoid using lookup tables and dependent texture fetches that slowed down the earlier efforts. In addition, we use advanced GPU features like multiple render targets and texture source mirroring to minimise the number of texture fetches.

We also show that by using the GPU, it is possible to offload the CPU so that it reduces its load from 29% to 1%. This is especially interesting for cardiac ultrasound scanners since they have a real-time requirement of up to 50 fps. The wavelet method developed in this thesis is so successful that GE Healthcare is including it in their next generation of cardiac ultrasound scanners which will be released later this year.

With our proposed method, High-definition television (HDTV) denoising and other data intensive wavelet filtering applications, can be done in real-time.

# Acknowledgments

I would like to extend my gratitude to Dr. Anne C. Elster for being my primary supervisor during this thesis. Without her advice and high standards, this work would have ended as a collection of incoherent ideas. Gratitude is also extended to her Ph.D. student Thorvald Natvig, who provided great help during the formulation of the theoretical models in the thesis.

The help from Dr. Sevald Berg, Dr. Stein Inge Rabben, and especially my co-advisor Dr. Erik Steen, all from GE Healthcare, has been invaluable during this project.

Great gratitude goes to Jørgen Braseth and Elisabeth Dornish for having the patience to proofread the thesis and correct my many spelling errors. Also, I would like to thank the people at 'Ugle' computer lab for many coffee breaks and interesting discussions that made the thesis work worthwhile.

# Contents

# List of Tables

# List of Figures

Chapter 1

# Introduction

*'Begin at the beginning,' the King said, very gravely, 'and go on till you come to the end: then stop.'*
**-Lewis Carroll**

This chapter provides the motivation and the outline of this thesis. The intended audience of this thesis are professionals in the field of medical visualisation. Experience with computer visualisation is not required, but recommended for a thorough understanding of the thesis.

## 1.1 Motivation

When Röntgen first discovered x-rays in 1895, he triggered off a medical revolution. Today it is unimaginable to have a modern hospital without x-ray equipment like Computed Tomography (CT). Later, other methods like Magnetic Resonance Imaging (1967), and medical ultrasound (1956) has also been developed. A more recent breakthrough (1980s) is the use of computers to process and enhance the images to facilitate viewing. Image enhancement drastically improves the clinical value of the equipment and therefore it is ubiquitous today. As the physical imaging equipment has been improved over the years, more and more data needs to be processed. At the same time, the practitioners require better quality and real-time access to the data. Ultrasound has been an especially popular choice for real-time and portable imaging, mainly because of its non-invasive nature and small size.

Medical equipment used to be composed of special-purpose computer parts designed for the task. In later years, general-purpose hardware is increasingly replacing these parts. The rationale is twofold. First, it lets the manufacturer concentrate its efforts on the software. Secondly, the price of general purpose hardware is usually much lower than that of special purpose because of the mass production of the former. Actually, a large share of the hardware in a modern ultrasound scanner is the same as the hardware that resides in a computer under an office desk.

The ever-increasing amount of medical data, combined with the use of general purpose hardware, makes the task of fully utilising this hardware very important. The need for more computing power easily outgrows the hardware, and so the algorithms have to be tuned to a compromise between quality and speed. This problem is not new, and much effort has been put into investigating algorithms and methods for the x86, Intels dominating processor architecture. Such efforts are still important and will continue to be important in the future. However, there are other ways to increase the computational power of the computer system.

One component that has gone almost unnoticed so far is the graphics processor (GPU). The explanation for this oversight is simple Until recently, it couldn't be programmed at all. However, the gaming industry has driven the development forward, and today's GPUs are not just programmable, they are also much more powerful than the CPU in terms of raw computing power.

One important image-enchancement method used in medical imaging, and for ultrasound specifically, is wavelet filtering. Wavelets are a mathematical tool that makes it possible to decompose images and describe them in terms of a course approximation and several levels of detail. It is often said that by using wavelets it is possible to 'see the forest *and* the trees'.

In this thesis, we will investigate the possibility of using the GPU for wavelet image filtering in the context of medical ultrasound. The main goal is to offload the CPU so that it can be used for other tasks. Another and more ambitious goal is to make the enhancement faster than the CPU, and open up for possibilities of even more refined methods than the ones used today.

## 1.2 Thesis outline

This thesis has three main parts. The first part (Chapter one) contains background information and theory about ultrasound, GPU and wavelet filters. The second part (Chapter two) contains a description of the algorithms and methods developed during this thesis. The final part (Chapter four and five) holds the results and a conclusion. In addition to these three parts, an appendix includes practical information about the running the programs developed and listings of some of the important parts of our GPU code.

# Chapter 2

# Background

*A dwarf on a giant's shoulders sees farther of the two*
**-Didacus Stella**

In this chapter, we will present the main theories behind our work. In Section 2.1, we introduce the graphics processor (GPU), discuss the hardware features and how to program it. We will also explain why we choose to program the GPU. In Section 2.2, we present medical ultrasound, the context of this thesis. Section 2.3 discusses the theoretical background on wavelets. Section 2.4, the theory behind the filters we have implemented is discussed.

## 2.1 Graphics Processing Unit

The Graphics Processing Unit (GPU) is a special-purpose computing unit designed and optimised for computation related to 3D graphic rendering. In recent years, the GPU has been transformed from a static pipeline with modest capabilities to a very flexible and powerful computation unit. Owens et al. [1] is an excellent survey of the current state of General-Purpose computation on GPUs (GPGPU). For practical programming info, tips and tricks Gpu Gems [2] is highly recommended. Both resources are used heavily in this section. In this chapter, we will discuss the the differences between CPUs and GPUS (Section 2.1.1). Then (in Section 2.1.2) a GPU programming model will be discussed. Finally, the programming techniques and GPU architecture will be discussed.

### 2.1.1 The what and why, CPU vs. GPU

The Central Processing Unit (CPU) in a modern PC, is composed of millions of transistors. Each year the number of transistors in a CPU increases. In 1965, Intel's chairman Gordon Moore predicted that the number of transistors that could be fabricated on a single processor die would double each other year. This prediction still holds this day. However, the computational power does not double each year. Approximate numbers from [3] indicate that each year the

- Capability of a processor increases 71 percent.
- Transistor speed increases 15 percent.

- Memory bandwidth increases 25 percent.
- Memory latency increases 5 percent.

It is easy to see from this list that the computing power increases much faster than the communication ability. This has been a known problem in VLSI design for a long time. Until recently the solution to this problem has been to add layers of faster memory between the processor and the main memory. We have registers, on-die caches, off-die caches, main memory, disk and tapes. This additional caching requires a lot of transistors. On a modern CPU the computing unit only occupies a small part of the total space on a chip. As an example, the ALU (Algorithmic-Logic Unit) on the Itanium 2 occupies 6.5% of the total die space [4]. In addition to cache, the processor also uses prefetching to speed up memory access. To make sure that the correct memory is prefetched additional transistors have to be used to implement techniques like branch-prediction, instruction snooping, etc. For a general-purpose processor like the CPU, the methods mentioned above and the transistor resource utilisation has been a successful tactic for the last decades. However, a major drawback to this tactic has been that it is inherently optimised for single threaded programs with no parallelism.

For applications that actually have data parallelism, the route that the traditional CPU has gone is not the best option. A good example is media processing. A typical media operation is to execute a small series of computations on each data element in the media stream. For these kinds of operations, both the cache and the extra prefetching is useless. In fact, they may both hurt the execution speed because of cache thrashing. Another similar example of parallel computation is 3D-rendering. As 3D applications, and especially games, required more and more computational resources during the 1990's it became apparent that the CPU wouldn't be able handle the load by itself. The first modern consumer GPU was born in 1996. [1] By the turn of 2000 almost all consumer PCs contained a GPU, and today it is ubiquitous. A timeline with the computing power of GPUs and CPUs is shown in Figure 2.1.

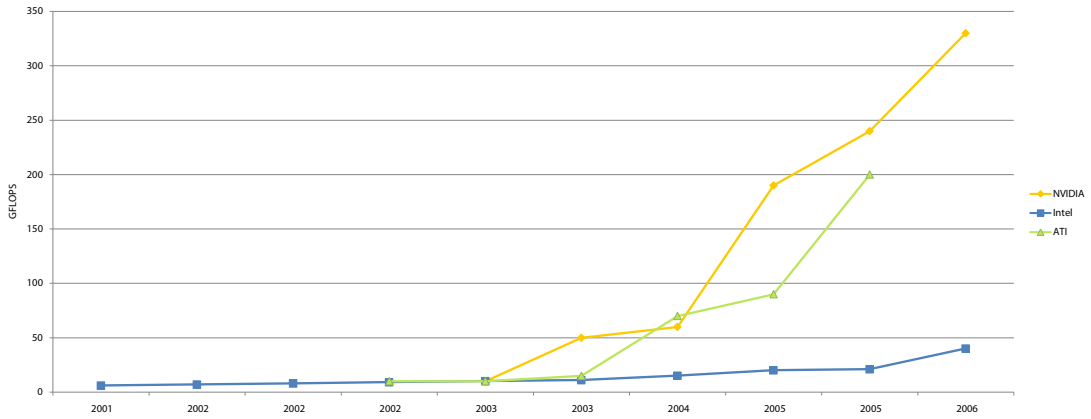
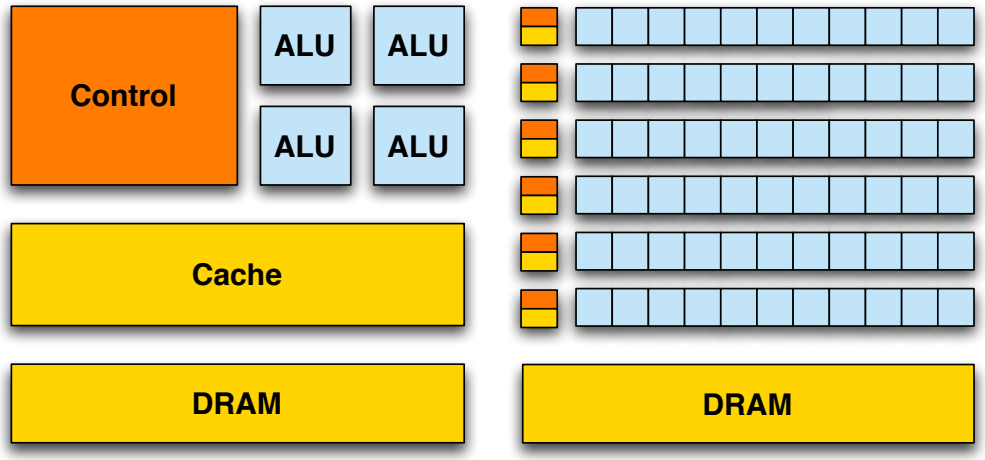



Figure 2.1: Performance timeline, GPU vs. CPU. Figure reproduced from [1]

The GPU is a good example of special-purpose hardware that is tailored for one specific task, 3D rendering. In the beginning, 3D rendering was also all it could be used for. As the GPUs evolved they have become more flexible, and it is now possible to program them to do a

[1] We define a modern GPU to be a graphic board with 3D processing abilities, not just a framebuffer.

variety of tasks. In a recent survey carried out in [1], a number of applications were listed, such as rigid body simulations, linear algebra solvers, fluid simulations, image segmentation, Fast Fourier Transforms (FFTs), and Discrete Cosine Transforms (DCTs), database searches, to mention a few. Not only is it *possible* to implement these methods on the GPU, for most of these applications the GPU will outperform the CPU several-folds. How is this possible? As mentioned, one of the main problems with the CPU is that it assumes serial data and that it focuses its transistors on techniques to 'avoid' memory latency. The GPU, on the other hand, assumes massively parallel data and a relatively simple series of computations on this data, so it can concentrate its transistors on the computational part. An illustration of this can be seen in Figure 2.2 taken from a recent NVIDIA document [5].




(a) CPU (b) GPU

Figure 2.2: The relative number of transistors used for ALU and cache by a CPU and a GPU. Figure reproduced from [5].

This causes the GPU to have much more raw computing power than a similarly sized and priced CPU. In Table 2.1, a number of high-end GPUs and CPUs are compared. Price esti-

| xPU \ Property | Cost | Bandwidth | GFLOPS |
|---|---|---|---|
| Radeon X1900 XTX | $400 | 51.0 | 240.0 |
| GeForce 7900 GTX | $340 | 38.4 | 240.0 |
| Geforce 8800 GTX | $599 | 86.4 | 345.0 |
| Intel DualCore X6800 | $999 | 8.5 | 50.0 |

Table 2.1: Comparison of high-end CPUs and GPUs on some key figures.

mates are gathered from [7, 6], GPU performance estimates from [8] and CPU performance estimates from [1, 9].

### 2.1.2 Stream Programming Model

As we have seen in the previous section, the GPU has massive amounts of raw computational power compared to the CPU. The main reason for this is that it is tailored for 3D graphics programming, which in turn implies that we cannot apply the standard CPU programming models directly to the GPU. Instead one typically use a model called the Stream Programming Model which fits nicely on the GPU [4]. The stream model consists of two key concepts [3]:

- **Stream** An ordered set of data consisting of the same data type. Streams can be of any length, but they are usually long. The datatype can be arbitrarily complex. A stream can be copied, divided into substreams and operated on by *kernels*.
- **Kernel** A basic operation consisitng of a small number of operations. A kernel performs a computation on one or more *streams* and outputs one or more *streams*. [2] The kernel output are functions only of their inputs. The operation on one element of a stream is never dependent on the computation of another element in the same stream. This is a key property to make the model easily parallelisable.

Applications are made by chaining multiple kernels together. In Figure 2.3, an example stream model from [3] is shown. The application is the 3D graphics pipeline. There are two kinds

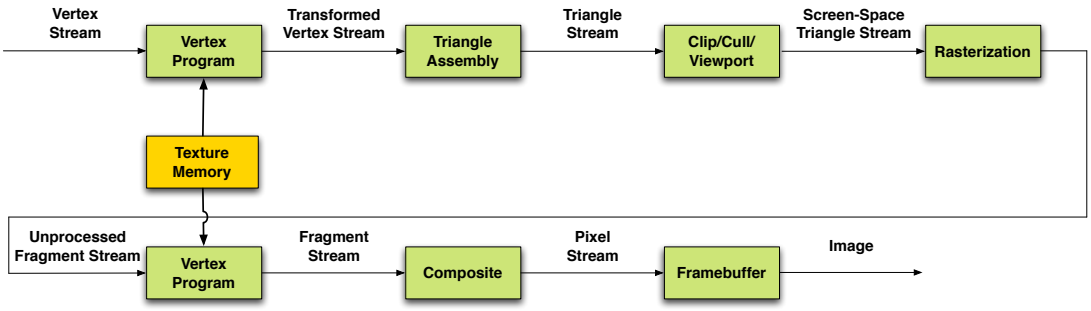



Figure 2.3: The graphics pipeline as a stream model. Reproduced from [3].

of parallelism exposed by this model. First, the data-level parallelism is quite easy to imagine because each element in a stream is of the same data type and they undergo the same operations. Also, since we can chain multiple kernels, this model can be task-parallelised and deeply pipelined.

### 2.1.3 The Graphics Pipeline

We have seen that a GPU contains massive computing power and that the stream programming model fits nicely to its graphics pipeline. In this section, we will take a closer look at the internal structure of a GPU and how to actually execute kernel programs on a GPU.

#### The rendering process

The rendering process starts with the application creating a scene description based on polygons. The process of rendering these polygons to the screen is called display traversal [10].

[2] A very similar concept to a kernel is the *map* function known from functional programming.

We can break this process into three big parts:

1. Geometry processing
2. Fragment operation
3. Compositing and outputting to buffer

This process used to be fixed. This means each part of the process had a fixed number of steps and features, and the developer could only set some parameters controlling the rendering. These parameters usually controlled:

- Transformation from local models 3D-coordinates to 2-D screen coordinates.
- Lighting and shading. This could be the number of light sources and which colours.
- Some parameters controlling depth-culling and other efficiency measures.

The last sub-process would output the end result to the GPU's frame buffer. The frame buffer is the part of memory on the graphics board that will be rendered to the screen. Recently, however, the fixed function pipeline was replaced with a much more flexible pipeline. Such a pipeline is shown in Figure 2.4.

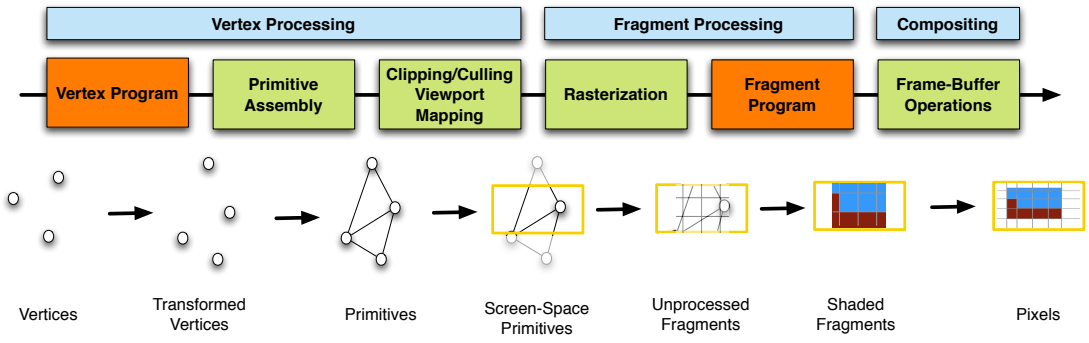



Figure 2.4: The new programmable graphics pipeline. Reproduced from [11].

### Programmable pipeline

The display traversal in a programmable pipeline is the same as for fixed function pipelines, but as the name implies, it is now possible to program some parts of the pipeline. This can be done with special micro programs, called *shaders* in the graphics literature. Two kinds of shaders exist, *fragment shader* and *vertex shader*. Until quite recently, these microprograms had to be developed in assembly language., However, several special purpose high-level languages has been developed in the last three years. The three main ones are:

1. GL Shading Language (GLSL), developed by the Open GL committee.
2. High Level Shading Language (HLSL) which is Microsoft's shading language.
3. C for graphics (Cg) which was developed by NVIDIA

The three languages are very similar and show strong similarities to C. In this thesis, HLSL will be used. The reason for this choice was the author's previous exposure to the accompanying DirectX framework. The choice of framework and language showed to have very little effect on the thesis work and porting our methods to another framework should be fairly easy.

We will now take a look at the different processing stages and their programmability.

#### Geometry processing

As mentioned, the scene graph is represented as polygons. The application feeds these polygons to the GPU as vertices. A vertex may contain a number of properties. The obvious ones are position, colour, opacity and texture coordinates. The developer may specify any kind and any number of extra properties by defining a custom *vertex declaration*. As vertices enter the GPU, the first step is to perform linear operations on these vertices. This is done by a microprogram called the *vertex shader*. The vertex shader may alter any of the properties of the vertex. The usual operations performed are transformation from model coordinates to screen coordinates and per-vertex lighting. In addition to the vertex properties, the vertex shader may also access per frame parameters called *uniforms*. An example of a uniform is the matrix used to transform the vertex into screen space. The vertex shader may also access the texture memory. An important limitation, however, is that unlike vertices may not be deleted or created in the shader. Only modification of the properties are allowed. After being processed by the vertex shader, the vertices are assembled into geometric primitives: points, lines and triangles. The primitives are clipped, culled, mapped to the viewport and rasterised.

#### Fragment operation

As the primitives are rasterised, the GPU decomposes each primitive into fragments. Each fragment corresponds to a pixel in screen space. For each fragment, a microprogram called the *fragment shader* is run. The inputs to the fragment shader are interpolated versions of the output values from the vertices contributing to the primitive owning the fragment. Like the vertex shader, the fragment shader may also use uniform variables and texture memory for its operation. In addition, the fragment shader may use *dependent texture reads*. These are two texture reads where the memory position of the latter is dependent upon the content of the first read.

#### Compositing and outputting to buffer

After each fragment is processed in the fragment shader, we have a fragment ready to be put on the screen. Still there are some fixed processing options left. These include:

- **Culling** The GPU can discard a fragment based on a depth test, scissor test, alpha test or stencil test.
- **Blending** The fragment colour can be combined with the colours already in the buffer.

Blending is the act of combining the fragment created in the current rendering pass, the *source fragment*, with the fragment already present in the output buffer, the *destination fragment*. Denoting each fragment as a vector $\mathbf{c}_i = \{r_i, g_i, b_i, a_i\}$, the fraction of the fragment used in the blending is $\mathbf{f}_i$ where $\mathbf{f}_i$ is the vector $\mathbf{c}_i$ with each component multiplied by a constant $f_i$.

$i \in \{src, dst\}$. This gives us

$$\begin{aligned} \mathbf{f}_{src} &= f_{src}\mathbf{c_{src}} \\ \mathbf{f}_{dst} &= f_{dst}\mathbf{c_{dst}} \end{aligned} \tag{2.1}$$

Any number may be used for the constants but the most used are:

$$\begin{aligned} f_{src} &= a_{src} \\ f_{dst} &= 1 - a_{src} \end{aligned} \tag{2.2}$$

which is called alpha blending.

If the fragment passes all culling tests and is blended, we can finally render it to the output buffer. Usually the output buffer is the *frame buffer*, a special memory location on the graphic board that represents the screen. With a programmable pipeline we can also render to texture memory or even several output buffers at once.

### 2.1.4 General-Purpose computation on GPUs (GPGPU)

In this section, we will look at how we can employ the GPUs described in the previous section for general purpose computation (GPGPU) using the stream model.

Figure 2.5 shows the workflow of a typical GPGPU program.

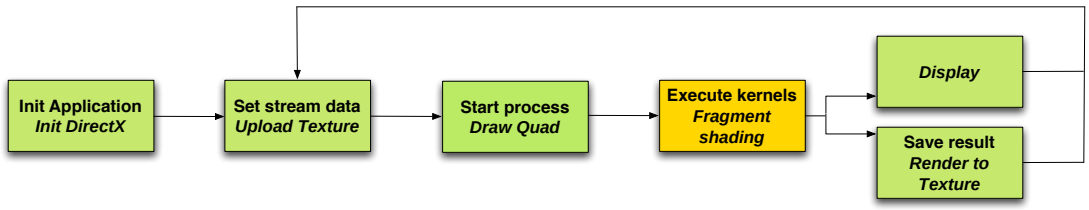



Figure 2.5: The workflow for an image filter

#### Streams

The streams of the GPU are textures. These are very similar to arrays used on the CPU. They can contain up to four 32-bit floating point pr. element. These are called r,g,b,a (red, green, blue and alpha). The names stem from graphics, but the names are irrelevant when programming GPGPU. We hence have basically the same as a *float[x][4]* on the CPU. However, unlike the CPU, these can be *swizzeled* without cost. To swizzle is to rearrange or select the components of a single element. In Listing 2.1, a small example of swizzle operation can be seen.

```
float4 vector = {1,2,3,4}; // A four component element
vector.rgba; // same as just using vector, gives 1,2,3,4
vector.rrrr; // will select 1,1,1,1
vector.gb; // will select only element 2,3
```

Listing 2.1: Nine Tap Vertex Shader

The stream has to be explicitly uploaded from main memory to GPU memory to be usable.

#### Kernels

The GPU has two types of programmable processors, the vertex processor and the fragment processor. For GPGPU the fragment processor is usually the preferred choice. There are two reasons for this. First, GPUs typically have more fragment processors than vertex processors. As an example the GeForce 7800 GTX has 8 vertex processors and 24 fragment processors. Second, the fragment processor has the ability to write directly to texture memory. The output from the vertex processor, on the other hand, has to travel through the rasteriser and fragment processor. Which makes it more difficult to program. Thus, most of the computational work in GPGPU programs is done in the fragment processors, but the vertex processors can perform some auxiliary functions as we shall see in later chapters.

#### Invocation

The fragment shaders are executed by drawing geometry. Usually, we draw a quadrilateral with the same amount of fragments as the number of elements in the desired output. In Figure 2.6(a), we can see an example quadrilateral drawn. One example fragment is shown with its interpolated texture coordinates. In Figure 2.6(b), a quad with interpolated colours are shown to help visualise the interpolation. By adjusting the texture coordinates in the vertices used in the quadrilateral, we can adjust which texture memory each fragment shader uses. We will see examples of this in the later chapters.

#### Reading back

The primary method of reading back from GPU memory is by rendering to texture. It is also possible to read back from the frame buffer, but this is much slower. After the result is rasterised to the texture we have to explicitly transfer the results back from GPU memory to main memory. This operation flushes the GPU pipeline and should be used with care. In many applications, discussed in this thesis, including our ultrasound framework, the result is to be displayed on the screen, so it isn't necessary to read back to the CPU at all. This speeds up the applications noticeably.

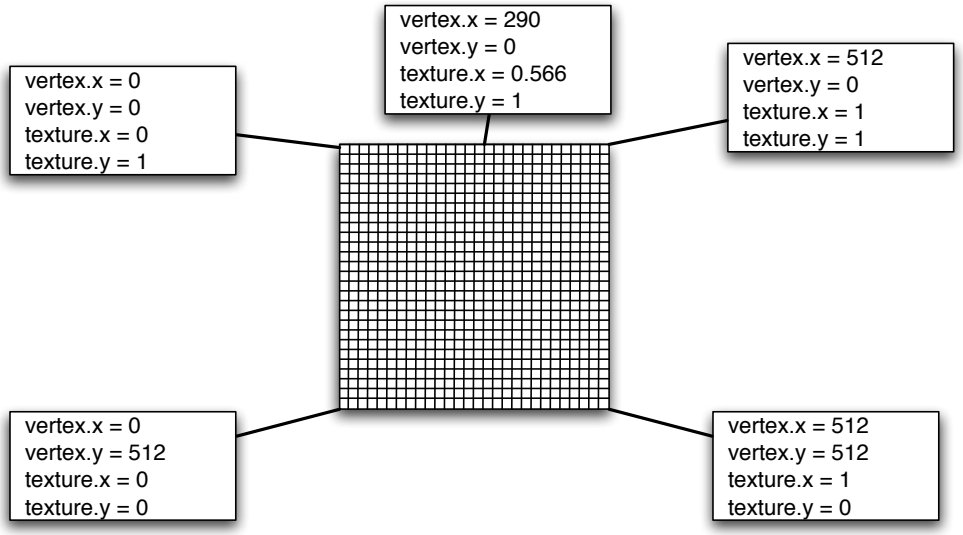
#### Multiple output streams

In the stream model, it is possible to output several streams from a kernel. This is also possible to do on the GPU with the use of multiple render targets (MRT). One important limitation is that the streams written to (the textures) have to be of the same size.

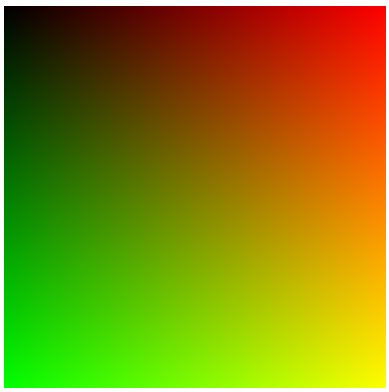
### 2.1.5 Suitable applications for the GPU

As mentioned in the introduction, [1] contains a comprehensive survey over applications that have been implemented on the GPU. Larsen [12] describes a GPU implementation of the Discrete Cosine Transform. This is a nice example of a transform In this section we will discuss which traits an application should have to be successfully ported to the GPU. *Successfully* can be defined in various ways, but some important keywords are speedup, more power effective, faster, more computation per dollar.

Arguably, the most important trait for a successful GPU application, is that it has high arithmetic intensity [13].




(a) Example of texture coordinates



(b) Interpolation of colours

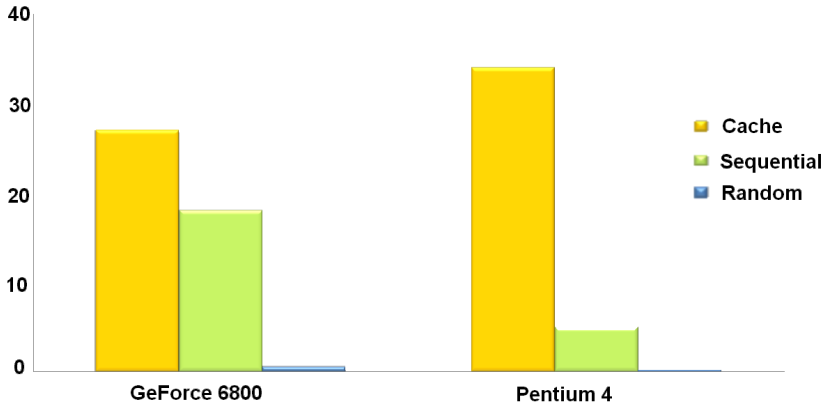
Figure 2.6: Interpolation example

arithmetic intensity = operations / words transferred

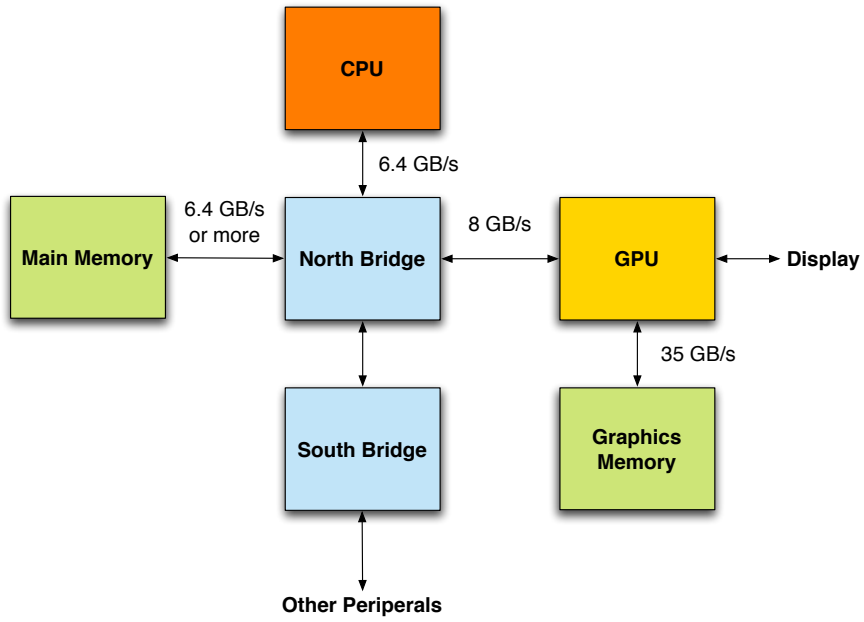
As an example, consider simple vector addition: Even if the vectors are big this will certainly go faster on the CPU than on a GPU. There is just too much overhead involved in copying the vectors from main memory to GPU memory and back again compared to the number of operations. In addition, it is important that we have enough data to work with. Even if we have a complicated operation, if the data is small, the overhead of invoking the GPU will dominate the time and the CPU will be faster. The question of how complicated a function should be, or how big the data should be, to make it worthwhile using the GPU, is a difficult one. There is no simple answer, but a rule of thumb could be: at least a couple of thousand of elements and several multiple-add operations.

How the data is accessed is another important element. In GPU Gems 2 [14], the memory read speed on a GPU and a CPU was measured. The GPU reading has become a bit faster

since, but the general trend should still apply. The GPU reads sequential and random data faster than the CPU, especially the sequential, but reading from the cache is much faster on the CPU. These data are easily understood when we take the discussion from Section 2.1.1 into account, supporting the view of the GPU as a stream processor.




(a) Reading a single floating point with different memory access methods.




(b) How the GPU connects to the rest of the PC

Figure 2.7: GPU memory overview. Figures reproduced from [14].

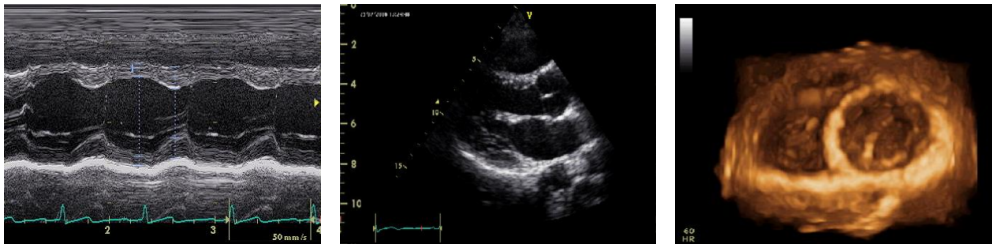
Flow control is one of the most important concepts in computation. Because of the highly parallel nature of the GPU it doesn't support flow control very well. On old architectures it didn't even support loops, but that has luckily been handled in the later years. It is possible to do branching in a fragment or vertex program, but extensive branching is not recommended. There exists a couple of methods to avoid branching, [1] has a good overview of the different methods.

In parallel computing, the *gather* and *scatter* memory operations are two basic primitives that can be used to explain the memory pattern of an algorithm or the capability of an API. In stream computing, *gather* occurs when a kernel processing a stream element requests information from other parts of the stream [13]. *Scatter*, on the other hand, distributes information to other streams. GPU supports *gather* very well, but not *scatter*. Hence, applications should be rewritten if they use scatter. One example of this is sorting. The scattering quick-sort is not a good fit for the GPU, while the gather oriented bitonic sort is [15].

## 2.2 Ultrasound

Medical ultrasound is an non-invasive, real-time, portable method of performing medical imaging. These properties makes it an attractive alternative to other imaging methods like Magnetic Resonance Imaging (MRI) and Computed Tomography (CT) that require large imobile equipment. Ultrasound is used in many branches of medicine, but especially in cardiology, obstetrics and gastroenterology. An introduction to modern ultrasound can be found in [16].

An ultrasound machine shoots ultrasound pulses, in the 5-15 MHz range, from a transducer into a body. The wave propagates into the tissue, while a small amount is reflected along the path. The main condition for the reflection is 'impedance jumps' which occur at the interface between two tissues with different sound transmission interfaces. The transducer then measures the strength and time of the reflected waves. Based on the time the pulse was sent, the distance sound travels in soft tissue, and when the reflection returned, the distance the returning wave has travelled into the tissue can be calculated. The resulting visualisation is a beam where the pixel intensity is based on the energy of the reflected wave. By adding more ultrasound probes in the transducer and changing the angle at which the ray is shot 2D and 3D ultrasound becomes possible. In Figure, 2.8 different ultrasound modes are shown.



(a) 1D (time on the X-axis) (b) 2D (c) 3D

Figure 2.8: Different Ultrasound modes

### 2.2.1 Ultrasound noise formulation

One of the fundamental problems faced by ultrasound imaging is the presence of speckle noise. This kind of noise is common to all coherent systems and affect both human interpretation and computer-assisted techniques. Goodman (1976) [17] studied the statistical properties of speckle noise and showed that when the resolution cell is small compared to the spatial detail in the object and the image has been sampled coarsely enough the degradation at any pixel can be assumed to be independent of degradation at all other pixels. He also showed

that the degradation at each pixel can be modelled as multiplicative noise. In Jain [18], this is formulated as:

$$f(x,y) = g(x,y)\eta_m(x,y) + \eta_a(x,y) \tag{2.3}$$

$g(x,y)$ is a piece-wise constant 2D function representing the unknown error-free image, $f(x,y)$ is the observed values of $g(x,y)$, $\eta_m(x,y)$ is the multiplicative noise, $\eta_a(x,y)$ the additive noise, $(x,y) \in \mathbb{R}$. The additive noise $\eta_a$ is relatively small compared to the multiplicative noise in ultrasound images, so we can approximate Equation 2.3 by

$$f(x,y) = g(x,y)\eta_m(x,y) \tag{2.4}$$

We can seperate noise and signal by applying the log transform:

$$log(f(x,y)) = log(g(x,y)) + log(\eta_m(x,y)) \tag{2.5}$$

Actually this log transform is done anyway on commercial ultrasound systems because of the limited dynamic range of the monitors. We can write 2.5 as

$$f(x,y)^l = g(x,y)^l + \eta_a(x,y)^l \tag{2.6}$$

The subscript of the noise is changed from $m$ to $a$ to point out that the noise is now actually the same as white Gaussian additive noise. So the image enhancement algorithms has to deal with additive Gaussian noise instead of multiplicative noise. The former task is much easier than the latter.

There exists other formulations of ultrasound speckle noise, most of them quite complicated. The formulation presented here is advanced enough for our purposes and has been shown to work well in practice. For these reasons, this noise model will be used.

### 2.2.2 Ultrasound image restoration

Image restoration is an old field in computer science and a large number of methods has been developed. Most methods proposed for ultrasound image enhancement are based on these existing methods, but adjusted to suit the characteristic of the ultrasound imaging process. In [19], some methods are listed: Median filtering [20, 21], Wiener filtering [18], adaptive weighted median filtering [22], adaptive speckle reduction (ASR), wavelet shrinkage and Nonlinear Anisotropic Diffusion (NAD).

In this thesis, we will only look at some of the methods proposed. We will concentrate most of our effort on the multiresolution methods. These methods are based on wavelets described in Chapter 2.3. This thesis has been written in cooperation with the medical industry and the selection of techniques are based on the techniques that are actually used in practice [23]. In literature, it is popular to establish statistical methods that will work on all kinds of ultrasound equipment. In practice, it is often better to adjust the enhancement parameters to suit different equipment. This thesis will assume hand-tuned instead of auto-tuned parameters.

## 2.3 Wavelets

Transforms on digital signals are used to obtain information that are not readily available by analysing the signal in its raw form. Examples include the Hilbert transform, the Wigner distribution, the Radon Transform and the most popular one: the Fourier Transform (FT).

One reason for this popularity is due to the Fast Fourier Transform (see Cooley et al. [24] for more information), which is a fast algorithm for computing the Fourier Transform. The FT transforms a signal from the raw format into the frequency domain. The raw format is usually the time domain, but when we are analysing images it is the spatial domain. The Fourier transform is defined as:

$$\hat{f}(\omega) \stackrel{\text{def}}{=} \int_{\mathbb{R}} x(t)\, e^{-i\omega t}\, dt$$

One important fact to notice is that the Fourier transform is integrated over the entire time range. This gives Fourier one of its most important properties: A transformed signal will contain the exact frequencies that existed in the original signal. All time (or spatial) information is lost. [3] This may or may not be a big problem. For so-called stationary signals it has no effect. Stationary signals are signals which are constant in their statistical parameters over time But for non-stationary signals we loose potentially important information in our transform. Several schemes have been devised to counter this. One approach has been the Short-Time Fourier Transform (STFT). The STFT assumes that the signal is stationary for some portion of the signal. It then cuts the signal into small segments. The problem is that now we no longer integrate over the whole time segment. According to Heisenberg Uncertainty Principle (HUP), we now no longer get exact frequency information. We can adjust this by changing the segment length. If we let $\sigma_t$ denote a time-spread around a time instant, and $\sigma_\omega$ denote the frequency spread around an instant, HUP states:

$$\sigma_t^2 \sigma_\omega^2 \leq \frac{1}{4}$$

For our transformations the tradeoffs can be summed up as:

| Segment size | Time resolution | Frequency resolution |
|---|---|---|
| Small | Good | Poor |
| Large | Poor | Good |

Another approach is using a related transform, the wavelet transform.

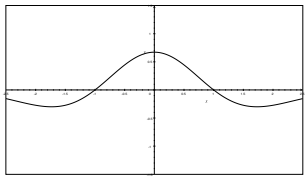
### 2.3.1 Wavelets

In this chapter, we will present the current state as presented by [25, 26, 27, 28].

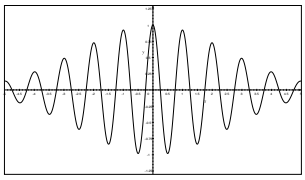
The word wavelet is a direct translation from French 'ondolette' which means small wave, and this is exactly what wavelets are. Functions which oscillates like a wave in limited space. Wavelets have to satisfy certain requirements. $\psi(x)$ is a wavelet given that oscillates, it averages to zero and that it is well localised. We use the wavelets quite similar to the way sines and cosines are used in Fourier. But an important concept in wavelet analysis is *scale*. One choses a wavelet, scales it and translates it to see its correlation with the signal being analysed. When the signal correlates to a large (rude) scale we can see the course features, while small scales will show the fine features. It is often said you can 'see the forest *and* the trees'.

The prototype function is called the mother wavelet. The scaled and translated wavelets are called the baby wavelets. Given a wavelet $\psi(x)$ we create baby wavelets $\psi_{s,\tau}(x)$ by scaling

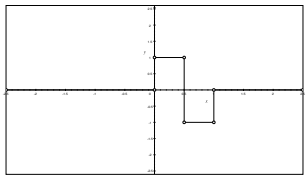
[3] Since the transform is reversible we don't actually 'loose' any information. It is just unavailable in the transformed signal.



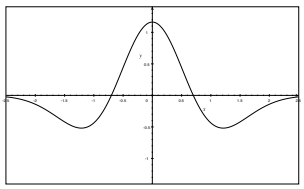
(a) Hermitian hat



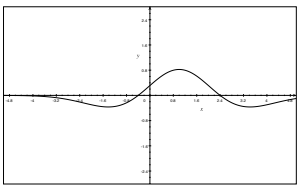
(b) Morlet with $z_0 = 5$



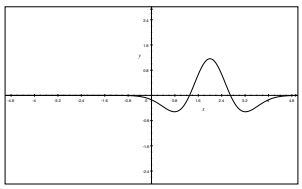
(c) Haar



(d) Mexican hat



(e) Mexican hat with $\tau = 1$ and $s = 2$



(f) Mexican hat with $\tau = 2$ and $s = 1$

Figure 2.9: Examples of different wavelets

with $s$ and translating by $\tau$:

$$\psi_{s,\tau}(x) = \frac{1}{\sqrt{s}}\psi(\frac{x-\tau}{s}) \tag{2.7}$$

### 2.3.2 Continuous wavelet transform

As indicated in the introduction we are interested in using the wavelets in transforms. The continuous wavelet transform (CWT) is defined as:

$$W_\psi(s,\tau) := \int_{\mathbb{R}} f(x)\psi^\star_{s,\tau}(x)dx \tag{2.8}$$

where $\psi^\star$ is the complex conjugate of the wavelet $\psi$. To use a wavelet in the CWT it has to abide the admissibility condition

$$0 < C_\psi := \int_{\mathbb{R}} \frac{|\hat{\psi}(\omega)|^2}{|\omega|}d\omega < \infty \tag{2.9}$$

where $\hat{\psi}$ denotes the Fourier transform of the wavelet $\psi$. In addition, the wavelet has to be chosen from the space of $L^2(\mathbb{R})$. These conditions are usually satisfied by:

$$\int_{\mathbb{R}} \psi(x)dx = 0 \text{ and } \psi(0) = 0$$

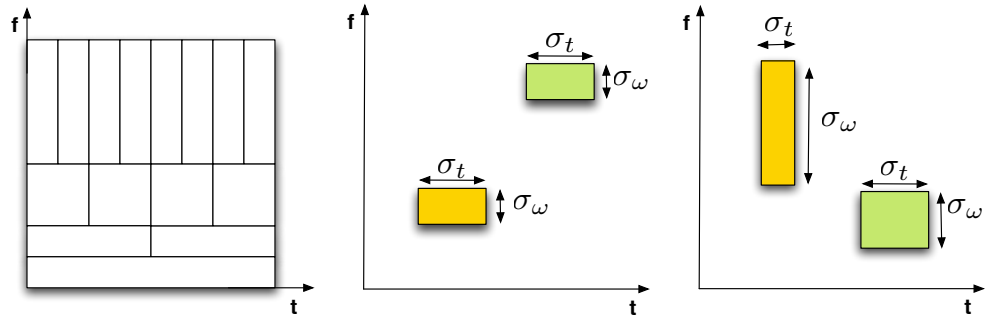
and that $\psi(\omega) \to 0$ as $\omega \to \infty$ fast enough to make $C_\psi < \infty$.

It can be shown [4] that the CWT can approximate any function in $L^2(\mathbb{R})$ at arbitrary precision. As a matter of fact, we can restrict our scaling factor to $s = 2^j$, $j \in \mathbb{Z}$ and still be able to represent any function. This transform, $W_\psi(2^j, \tau)$, is called the *Dyadic* transform.

[4] See [27] for details

The inverse wavelet transform is defined as

$$f(x) = \frac{1}{C_\psi} \int_{\mathbb{R}} \int_{\mathbb{R}} W_\psi(s,\tau) \frac{\psi_{s,\tau}(x)}{s^2} d\tau ds \tag{2.10}$$




(a) Wavelet decomposition (b) Short-Time Fourier (c) Wavelet

Figure 2.10: Time-Frequency resolution. Figures reproduced from [27]

Some example wavelets can be seen in Figure 2.9.

Since we are working with computers we would like to see how we could make this transform discrete. This however, is not straight forward, and we have to make a detour into Multiresolution Analysis first.

### 2.3.3 Multiresolution Analysis

Multiresolution Analysis (MRA) is a method where a signal is successively decomposed into coarser details and approximations. We project a signal onto a sequence of approximation spaces

$$\{0\} \subset \dots V_{j+1} \subset V_j \subset V_{j-1} \dots \subset L^2(\mathbb{R}) \tag{2.11}$$

where the index $j$ indicates resolution scale $2^j$. We can restrict ourselves to these subspaces since we are working with dyadic transformations as discussed in Section 2.3.2. The differences between two approximation spaces are contained in the *detail space* $W_j$:

$$V_j \cap W_j = 0 \text{ and } V_{j-1} = V_j \oplus W_j \tag{2.12}$$

Another way of saying this is that $V_j$ are the sums of the $W_j$. We can write this explisitly as: (this is the same as writing Equation 2.12 out)

$$V_J = \bigoplus_{j>=J+1} W_j \tag{2.13}$$

If the vector spaces $V$ and $W$ also satisfy:

- The basis functions that span the space are orthogonal to its translates by integers.
- The approximation at a resolution is self-similar:

$$f(\cdot) \in V_0 \Leftrightarrow f(2^{-j}\cdot) \in V_j$$

- In addition to the fact that they are nested vector spaces, all functions in all subspaces can approximate all function in $L^2(\mathbb{R})$:

$$\begin{aligned} \bigcap_{j\in\mathbb{Z}} V_j &= 0 \\ \bigcup_{j\in\mathbb{Z}} V_j &= L^2(\mathbb{R}) \end{aligned} \tag{2.14}$$

they are called a multiscale approximation of $L^2(\mathbb{R})$ [27].

#### Scaling and Wavelet functions

Since we would like to approximate an arbitrary signal within each subspace $V_j$ we are looking for basis functions that span the all the subspaces. Actually, it can be proven [5] that if $V_j$ is a multiscale approximation in $L^2(\mathbb{R})$ then there exists a single *scaling* function $\varphi$ such that

$$\left\{ \frac{1}{\sqrt{2^j}} \varphi\left(\frac{x - k2j^2}{2^j}\right) \right\} \tag{2.15}$$

is an orthonormal basis of $V_j$. From this definition we can see that $f(x) \in V_{j+1} \Leftrightarrow f(2x) \in V_j$. Based on our previous assumptions we can write this out as an explicit recursive function:

$$\varphi(x) = \sqrt{2} \sum_{k\in\mathbb{Z}} h_k \varphi(2x - k) \Leftrightarrow \frac{1}{\sqrt{2}} \varphi\left(\frac{x}{2}\right) = \sum_{k\in\mathbb{Z}} h_k \varphi(x - k) \tag{2.16}$$

$h$ is called the filter mask for scaling function $\varphi$.

For $W_j$, we call the basis spanning the vector space *wavelets* and they are denoted $\psi$. Every basis function $\psi_{j-1}$ of $W_{j-1}$ is orthogonal to every basis function $\varphi_{j-1}$. Analogous to Equation 2.15 we have:

$$\left\{ \frac{1}{\sqrt{2^j}} \psi\left(\frac{x - k2j^2}{2^j}\right) \right\} \tag{2.17}$$

is an orthonormal basis of $W_j$. And analogous to Equation 2.16 we have

$$\psi(x) = \sqrt{2} \sum_{k\in\mathbb{Z}} g_[k \psi(2x - k) \Leftrightarrow \frac{1}{\sqrt{2}} \psi\left(\frac{x}{2}\right) = \sum_{k\in\mathbb{Z}} g_k \psi(x - k) \tag{2.18}$$

where $g$ is the detail filter, and can be calculated as:

$$g[k] = \langle \psi(x), \sqrt{2}\varphi(2x - k) \rangle$$

The linkage between multiscale analysis and wavelet theory was first done by Mallat [29] and it enables us to create a Discrete Wavelet Transform.

### 2.3.4 Discrete Wavelet Transform

The Multiscale Analysis presented above gives us a method of decomposing a signal into components of different resolutions. We get the details by applying the wavelet function $\psi$

[5] see [29]

(or its filter mask $g$) and the approximations with the scaling function $\varphi$, (or its filter mask $h$). To the coarse level we can further apply the filters on the approximation recursively. If $\lambda(x)_j$ is the approximation at level $j$ and $\gamma(x)_j$ the detail we can write this as:

$$\begin{aligned}\lambda(x)_{j+1} &= \sum_{k=-\infty}^{k=\infty} h(k)\lambda_j(2x+k) \\ \gamma(x)_{j+1} &= \sum_{k=-\infty}^{k=\infty} g(k)\lambda_j(2x+k+1)\end{aligned} \tag{2.19}$$

This algorithm is called the Discrete Wavelet Transform. Note that unlinke the CTW the DWT contains no redunancy. A block diagram showing three levels of the DWT is shown in Figure 2.11.

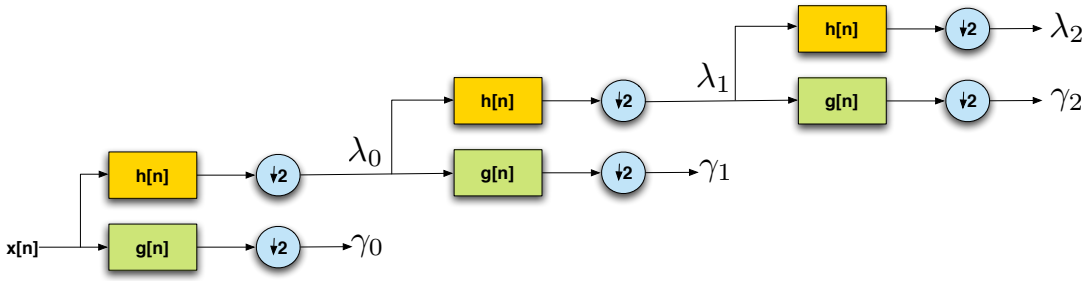



Figure 2.11: An example with 3 levels DWT

The Discrete Wavelet Transform is sometimes called the Fast Wavelet Transform as an analogy to the Fast Fourier Transform (FFT). Actually, the DWT is faster than the FFT, its complexity is $O(n)$ compared to FFT's $O(n\ log(n))$.

It is important to understand that the DWT is not simply sampling the CWT. The wavelets (and the associated filters) now have to be chosen carefully so that they are a basis of $L^2(\mathbb{R})$ and in the form of Equation 2.17. So our choice for wavelets is quite restricted. If we further restrict our filters $g$ and $h$ to have finite response (FIR)and be orthogonal we can create the inverse filters $h'$ and $g'$ in such a way that we get perfect reconstruction. We call this reconstruction *synthesis* and it can be written as

$$\lambda_j(x) = \sum_{k=-\infty}^{k=\infty} h'(k)\lambda_{j-1}(x+k) + \sum_{k=-\infty}^{k=\infty} g'(k)\gamma_{j-1(x+k)} \tag{2.20}$$

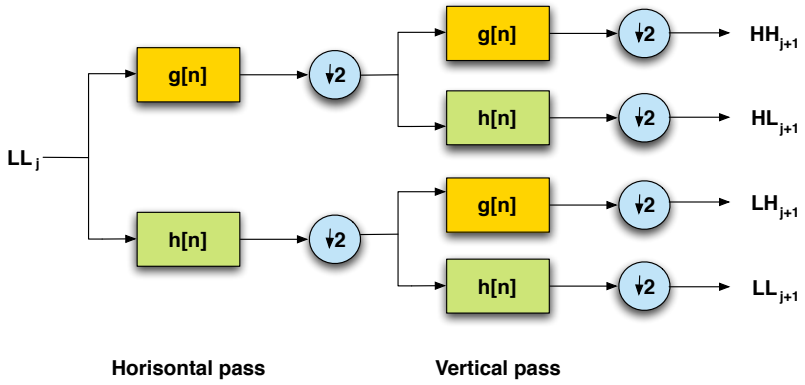
Creating a wavelet that meets all the required conditions is quite tedious and outside the scope of this thesis. We shall instead use some of the already designed wavelets that have been proven to satisfy the needed conditions.

### 2D Discrete Wavelet Transform

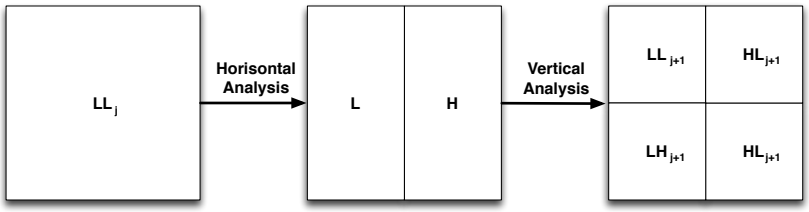
Since this thesis concentrates on images, we would like to extend the DWT to two dimensions. There exists several methods for doing this. The easiest and most used is the separable

approach. The separable 2D DWT is achieved by first applying the 1D DWT on the rows, and then on the columns. This gives us four decomposed signals for each level of DWT:

- LL: The approximation. This is the signal that will be recursed further upon.
- LH: Horisontal approximation, vertical detail. This signal will contain specifically the vertial details and can be used if one wants to apply a special filter for the vertical details. (Like searching for vertical lines.)
- HL: Horisontal detail, vertical approximation.
- HH: Detail in both vertical and horisontal direction.




(a) One level of 2D forward DWT



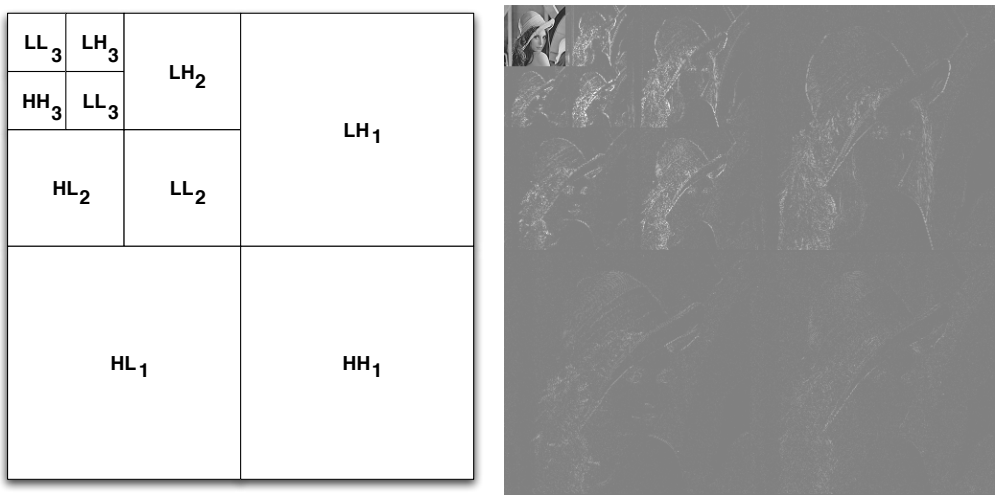

(b) How (a) is usually visualised on screen.

Figure 2.12: 2D Forward DWT

In Figure 2.12, the 2D Forward DWT is shown, Figure 2.12(b) shows how the four signals are visualised. LL is used for further decomposition, Figure 2.13 shows how a three level forward DWT is visualized. Backward DWT is done the exact opposite way, first on the columns and then on the rows. The other approach to 2D is called the non-separable approach and is still on a basic research level. It will not be discussed further in this thesis.

### Choice of Wavelets

Unlike the FT, we have to choose which wavelets to use for the wavelet transform. The main reason we are using wavelets in this thesis is for denoising. Therefore, the wavelet we chose




(a) Three level forward DWT (b) The forward DWT on a real picture.

Figure 2.13: Multiple levels of 2D DWT

have to fit this purpose. In [28], there is a discussion of the important properties the wavelet should have for denoising. There are two main points.

- Denoising is facilitated by a sparse environment, i.e. the detail should contain few non-neglible coefficients.
- The visual quality is important.

The first objective is met by having many *vanishing moments* $\tilde{N}$ and as small *support size* $K$ as possible. The theoretical limit is $K = 2\tilde{N} - 1$ and is achieved in the Daubechies wavelets. This is a family of wavelets usually denoted $db\tilde{N}$. Visual quality is usually achieved by having a regular and symmetric wavelet. Daubechies symlets $sym\tilde{N}$ are the most compactly supported symmetric wavelets. In this thesis, the Daubiches $9/7$ and $5/3$ wavelets are used. They are chosen because they are the most used in denoising litterature and they are used in the JPEG2000 standard (indicating good visual quality) as well.

## 2.4 Filters

As part of this thesis, one goal was to assess the possibility of doing simple spatial filters used in ultrasound processing on the GPU. In this section, we will take a look at these filters. The theory of these filters can be found in any introductionary textbook on image processing like Gonzales and Woods [30].

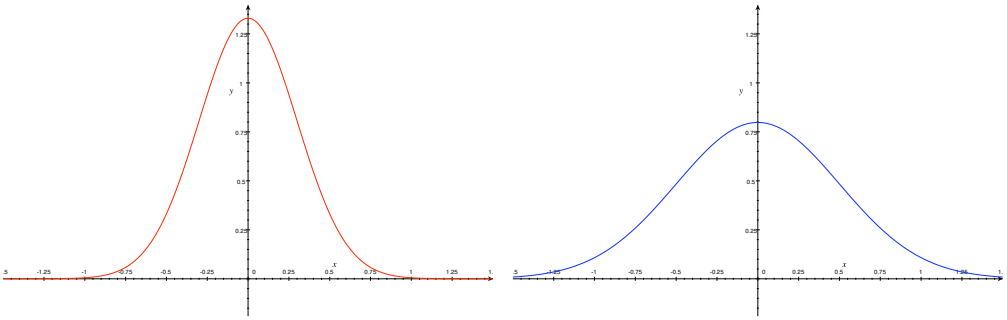
### 2.4.1 Gaussian blur

Gaussian blur is a widely used image filter. The effect of a Gaussian blur is to reduce noise and detail of an image, i.e. a low pass on an image. The name Gaussian blur comes from the

use of the normal distribution

$$G(r) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-r^2/2pi^3} \tag{2.21}$$

which is also called the Gaussian distribution. $r$ is the blur radius $r^2 = x^2 + y^2$, $\sigma$ the standard deviation or 'the amount of blurring'. Applying a Gaussian blur to a image is simply convolving the image with the Gaussian distribution.



(a) $\sigma = 0.3$ (b) $\sigma = 0.5$

Figure 2.14: Gauss function

(a) The original picture, Lena (b) Gaussian filter applied (c) Laplacian edge enchancement applied

Figure 2.15: Example of image filtering

### 2.4.2 Edge enhancement

To do edge enhancement we have chosen to use the Laplacian. The Laplacian $\bigtriangledown^2 f$ of an image $f(x, y)$ is given by:

$$\bigtriangledown^2 f = \frac{\delta^2 f}{\delta x^2} + \frac{\delta^2 f}{\delta y^2} \tag{2.22}$$

[30]. We are working on images, so we need a discrete representation. The discrete partial derivatives are:

$$\begin{aligned} \frac{\delta^2 f}{\delta x^2} &= f(x+1, y) + f(x-1, y) - 2f(x, y) \\ \frac{\delta^2 f}{\delta y^2} &= f(x, y+1) + f(x, y-1) - 2f(x, y) \end{aligned} \tag{2.23}$$

which gives us:

$$\bigtriangledown^2 f = f(x, y+1) + f(x, y-1) + f(x+1, y) + f(x-1, y) - 4f(x, y) \tag{2.24}$$

Using only these equations directly would give us the features as greyish lines on a dark background. If we add the original image back again we will get images with feature enhancement. So the final filter equation is given in Equation 2.25.

$$g(x, y) = \bigtriangledown^2 f + f(x, y) \tag{2.25}$$

### 2.4.3 Wavelet spatial filters

> "The purpose of subband filtering is of course not to just decompose and reconstruct. The goal of the game is to do some compression or processing between the decomposition and reconstruction stages." – Daubechies [25]

In Section 2.3, we introduced wavelets as a method of decomposing an image into a number of detail coefficients and approximations. By applying the backward DWT we get the original image back. If we apply filters to the decomposed results before composition it is possible to refine the end result. There are some options on which of the coefficients to filter. In Figure 2.16, these options are shown on a two-level wavelet.

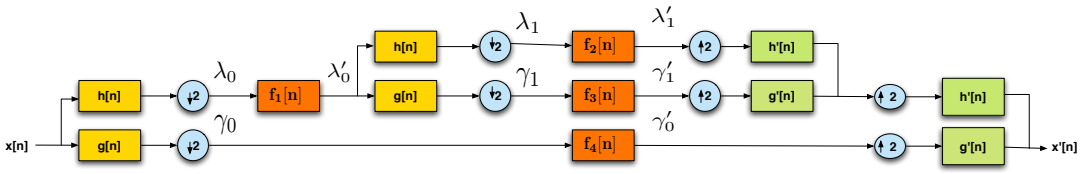



Figure 2.16: Figure showing where filtering may be applied between forward and backward DWT.

The most used option is to decompose the image entirely and then apply filter to the decomposed coefficients ($f_2$, $f_3$ and $f_4$. It is also possible to apply a filter to the approximation before further decomposing it ($f_1$). Usually we apply different kinds of filters to the low pass part $f_2$ and the high pass part ($f_3$ and $f_4$). We will now discuss a popular wavelet filter called Wavelet Shrinkage.

#### Wavelet Shrinkage (WS)

Wavelet Shrinkage is a filtering method that was first applied to denoising by Donoho [31]. The idea is that most speckle noise lies in the high frequencies of the image. In addition, the noise will usually contribute less than the features. So if we reduce the detail coefficients we can reduce the noise energy while still preserving most of the features. Since the noise is most evident in the higher frequencies, WS applies more reduction to the finer detail coefficients, and less to the coarser. In terms of levels, the low numbered levels are reduced more. The reduction method used is thresholding. Especially two thresholding methods are used much in literature, soft and hard thresholding. From [19] we have a definition of soft thresholding

$$u = S(v, t) = sign(v)(|v| - t)_+ \tag{2.26}$$

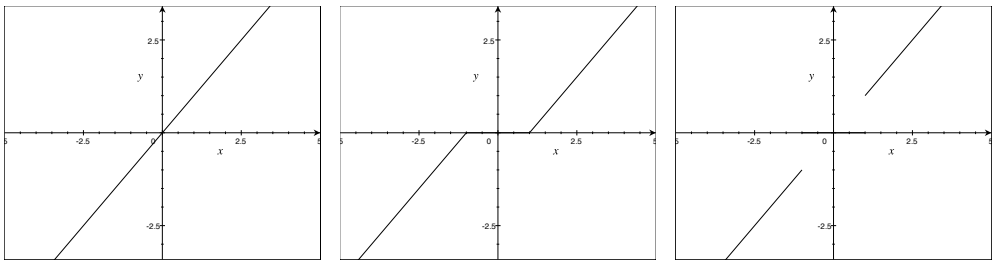
where

$$(|v| - t)_+ = \begin{cases} |v| - t & \text{if } |v| > t \\ 0 & \text{otherwise} \end{cases} \tag{2.27}$$

Hard thresholding is defined as:

$$u = H(v, t) = \begin{cases} 0 & \text{if } t > v > -t \\ v & \text{if otherwise} \end{cases} \tag{2.28}$$

The thresholding functions are shown in Figure 2.17 The difference between the two thresh-



(a) The original function (b) Soft thresholding (c) Hard thresholding

Figure 2.17: Thresholding functions

olding functions are small in practice. Usually hard thresholding contains features better, but also it contains more artefacts from the wavelet synthesis. An extreme example of these artefacts can be seen in Figure 2.18. As can be seen, these artefacts are quite similar to the artefacts produced by hard filters in the Fourier domain.

(a) The original image (b) Thresholded image

Figure 2.18: Example of too much hard thresholding

## 2.5 Previous GPU work on wavelets

The simple spatial filters has been implemented on the GPU by numerous authors. [32] lists sample implementations of a Gaussian Filter. The Laplacian can be implemented in a similar manner. Our implementation of these techniques is similar and exhibits similar performance.

As far as the authors knows, there has only been one previous attemt at implementing the DWT, presented in [33]. The method presented uses a lookup table to check which data values and kernel values should be used for each result. This table takes care of boundary conditions and it also stores information about which kernel (high or low) that should be used. One of the positive results of this design is that the decomposition and composition are very similar. Also the design allows for wavelet kernels of different size without to much re-coding. The downside to this approach is that it uses many texture reads and all but one of them are dependent. Dependent texture reads are texture reads that depend on earlier texture reads in the same kernel computation. A more detailed comparison of the previous method and our contribution will be presented in Section 3.4.

All of the methods presented in this thesis have previously been implemented on the CPU. We will hence also compare all our methods to CPU-equivalent implementations.

Chapter 3

# Novel Wavelet Techniques

*The devil is in the details*
**-Proverb**

The background theory for GPUs, Wavelets and Ultrasound were discussed in the previous chapter. In this chapter, our novel methods will be presented, including a texture addressing optimisation that we have used for both DWT and spatial filtering (Section 3.1). The main techniques developed, DWT on the GPU, is described in Section 3.2. We have also implemented spatial filters that are used between forward and backward DWT. These are described in Section 3.3. In Section 3.4, we derive a theoretical model that can be used to analyse the performance of the algorithms on the GPU.

## 3.1 Texture addressing on the GPU

One of the novel techniques developed is the texture addressing optimisation used in our filters, both for the DWT and the spatial filters. The idea itself is not new, but it has not been used in conjunction with DWT before.

In its most basic form, filters are just functions that take in an input image and outputs a transformed output image. Usually, the image filters are defined by their filter kernels. A filter kernel define how to construct one pixel in the output image. The input to the kernel is usually at least the pixel in the same position in the input image as the current output value calculated by the kernel, but also the pixels surrounding them are often used. The inputs are often called 'taps'. This kind of computation maps perfectly to the GPU; One output and several inputs. In Figure 3.1, a schematic overview of filtering process on a GPU is shown. In order to execute a filter kernel (fragment program), we will have to know the address of the corresponding pixel in the input texture. This will be achieved using the texture input coordinates. As discussed in Section 2.1.4 the texture coordinates are given as arguments to the vertices in the quad. They are then interpolated across the quad. Texture coordinate addresses in DirectX are in $[0, 1]$ regardless of the size of the source texture.

If we use a quad with texture coordinates, we can access the input pixel with same coordinates as the current output pixel with:

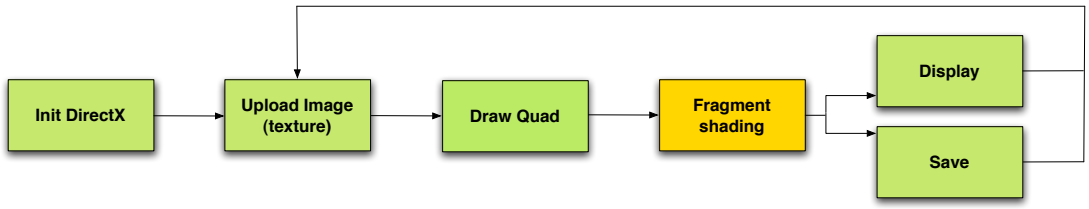
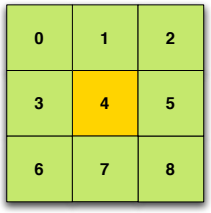
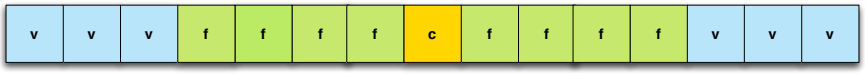



Figure 3.1: Schematic overview of the filter rendering process.




(a) A 3x3 tap




(b) A 1x15 tap

Figure 3.2: Explanation of different filter kernels.

```
float4 value = tex2D(Source, Tex);
```

We now have the address to the centre pixel. Almost all filter kernels use more taps. E.g. a simple 3x3 kernel like the one in Figure 3.2(a). To compute the addresses we will have to add or subtract the texel [1] width and height. The obvious way to do this would be to do it in the fragment shader like this: (in this example for tap 5)

```
tap[5] = tex2D(Source,float2(Tex.x + texelWidth, Tex.y))
```

Since source texture coordinates go from 0 to 1, texelWidth is $1/sourceWidth$. However, this is quite inefficient. For a picture with $1024^2$ elements, each coordinate will be computed 1 048 576 times! We can instead precompute the offsets in the vertex shader and pass them along as parameters to the fragment shader. Listing 3.1 shows a vertex shader calculating the addresses for the sample 9-tap kernel.

[1] A texel is one element in texture memory.

```
NINE_TAP_STRUCT SevenTapVertexShader(INPUT_TEX input) {
  NINE_TAP_STRUCT output = (NINE_TAP_STRUCT)0;

  output.Position = mul(input.Position, mvpMatrix);
  output.Tap0 = input.Tex + float2(0 - texelHeight, texelWidth);
  output.Tap1 = input.Tex + float2(0 - texelHeight, 0 );
  output.Tap2 = input.Tex + float2(0 - texelHeight, texelWidth);

  output.Tap3 = input.Tex + float2(0,     -texelWidth);
  output.Tap4 = input.Tex
  output.Tap5 = input.Tex + float2(0,     texelWidth);

  output.Tap6 = input.Tex + float2(0 + texelHeight, -texelWidth);
  output.Tap7 = input.Tex + float2(0 + texelHeight, 0);
  output.Tap8 = input.Tex + float2(0 + texelHeight, texelWidth);
  return output;
}
```

Listing 3.1: Nine Tap Vertex Shader

Since it is only possible to transfer a small number, typically no more than nine, texture coordinates from the vertex to the fragment shader, this method doesn't always work. In this case, some of the values can be precomputed in the vertex shader, while the rest is computed in the fragment shader. In Figure 3.2(b), a kernel that can be used for separable 15x15 kernels is shown. Here, the dots are precomputed in the vertex shader, while the crosses are computed in the fragment shader.

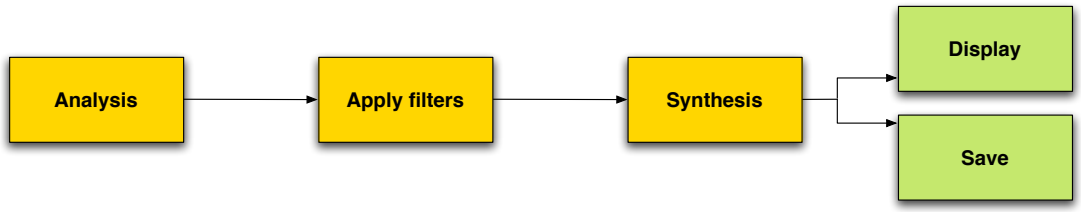
The GPU allows for several interpolation methods when accessing texture memory. This means that if you try to access an address between two elements, you can get a blend of the two elements. For our application this is rarely what we want, so we have disabled this option and chosen 'point' selection. This mode chooses the texture element closest to the address given.

## 3.2 DWT on the GPU

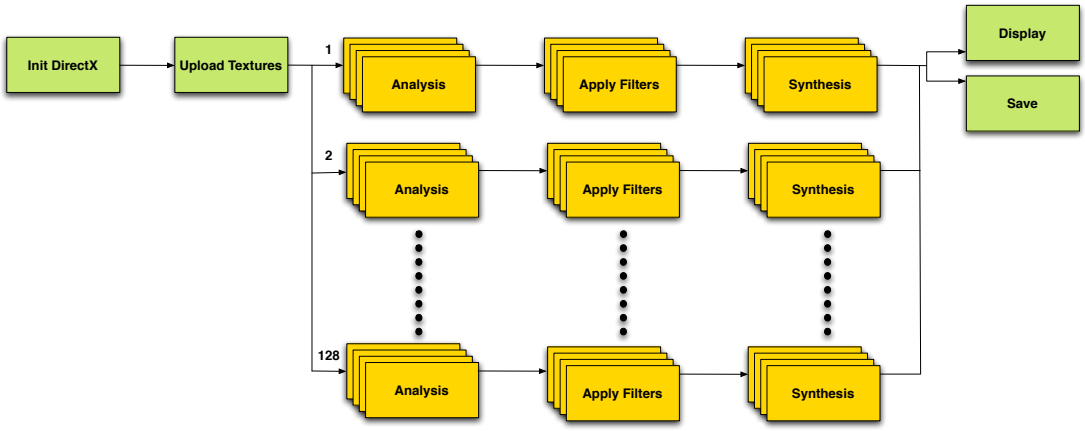
In this Section, we will discuss the main contribution of this thesis: A novel fast algorithm for computing the DWT on the GPU. The method presented here is faster than any other method, CPU or GPU, that the author knows about.

To recapitulate from Section 2.3.4, the two algorithms, or equations that the DWT consists of are: Equation 2.19 (Forward DWT)

$$\begin{aligned} \lambda(x)_{j+1} &= \sum_{k=-\infty}^{k=\infty} h(k)\lambda_j(2x+k) \\ \gamma(x)_{j+1} &= \sum_{k=-\infty}^{k=\infty} g(k)\lambda_j(2x+k+1) \end{aligned} \tag{3.1}$$




(a) CPU Version




(b) GPU Version

Figure 3.3: Schematic overview of the DWT process.

and Equation 2.20 (Backward DWT)

$$\lambda_j(x) = \sum_{k=-\infty}^{k=\infty} h'(k)\lambda_{j-1}(x+k) + \sum_{k=-\infty}^{k=\infty} g'(k)\gamma_{j-1(x+k)} \tag{3.2}$$

There are two known ways of implementing the DWT on the CPU. The most obvious is to implement the convolution in Equation 3.1 and 3.2 directly. The other is called the *lifting scheme* [34], which is an optimised version which uses approximately half of the floating-point operations used in the direct version. For the CPU, the lifting scheme is clearly the fastest algorithm. However, it uses two sub steps for each forward DWT level called *lifting* and *update*. The *update* step is dependent upon the *lifting* step. To implement this on the GPU we would have to use two render passes for each forward DWT, for a total of four passes (2 steps * 2 directions) for each level of . The direct computation can be computed in one step pr. direction, two in total for each forward DWT level. The situation is the same for the inverse transform. A render pass causes a switch in the rendering context. This gives quite a lot of overhead and should therefore be avoided. Our implementation of the DWT uses the direct computation method. The algorithm contains five phases: initialisation, four computational phases and display. The phases can be seen in Figure 3.3(b). The initialisation, uploading of textures and display phases will not be described here as they have already been covered in Chapter 2.1. Forward and Backward DWT will be discussed in Section 3.2.1 and 3.2.2 respectively.

### 3.2.1 Forward Discrete Wavelet Transform

The equation presented in the previous section has an infinite filter size, but for practical applications we will always have a finite filter. For a filter of size $K$, we can rewrite Equation 3.1 to:

$$\begin{aligned} \lambda(x)_{j-1} &= \sum_{k=0}^{k=K} h(k)\lambda(2x - \frac{K}{2} + k) \\ \gamma(x)_{j-1} &= \sum_{k=0}^{k=K} g(k)\gamma(2x + 1 - \frac{K}{2} + k) \end{aligned} \tag{3.3}$$

A straightforward implementation of this would be to compute the high and low values as shown in Listing 3.2.

```
lowResult[n] = 0;
for(int n=0;n<N;n+=2) {
  for(int k=0;k<K;k++) {
    lowResult[n/2] += source[n+k-K/2] * lowKernel[k]
  }
}

highResult[n] = 0;
for(int n=0;n<N;n+=2) {
  for(int k=0;k<K;k++) {
    highResult[n/2] += source[n+k-K/2+1] * highKernel[k]
  }
}
```

Listing 3.2: Forward DWT Pseudocode using two for-loops

For the CPU, this algorithm will run fast, and it is possible to implement it on the GPU using two passes. By drawing a quad with size $\frac{N}{2}$ results in an equal number of fragment executed. The inner loops can be directly implemented as fragment programs. However, this approach has several drawbacks. First, it needs two render passes, and as we discussed this will hinder performance. Secondly, it doesn't take into account boundary conditions. This happens when $n < \frac{K}{2}$. In the previous implementation of DWT on the GPU, both of these problems were solved by combining the for-loops and using a lookup table. The essence of their approach can be seen in Listing 3.3.

```
for(int n=0;n<N;n++) {
  if(n < N) {
    allResults[n] = 0;
    for(int k=0;k<K;k++) {
      allResults[n] += source[n+k-K/2] * lowKernel[k]
    }
  } else {
    allResults[n] = 0;
    for(int k=0;k<K;k++) {
      allResults[n] += source[n+k-K/2+1] * highKernel[k]
    }
  }
}
```

Listing 3.3: Forward DWT Pseudocode, old version

The if-else statement has been written for clarity, it is not in their code. Instead, a lookup table is used. This approach has some performance problems which we will discuss later, we have chosen another route. Just like the previous implementation, we have chosen to rewrite Listing 3.2 so it just consists of one for-loop instead of two. Our version is shown in Listing 3.4

```
for(int n=0;n<N;n+=2) {
  lowResult[n] = 0;
  highResult[n] = 0;
  for(int k=0;k<K;k++) {
    lowResult[n/2] += source[n+k-K/2] * lowKernel[k]
    highResult[n/2] += source[n+k-K/2+1] * highKernel[k]
  }
}
```

Listing 3.4: Forward DWT Pseudocode using one for-loop

This solves the problem of two render passes, but another problem arises because a fragment shader can normally only output one value (the pixel that should be displayed on the screen). This is solved by using the advanced technique Multiple Render Target (MRT) which makes it possible to output up to eight values at the time. Using MRT itself is quite easy, but very few GPGPU algorithms have used it so far because it is fairly new. One added benefit of computing two values in one fragment shader is that we will need less texture lookups (memory fetches). As we shall discuss later, avoiding texture lookups is critical to performance.

The second challenge is boundary conditions. The usual way to solve this for DWT is to mirror the source so that lookups that occur outside the normal source bounds yield a mirrored source. The previous implementation stated this as one of the most important reasons to use a lookup table. In our implementation, we have chosen instead to use the build in support for mirroring textures in the hardware.

The refactoring of the code from Listing 3.2 to Listing 3.4 might not seem intuitive at first, but as we have shown, it turns out that it is a smart way to do the calculations considering the hardware limitations and abilities of the GPU. We will now discuss the algorithm in greater detail and try to explain some implementation intricacies that arise.

As the reader might have noticed, we have so far only discussed the 1D (horizontal) case. We will continue to do so, because as we explained in Section 2.3, the DWT is separable and the vertical forward transform is very similar to the horizontal case. We will leave the discussion of the vertical version to the end of this section.

As discussed in Section 2.1.4, the common way to execute GPGPU-programs is to draw a quad with the same size as the output. In forward DWT, the output is the same size as the input. Half of it low pass result, the other half high pass. In our algorithm, we output the low and high results into two different textures using the Multiple Render Targets (MRT) technique. The output, and the quad we draw is really half the size of the input: $[\frac{w}{2}, h]$. This causes the fragment shader to be called $\frac{wh}{2} = \frac{n}{2}$ times, which is the same number of times the outer loop in Listing 3.4 is called. If we let *lowResult* be the first texture target, and *highResult* the second, the inner part in the listing is what we need to implement in the fragment program. It can be very similar, but there are some technicalities involved in addressing the source texture. Figure 3.4 and 3.5 shows the addressing used. The main point is letting the source texture address run from 0 to 1 as usual, but in increments of 2 per drawn pixel. The reason we need to do this is because we compute two values for each drawn pixel, both high and low value. This corresponds to the $+= 2$ part in Listing 3.4. In Figure **??**, the calculation of one pair of low/high values is depicted.

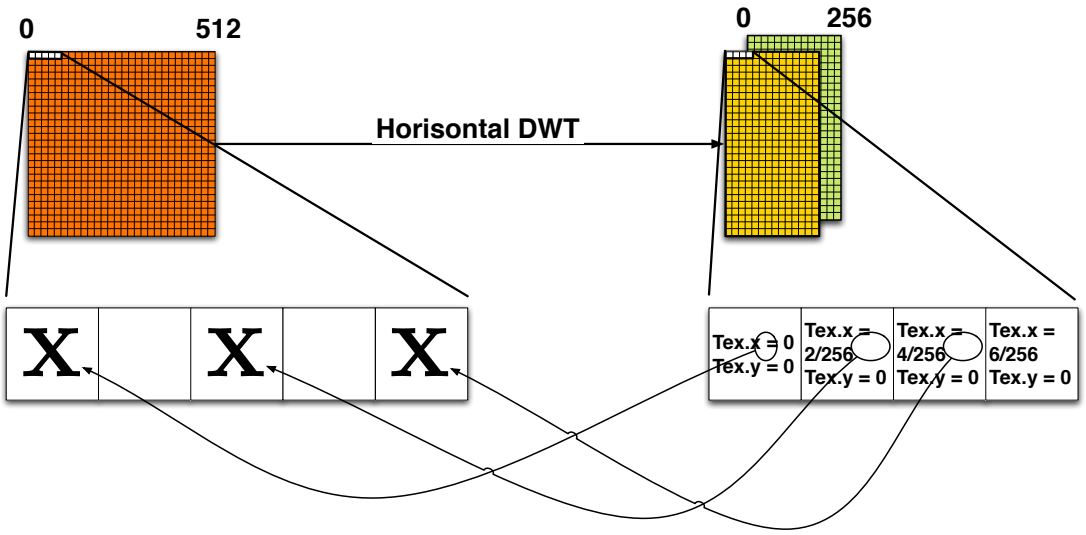



Figure 3.4: Forward DWT addressing

The vertical forward transform will be carried out on both results from the horizontal forward transform, both high and low pass. First, we bind the high result as the source and draw a quad with $\frac{w}{2}, \frac{h}{2}$. This will produce HH and HL. Then we bind the low result and calculate LH and LL.

### 3.2.2 Backward Discrete Wavelet Transform

Unfortunately, the algorithm for the backward DWT is different from the forward transform. This means that the method has to be developed independently. The previous GPU implementation managed to keep the GPU implementation similar by redefining the lookup table. But since we are trying to get the maximum performance from the hardware our solution has to be different.

We rewrite equation 3.2 to:

$$\lambda_j(x) = \sum_{k=0}^{k=K} h'(k)\lambda_{j-1}(x + k - \frac{K}{2}) + \sum_{k=0}^{k=K} g'(k)\gamma_{j-1}(x + k - \frac{K}{2}) \tag{3.4}$$

As we can see, $\lambda_{j-1}(x)$ and $\gamma_{j-1}(x)$ is the upsampled and boundary extended low-pass and high-pass signal respectively. In Figure 3.6, the backward transform of a single element using a kernel with five elements is shown. As is shown, not all high and low values are used from each filter kernel as it may seem from equation 3.4. Instead we create an 'interleaved kernel' with every second parameter from the high and low kernels. The computation of element $r_2$ is shown again in Figure 3.7(a), this time with the interleaved kernel. Every even calculation will use this version of the interleaved kernel. The odd calculations will use the 'inversed' version, swapping high and low elements. This is shown in Figure 3.7(b) for element $r_3$. We name the new interleaved kernels *even kernel* $\hat{h}$ and *odd kernel* $\hat{g}$ and define them as:

$$\hat{h}(x) = \begin{cases} h'(x) & \text{if } x \text{ is odd} \\ g'(x) & \text{otherwise} \end{cases} \tag{3.5}$$

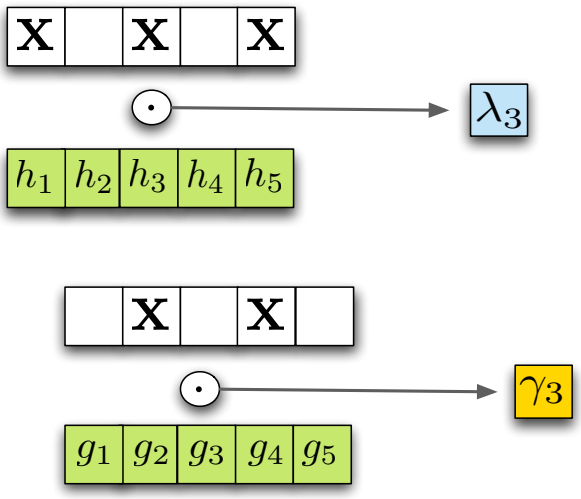



Figure 3.5: Calculation of a single high/low pair

$$\hat{g}(x) = \begin{cases} h'(x) & \text{if } x \text{ is even} \\ g'(x) & \text{otherwise} \end{cases} \tag{3.6}$$

The low pass kernel elements of the new kernels has to be used with results from the low pass forward transform, and high pass kernel elements with the results from high pass forward transform. This can also be seen in Figure 3.7. We define selectors $z_{even}$ and $z_{odd}$ to be:

$$z_{even}(x) = \begin{cases} \lambda(x) & \text{if } x \text{ is odd} \\ \gamma(x) & \text{otherwise} \end{cases} \tag{3.7}$$

$$z_{odd}(x) = \begin{cases} \lambda(x) & \text{if } x \text{ is even} \\ \gamma(x) & \text{otherwise} \end{cases} \tag{3.8}$$

We can now write a pseudocode program to compute the backward DWT.

```
for(int n=0;n<N;n++) {
  if(isEven(n)) {
    for(int k=0;k<K;k++) {
      result[n] += zEven[n+k-K/2] * hHat[k];
    }
  } else {// odd
    for(int k=0;k<K;k++) {
      result[n] += zOdd[n+k-K/2] * gHat[k];
    }
  }
}
```

Listing 3.5: "DWT Synthesis Pseudocode"

As we can see from Listing 3.5 and the discussion above we have to act differently on even and odd results. The straightforward way would be to use a conditional statement, but as discussed in Section 2.1.4 conditional statements can be very expensive on the GPU. On the

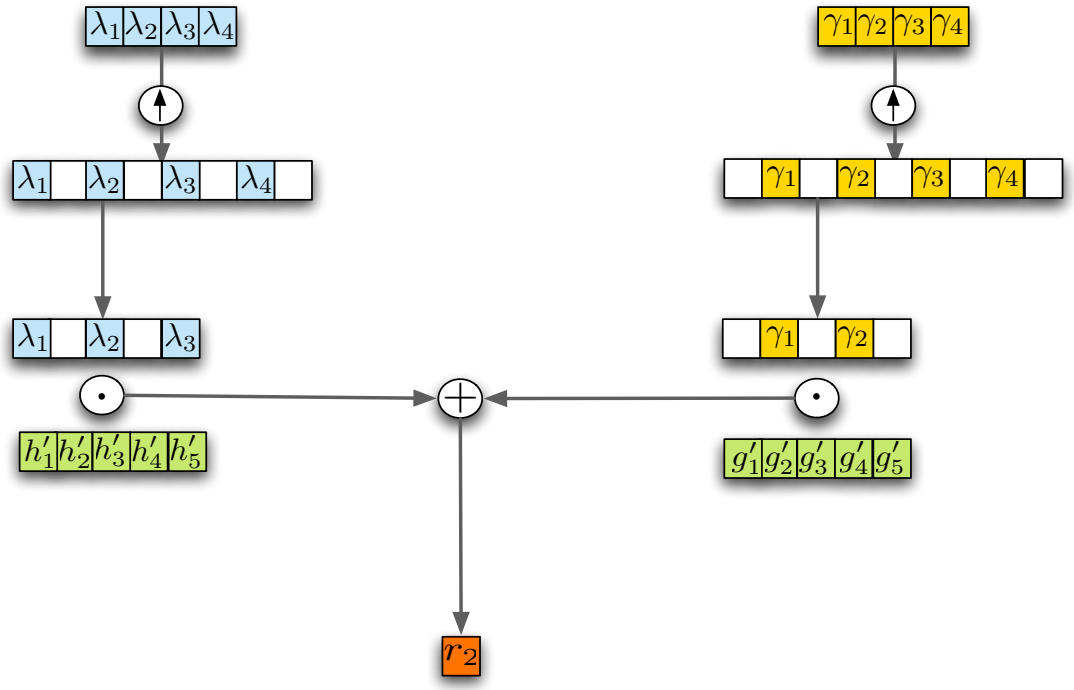
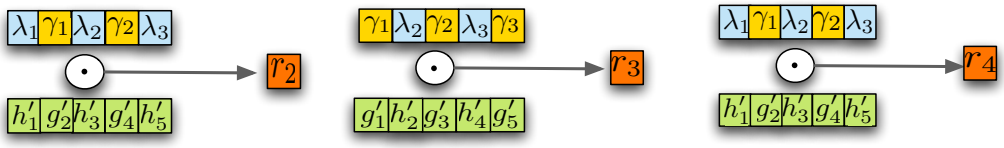



Figure 3.6: Performing backward DWT on element 2. The subscripts indicate position.




(a) Figure 3.6 simplified (b) The two next computations

Figure 3.7: Simplified view of the backward transform

CPU the situation would be similar. The conditional branching in the loop could be avoided by using static branch resolution, or 'moving the conditional up the pipeline'. So instead of having one loop with conditional branching we could create two loops with no branching. We can do the same on the GPU. By executing a different shader based on the result currently calculated, even or odd, we can achieve this. By drawing quads that cover only the parts of the result that we would like to execute we can choose the shader that should be excuted. We draw quads covering the even parts when we would like to execute the even fragment program and quads covering the odd parts when we would like to execute the odd fragment program. Figure 3.8 shows how this will look for the horizontal case. The even part is first calculated and then the odd. In total, the whole result is calculated.

The fragment programs are very similar to the inner loops of Listing 3.5.

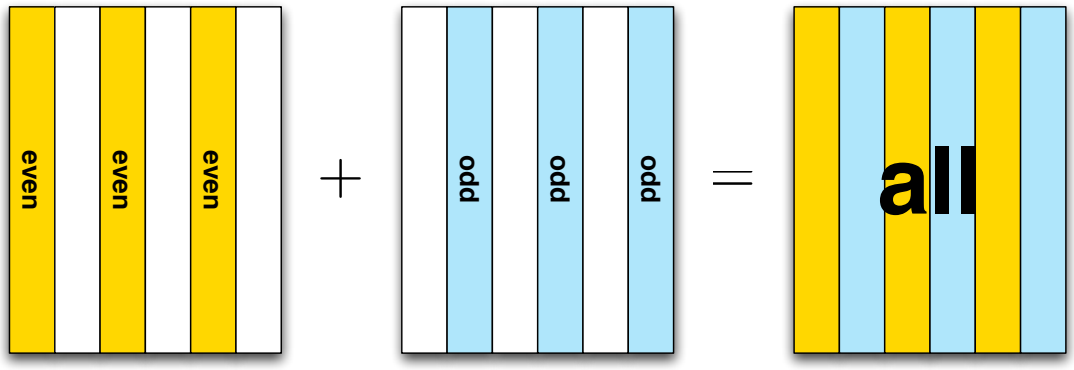



Figure 3.8: Static branch resolution by drawing quads.

### 3.2.3 Single channel version

In medical applications like ultrasound, the data usually consists of single channel values, and doesn't use the entire RGBA scale. The general DWT described in the previous sections doesn't efficiently compute these kind of single valued data. Actually, it will waste almost 1/4 of its capabilities. There exists at least two different approaches to this problem. The first and perhaps most obvious idea is to pack the data four and four, use the general DWT and then unpack it again. This is the approach taken by the various SSE CPU implementations of the DWT. SSE is a special SIMD instruction set for the Intel x86 which can operate on 4 data values simultaneously. This implementation becomes complicated because we want to do 2D DWT, and if the data values have to be unpacked and packed again for each of the two directions, the overhead becomes larger than the efficiency gained. A comparison of different SIMD implementations can be found in [35].

We have chosen another route. In our implementation, we use the additional channels to avoid the use of multiple render targets. We write to two channels in the same texture during horizontal forward transform and then to four channels during the vertical forward transform. Obviously, this doesn't use all the arithmetic capability of the GPU, but we benefit greatly from additional memory coherence. In addition, when the forward DWT is done, we will have all our data in one texture where it is easy to apply the filters to all the DWT results in a single render pass. This implementation is also very similar to the general RGBA implementation, and the surrounding framework is almost identical.

## 3.3 Spatial filters

Between the forward and backward transforms we apply spatial filters to the high and low pass data. In this section, we will discuss the filters developed during this thesis. Some of the filters have already been explored in literature and our contribution is merely to use them in a DWT setting. We have opted to include them for completeness. This includes Gaussian and Laplacian filtering discussed in Section 3.3.1 and 3.3.2. Soft and Hard thresholding however, have not, to the authors knowledge, been implemented on the GPU. We develop special versions of these filters tailored to the hardware in Section 3.3.3 and 3.3.4.

### 3.3.1 Gaussian blur

As mentioned in Section 2.5 the Gaussian blur and the Laplacian has already been implemented by previous authors. Our contribution is to create the filtering framework surrounding them, and this section is included mostly for completeness since these filters are used in conjunction with the Discrete Wavelet Transform.

In Section 2.4.1, we discussed the Gaussian blur. A practical filter is shown in Figure 3.9(a). It has $\sigma = 1$ and has been cut for radii larger than 2. The values at larger radii are so small that they can be neglected. By convolving this filter with the image, we get a blurred image. The Gaussian is separable, which means that we can do a horizontal and then a vertical blur. This makes it possible to compute it in $O(nMN) + O(mMN)$ instead of $O(mnMN)$ where $m, n$ are kernel dimensions and $M, N$ are image dimensions. In Listing 3.6, the fragment code for horizontal filtering is shown. For simplicity we have shown it without vertex shader optimisation discussed in the previous section.

```
float blurWeight[] = {0.0256,0.0952,0.1502,0.0952,0.0256};

PS_OUT GaussianBlurPS(INPUT_TEX input) {
  PS_OUT output = (PS_OUT)0;

  // Convolute
  for(int i=0;i<6;i++)
    output.rgb += tex2D(SourceSampler, float2(input.Center.x + (i-2)*
        imageWidth, input.Center.y) *
        blurWeight[i];
  output.a = 1; // we don't blur alpha
  return output;
}
```

Listing 3.6: Fragment shader code for Gaussian filtering

| | | | | |
|---|---|---|---|---|
| 0.0037 | 0.0146 | 0.0256 | 0.0146 | 0.0037 |
| 0.0146 | 0.0586 | 0.0952 | 0.0586 | 0.0146 |
| 0.0256 | 0.0952 | 0.1502 | 0.0952 | 0.0256 |
| 0.0146 | 0.0586 | 0.0952 | 0.0586 | 0.0146 |
| 0.0037 | 0.0146 | 0.0256 | 0.0146 | 0.0037 |

(a) 5x5 Gauss filter. $\sigma$=1

| | | |
|---|---|---|
| 0 | 0.1 | 0 |
| 0.1 | -0.4 | 0.1 |
| 0 | 0.1 | 0 |

(b) Laplacian as defined in Equation 2.24

| | | |
|---|---|---|
| 0.1 | 0.1 | 0.1 |
| 0.1 | -0.8 | 0.1 |
| 0.1 | 0.1 | 0.1 |

(c) A diagonal Laplacian

Figure 3.9: Filter kernels

### 3.3.2 Edge enhancement

In Section 2.4.2, we discussed the Laplacian and derived a discrete formula. Two filters for practical use are given in Figure 3.9. 3.9(a) is the one given in Equation 2.24. Figure 3.9(b) is

the diagonal version. We use these filters in the same way as we used the Gaussian filters in Section 3.3.1.

### 3.3.3 Soft thresholding

In Section 2.4.3, soft thresholding was discussed. On the CPU it has a quite simple and direct implementation. Listing 3.7 shows it. However this does not fit especially good on the GPU.

```
double softThreshold(double input, double threshold) {
  if(abs(input) < threshold) return 0.0;
  if(input < 0.0) return input + threshold;
  return input - threshold;
}
```

Listing 3.7: Soft Thresholding in C

The reason for this, is that this code uses branching, which unfortunately can be quite expensive on the GPU [1]. Additionally, branching is one of the few operations that are not SIMD on the GPU. Instead we have developed a method which uses the build-in intrinsics and avoids branching. We use the following intrinsics:

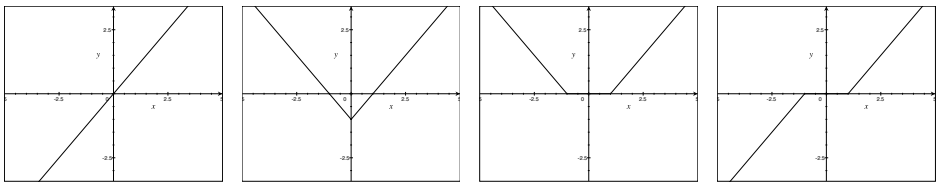
- *sign(val)*. Returns the sign of val. 0 if val is 0, -1 if val $< 0$ and 1 if val $> 0$.
- *max(valOne, valTwo)*. Returns the maximum of valOne and valTwo.
- *abs(val)*. Returns the absolute value of val.

All of these intrinsics work on vectors of up to 4 values and they are usually computed in one cycle. As an example, *max([1,2,3,4], [0,0,5,5])* will be *[1,2,5,5]*. If the function would only contain positive values we could have used $threshold(x) = max(0, val - T)$ to get the soft threshold, where $T$ is the threshold value. Since this function is going to be used on wavelet coefficients we have to generalise it to negative values as well. The function

```
float softThreshold(float x) {
  return sign(x) * max(0, abs(val) - T)
}
```

Listing 3.8: Soft Thresholding in HLSL

will accomplish that. In Figure 3.10, we illustrate the different parts of this function.



(a) The original function $f(x) = x$ (b) $abs(x) - T$ (c) $max(0, abs(x) - T)$ (d) Soft thresholding $sign(x) * max(0, abs(x) - T)$

Figure 3.10: Soft thresholding function explained

### 3.3.4 Hard thresholding

Hard thresholding is very similar to soft thresholding as can be seen in the pseudo-code in Listing 3.9.

```
double hardThreshold(double input, double threshold) {
  if(abs(input) < threshold) return 0.0;
  return input;
}
```

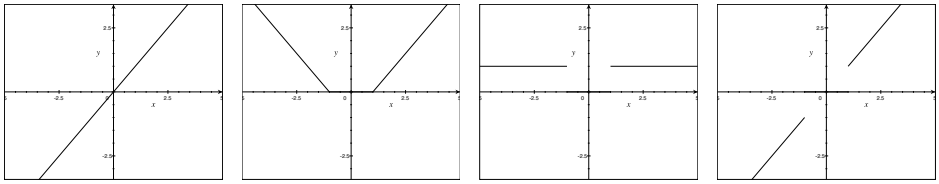
Listing 3.9: Hard Thresholding in C

Again we try to avoid the branching when implementing the threshold on the GPU. We use the same operators as in soft thresholding, but the function it self is a bit different.

```
float hardThreshold(float x) {
  return sign(max(abs(x)-T, 0) * x
}
```

Listing 3.10: Hard Thresholding in HLSL

The explanation for this function can be found in Figure 3.11.



(a) The original function $f(x) = x$ (b) $max(0, abs(x) - T)$ (c) $sign(max(0, abs(x) - T)))$ (d) Hard thresholding $sign(max(0, abs(x) - T)) * x$

Figure 3.11: Hard thresholding function explained

## 3.4 Theoretic GPU Performance model

In this section, we will formulate a model for the computation on the GPU. It is based on theory from [36]. The GPU is complicated piece of hardware, but we will keep our model very simple. One basic model of computation time breaks the computation time ($T_{total}$) into overhead ($T_{overhead}$), memory transfer ($T_{upload}$) and computation time ($T_{computation}$). Such a model is formulated in Equation 3.9.

$$T_{total} = T_{overhead} + T_{upload} + T_{computation} \tag{3.9}$$

Considering Figure 3.12, which shows the architecture of a PC with a GPU, we can break down the equation as follows:

The overhead consists mainly of the time spent in calling various functions from the DirectX API. It is assumed to be independent of $N_{elements}$, the number of elements computed.

$$T_{overhead} = constant \tag{3.10}$$

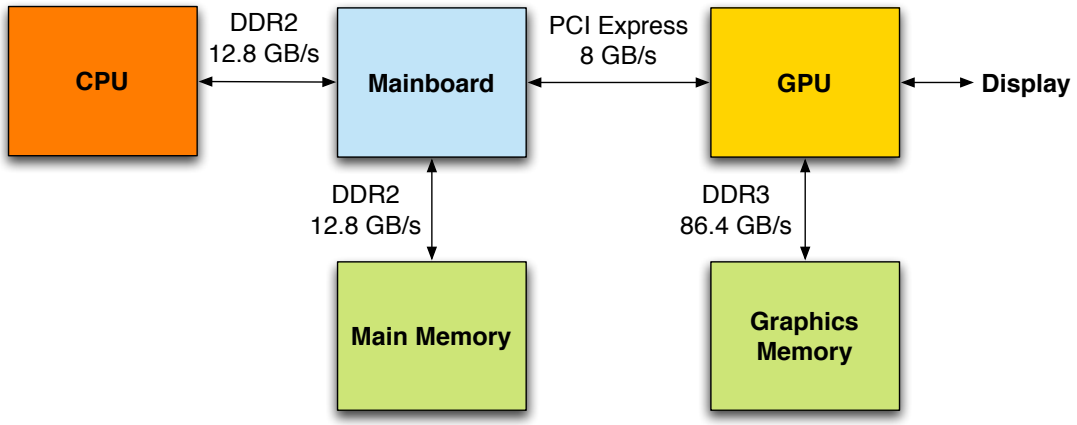



Figure 3.12: A model of the system, including GPU, CPU and memory

Each element consists of four subelements: red, green, blue and alpha, and when an element is transferred from main memory to the GPU, each subelement is represented by one byte. In total four bytes are transferred per upload element ($N_{uploadbyte}$). The memory transfer time is given by the following equation:

$$\begin{aligned} T_{upload} &= \frac{N_{uploadbyte}}{B_{cpu}} \\ N_{uploadbyte} &= 4N_{element} \end{aligned} \tag{3.11}$$

$B_{cpu}$ being the bandwidth between CPU main memory and GPU. The DWT requires floating point precision for correct computation, so when we do further calculations, we represent the elements as 16 bit floating point numbers. Thus, each texture memory element ($N_{texturebyte}$) occupies eight bytes. The GPU is able to do arithmetic operations and texture fetches simultaneously, which gives the following approximation for the computation time on the GPU:

$$\begin{aligned} T_{computation} &= max(T_{gpuarithmetic}, T_{gpumemtransfer}) \\ T_{gpumemtransfer} &= \frac{N_{texturebyte} * O_{texture}}{B_{gpu}} \\ T_{gpuarithmetic} &= \frac{4N_{element} * O_{arithmetic}}{O_{arithmetic}/s} \\ N_{texturebyte} &= 8N_{element} \end{aligned} \tag{3.12}$$

$O_{texture}$ and $O_{arithmetic}$ is the number of texture and arithmetic operations respectively. $B_{gpu}$ is the memory bandwidth between the GPU memory and the GPU, $O_{arithmetic}/s$ is the number of arithmetic operations the GPU can calculate per second. The arithmetic operations has to be multiplied by four because each element consists of four subelements. The execution time ratio ($R_{at}$) between arithmetic and texture operations is defined as follows:

$$R_{at} = \frac{4N_{element}/O_{arithmetic}/s}{N_{texturebyte}/B_{gpu}} \tag{3.13}$$

This ratio tells us how many artithmetic operations we can perform per texture operation before the arithmetic operations are the dominating factor.

### 3.4.1 Example graphic card

In this Section, we will use the equations derived in the previous section to analyse one example graphic card, the GeForce 8800 GTX. This is the card we later shall use for evaluation. The cards basic properties are listed in Table 3.1.

| Property | GeForce 8800 |
|---|---|
| Graphics Bus Speed | 86.4GB/s |
| Memory (MB) | 768 |
| Core Clock (MHz) | 575 |
| Shader Clock (MHz) | 1350 |
| Stream Processors | 128 |
| Memory Interface | DDR3, 684bits |
| Instructions per clock | 2 |

Table 3.1: GeForce 8800 GTX performance numbers

Estimating the exact number of floating point operations per second (GFLOPS) on a GPU can be difficult. We have used the following computation [2]:

$$575Mhz * 128processors * 2flops/instruction * 2instructions/clock = 332GFLOPS$$

Using the Microsoft `fxc` tool, which outputs assembly instructions, we have measured that our programs consist of approximately 70% Multiply-Add (*mad*) arithmetic operations The rest of the arithmetic operations are mainly declarations and *mov* operations. *mad* operates at 2 flops/instruction. The other arithmetic operations operates at 1-4 flops/instruction, which makes an approximation of 2 flops/instruction a reasonable estimate.

The transfers between main memory and GPU are limited by the PCI-express bus which has a maximum throughput of ~8GB/s. The GPU memory is DDR3 with a theoretical bandwidth of ~86.4GB/s.

We also used the `fxc` tool to estimate the number of operations needed by our programs. These values are listed in Table 3.2. It might seem strange that the forward transform uses

| | Forward DWT | | Backward DWT | |
|---|---|---|---|---|
| Algorithm \ Instructions | Arithmetic | Texture | Arithmetic | Texture |
| gpu-mono | 19 | 5 | 10 | 5 |
| gpu-colour | 20 | 5 | 10 | 9 |
| gpu-previous | 140 | 28 | 140 | 28 |

Table 3.2: The number of fragment instructions used by the GPU implementations.

twice as many arithmetic operations as the the backward transform. The reason is that forward DWT uses so many texture addresses that all can't be precalculated in the vertex shader, as discussed in Section 3.1. So it uses a couple of extra operations to calculate these addresses. Another peculiar result is that the previous implementation uses the exact same number of

[2] from [37]

operations in both backward and forward DWT. The reason is that they use a very similar fragment program for both executions as discussed in Section 3.2.2. The differences between colour and mono version will be further discussed in the evaluation chapter, in this analysis we will use the colour version. Combining the forward and backward transform, but excluding the filters, our algorithm uses 14 texture operations and 30 arithmetic operations.

In Table 3.3, the variables and calculations from the above discussion are summarised for $N_{elements} = 512^2$. There are a number of interesting results in this table. The first is the

| Variable | Value |
|---|---|
| $N_{element}$ | $512^2$ |
| $B_{cpu}$ | ∼8GB/s |
| $B_{gpu}$ | ∼86.4GB/s |
| $O_{arithmetic}$ | 30 |
| $O_{texture}$ | 14 |
| $O_{arithmetic}/s$ | 332 GFLOPS |
| $T_{gpuarithmetic}$ | 0.09ms |
| $T_{gpumemtransfer}$ | 0.37ms |
| $R_{at}$ | 8.3 |
| $T_{computation}$ | 0.37ms |
| $T_{upload}$ | 0.13ms |
| $T_{total}$ | 0.5ms |

Table 3.3: Summary of variables, GeForce 8800

high number for $R_{at}$, ∼8. A program can have eight times as many arithmetic operations as texture operations. The real number is probably a bit higher because we have not considered latency issues. Our program (*gpu-colour*) has a ratio of ∼2, and we can see that $T_{gpumemtransfer}$ dominates the GPU computation time.

Further, the table shows that approximately 25% of the time is spent transferring data to the GPU. This time is a pure extra cost compared to calculating the DWT on the CPU, and it is not possible to change it without changing underlying hardware.

### 3.4.2 Comparison with previous implementations

From Table 3.2 it is quite evident that our implementation uses less fragment operations than the previous GPU implementation. We use 30 arithmetic and 9 texture operations, while the previous implementation uses 280 arithmetic and 56 texture operations. The largest problem is the 56 texture operations, by the time these are fetched, 450 arithmetic operations could have been performed. The method uses texture operations for table lookup,kernel lookup and data lookup. The new implementation only need data lookup, and it combines lookups to compute several values. Another problem of the previous implementation, that doesn't manifest itself in the tables, is that it uses dependent texture reads. Dependent texture reads are texture reads that depend on previous texture reads. They effectively stall the pipeline until the read depended upon is completed. Our method uses no such reads. We therefore expect a speedup over the previous gpu implementation.

It is difficult to directly compare the GPU with a CPU implementation because of the great

difference in both hardware and algorithms used. An approximation is given in Equation 3.14.

$$\begin{aligned} speedup &= \frac{T_{cputime}}{T_{gputime}} \\ &= \frac{T_{cputime}}{T_{overhead} + T_{upload} + T_{computation}} \end{aligned} \tag{3.14}$$

where

$$T_{cputime} = \frac{N_{elements} \cdot O_{cpu}}{O_{cpu}/s}$$

Where $T_{gputime}$ and $T_{cputime}$ is the time to compute the DWT on the GPU and the CPU respectively. In our case, $T_{gputime}$ is the same as $T_{total}$ which we defined earlier. $O_{cpu}$ is the number of operations needed to compute the DWT on the CPU, and $O_{cpu}/s$ is the number of operations per second on the CPU. In [1] $O_{cpu}/s$ is approximated to 25 GFLOPS. We have estimated $O_{cpu}$ to 240 operations by using Microsoft Visual Studio.

$$T_{cputime} = \frac{512^2 elements \cdot 240 operations}{25e9 operations/s} = 3.15ms$$

This gives an estimated speedup of:

$$speedup = \frac{3.15}{0.5} = 6.3$$

It should be stressed that this calculation is only a rough approximation. The next chapter will show the actual results. We expect both $T_{cputime}$ and $T_{cputime}$ to be underestimated since we have ignored overhead and latency issues.

In this chapter, we have discussed how to implement different ultrasound filters on the GPU. The most important and complex was the wavelet filter. In the next chapter, we will benchmark and compare some of the different approaches that were discussed in this chapter.

Chapter 4

# Results

*Not to be absolutely certain is, I think, one of the essential things in rationality.*

**-Bertrand Russell**

In this chapter, the performance and visual quality of the proposed algorithms are described. In Section 4.1, the test setup is described. Performance benchmarking are given in Section 4.2 and visual quality, is analysed in Section **??**. The main results are summarised in Section 4.3.

## 4.1 Testing environment

Our testbed includes two different GPUs from two different GPU generations. Both GPUs were tested on an Intel(R) Pentium(R) 4 CPU 3.20GHz with 16KB L1 cache, 2MB L2 cache and 1GB DDR2 RAM installed. The graphics cards used were NVIDIA GeForce 7600 GS and NVIDIA GeForce 8800 GTX. Their properties are listed in Table 4.1.

The DWT filtering CPU program was tested on an Intel Core 2 Duo 2 Ghz (T7200) with 2 GB of DDR2 memory. This processor has 4 MB of L2 cache.

All the programs were compiled using Visual Studio 2005 (8.0.50727.762) and the following compile time optimisation options: **/O2** (fast code) **,/GL** (whole program optimisation), **/OPT:REF** (eliminate unreferenced data) and **/OPT:ICF** (remove redundant comdats). For the GPU programs DirectX9 April 2007 Retail version was used. The OS was Windows XP SP2, with visual settings set to 'for performance'.

| Property | GeForce 8800 | GeForce 7600GS |
|---|---|---|
| Graphics Bus Technology | PCI Express 16x | PCI Express 16x |
| Memory (MB) | 768 | 256 |
| Core Clock (MHz) | 575 | 400 |
| Shader Clock (MHz) | 1350 | NA |
| Fragment Processors [a] | NA | 12.0 |
| Vertex Processors [a] | NA | 5.0 |
| Stream Processors [a] | 128 | NA |
| Memory Interface (bits) | 684 | 128 |
| Memory Bandwidth (GB/sec) | 86.4 | 22.4 |
| Fill Rate (Billion pixels/sec) | 36.8 | 6.7 |
| Drivers used | 6.14.10.979 | Forceware 93.71 |
| Transistors (millions) | 681 | 178 |

[a] GeForce 7600 has specific pixel and vertex processors, while the 8800 can dynamically assign processors for both tasks.

Table 4.1: The GPU specifications used in the test.

## 4.2 Benchmarking

This section will compare different implementations of the DWT. All of the programs compute one analysis in three levels, Wavelet Shrinkage and a three level synthesis. For the GPU programs, the time includes both uploading the data to the GPU and processing it. The following programs were tested:

- *gpu-colour-32* GPU DWT as discussed in section 3.2. 32-bits
- *gpu-colour-16* Same as gpu-colour-32 but with 16 bits precision.
- *gpu-mono-32* GPU DWT on single channel data as discussed in section 3.2.3. 32-bits precision.
- *gpu-mono-16* Same as gpu-mono-32, but with 16 bits precision.
- *previous-gpu-32* The program from [33]. It works on colour data with 32 bits precision.
- *cpu-mono-32-single* A SIMD-optimised CPU implementation. It works on singel channel data.
- *cpu-mono-32-dual* Two cpu-mono-32-single programs running simultaneously. This is a crude way to utilise the dual core, but as we shall see it works surprisingly well.
- *gpu-*-*-7600gs* Same as the gpu-* programs, but tested on a GeForce 7600GS instead of the GeForce 8800 GTX.

The GPU supports both 16 bits and 32 bits of precision. The CPU, on the other hand doesn't support 16 bits, 32 bits is the lowest precision it supports. So one could argue that for a fair comparison only the 32 bits versions should be compared. However, the extra precision is unnecessary for the DWT, as the end results are identical for all practical purposes. [1] For this

[1] See section **??** for a comparison of 16 and 32 bits

reason their results are compared directly to the 32 bits CPU versions as well as our own 32 bits GPU version.

### 4.2.1 Raw processing

In Table 4.2, the number of elements each algorithm can process pr micro second are compared. Figure 4.3 shows the speedups of the algorithms relative to the CPU version.

| Algorithm \ N | $128^2$ | $256^2$ | $512^2$ | $1024^2$ | $2048^2$ |
|---|---|---|---|---|---|
| gpu-mono-16 | 29.2 | 93.7 | 255.5 | 425.0 | 394.3 |
| gpu-mono-32 | 29.5 | 92.7 | 255.5 | 321.2 | 273.8 |
| gpu-colour-16 | 41.4 | 138.9 | **340.1** | **435.7** | **528.3** |
| gpu-colour-32 | 41.3 | **139.6** | 326.1 | 435.6 | 441.8 |
| previous-gpu-32 | **102.1** | 129.1 | 116.3 | 121.1 | 79.5 |
| cpu-mono-32-single | 17.7 | 17.7 | 18.1 | 17.7 | 17.7 |
| cpu-mono-32-dual | 35.0 | 35.0 | 35.0 | 35.0 | 35.0 |

Table 4.2: Performance comparison of different wavelet algorithms, measuring the number of elements processed pr. micro second. The bold faced values indicate the largest number of elements processed.

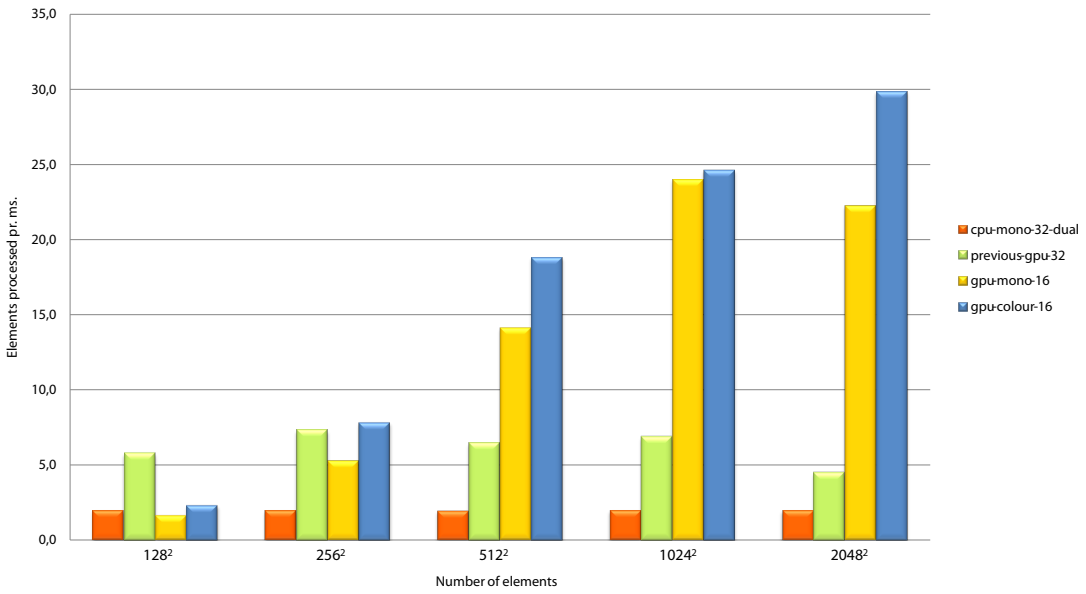



Figure 4.1: The number of elements processed by the different methods pr ms, calculated for different algorithms and data sizes.

Except for the smallest data size ($128^2$) the methods developed in this thesis are superior to the previous methods. Both the previously developed GPU method and the SIMD-optimised

| Algorithm \ N | $128^2$ | $256^2$ | $512^2$ | $1024^2$ | $2048^2$ |
|---|---|---|---|---|---|
| gpu-mono-16 | 1.6 | 5.3 | 14.1 | 24.0 | 22.3 |
| gpu-mono-32 | 1.7 | 5.2 | 14.1 | 18.1 | 15.5 |
| gpu-colour-16 | 2.3 | 7.8 | **18.8** | **24.6** | **29.8** |
| gpu-colour-32 | 2.3 | **7.9** | 18.0 | 24.6 | 24.9 |
| previous-gpu-32 | **5.8** | 7.3 | 6.4 | 6.8 | 4.5 |
| cpu-mono-32-single | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| cpu-mono-32-dual | 2.0 | 2.0 | 1.9 | 2.0 | 2.0 |

Table 4.3: Comparison of different wavelet algorithms, showing the speedups of the methods relative to the single-CPU version

CPU method. For the largest size the best method is almost 30 times faster than the single CPU version. The most important sizes are $256^2$ and $512^2$, since these are typical resolutions of ultrasound data on a real scanner. In these cases, the proposed method is almost 19 times and 24 times faster respectively.

Since the application in question, ultrasound, has a real time requirement, it is not only interesting to see the raw processing power, but also how many frames it can render pr second. This is shown in Table 4.4. Note that this table does not show an "apple to apple" comparison. The colour versions compute 4 times as many elements per frame, but as discussed in Section 3.2.3, ultrasound only needs one channel of data so computing four is a waste of resources if it is possible to compute one in less time.

| Algorithm \ N | $128^2$ | $256^2$ | $512^2$ | $1024^2$ | $2048^2$ |
|---|---|---|---|---|---|
| gpu-mono-16 | 1782 | **1430** | **975** | **405** | **94** |
| gpu-mono-32 | 1801 | 1414 | **975** | 306 | 65 |
| gpu-colour-16 | 632 | 530 | 324 | 104 | 31 |
| gpu-colour-32 | 631 | 532 | 311 | 104 | 26 |
| previous-gpu-32 | 1558 | 493 | 111 | 29 | 5 |
| cpu-mono-32-single | 1081 | 270 | 69 | 17 | 4 |
| cpu-mono-32-dual | **2162** | 541 | 138 | 34 | 8 |

Table 4.4: Comparison of different wavelet algorithms, showing the number of frames rendered per second. Bold indicates the highest fps.

For practical use with ultrasound the numbers in Table 4.4 are probably more interesting than the raw processing number. Not because we would like to render thousands of frames per second, the limit from the transducers are 20 fps anyway, but rather because it tells us how much more processing we can do before we dip below the required 20 fps. One of the goals of this thesis was to consider if porting the enhancement techniques to the GPU would open up the possibility of more advanced techniques. The gpu-mono techniques (the most interesting ones from ultrasound perspective) achieves an astonishing 975 frames per second. This leaves a lot of additional GPU resources available for more advanced techniques. And if we look at only the filtering part the results are even more convincing. As we shall see shortly, we only use $\sim$7% of the total spent time on filtering. The rest of the time is uploading, synthesis and analysis which will be constant even if the filtering techniques are made more advanced.

### 4.2.2 Further analysis

| Part \ N | $128^2$ | $256^2$ | $512^2$ | $1024^2$ | $2048^2$ |
|---|---|---|---|---|---|
| Forward DWT | 0.113 | 0.115 | 0.078 | 0.147 | 0.159 |
| Backward DWT | 0.307 | 0.401 | 0.694 | 1.157 | 2.033 |
| Filtering | 0.085 | 0.095 | 0.069 | 0.111 | 0.110 |
| Uploading | 0.033 | 0.058 | 0.159 | 1.002 | 8.295 |

Table 4.5: Breakdown of walltime for GPU Monochrome 16-bit

In Table 4.5, we have timed the different parts of the gpu-mono-16 program. In Figure 4.2, the numbers from Table 4.5 are shown on a log-log scale. We have chosen to only look at the gpu-mono-16 bit program because this is the most interesting for practical use in ultrasound applications, but the relative sizes of the data are also representative for the 32 bits and colour versions. This choice will be discussed in more detail in the discussion part.

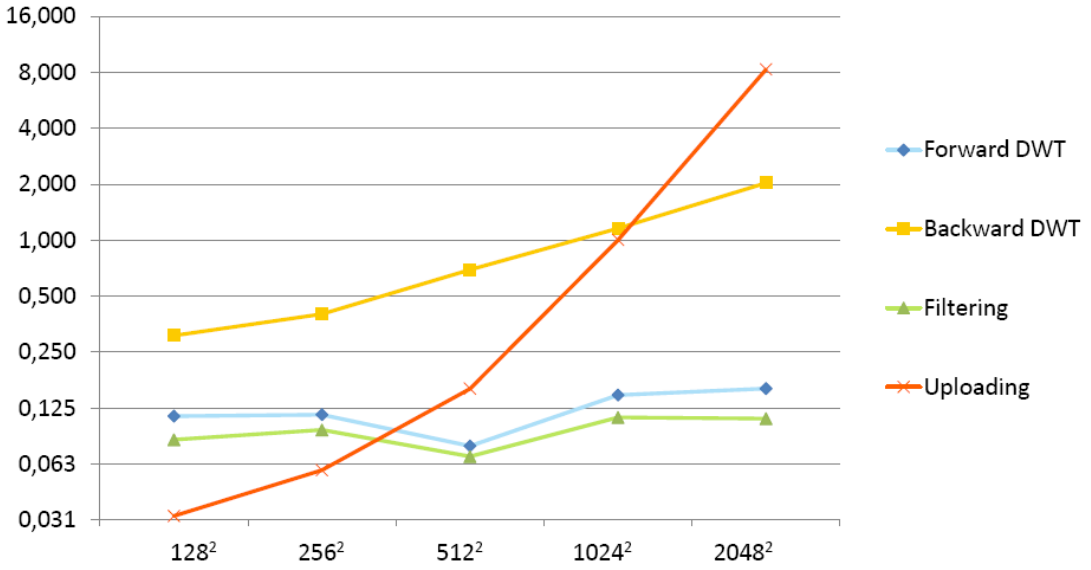
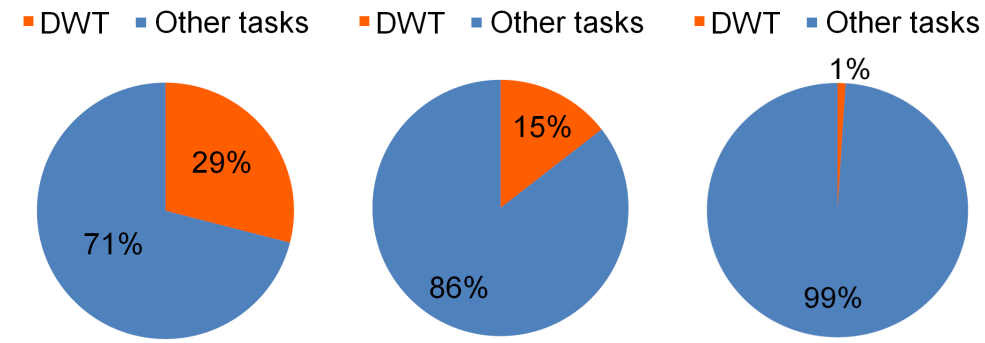



Figure 4.2: Breakdown of walltime GPU Monochrome 16 bit on a log-log sale

One of the most intriguing aspects of computing the DWT on the GPU is that it offloads the CPU which can be used to do other tasks. The real-time requirement of 3D ultrasound is 20 fps, and Table 4.6 shows the CPU load under this requirement. Only 20 fps is required because the 3D ultrasound probes can't gather data any faster. The 3D case is especially interesting since it is the most data intensive mode, and offloading the CPU can be very beneficial. To slow down the GPU versions, idle statements were inserted into the code to make the program render exactly 20 fps. The GPU numbers in Table 4.6 were measured using the 'Task Manager' application in Windows. The CPU numbers were calculated based on the numbers in Table 4.4. Numbers higher than 100% indicate failure to achieve 20 fps. Figure 4.3 shows the CPU load for different solutions on $512^2$ elements. These tables show that by computing the DWT on the GPU the CPU is offloaded considerably. On $512^2$ the load is 29% on the original single

| Algorithm \ N | $128^2$ | $256^2$ | $512^2$ | $1024^2$ | $2048^2$ |
|---|---|---|---|---|---|
| gpu-mono-16 | 1.0 % | 1.0 % | 1.0 % | 3.0 % | 8.0 % |
| cpu-mono-32-single | 1.8 % | 7.4 % | 29.0 % | 118.4 % | 473.6 % |
| cpu-mono-32-dual | 0.9 % | 3.7 % | 14.5 % | 59.2 % | 236.8 % |

Table 4.6: Load on the CPU while rendering 20 fps.




(a) The current status, single CPU

(b) Using dual-core CPU

(c) The proposed solution, using the GPU

Figure 4.3: Comparison of CPU load on $512^2$ elements.

CPU version. If we compute it on the GPU, the load is only 1%! This makes the CPU available for other tasks.

In Section 3.2.3, we discussed the rationale for having a mono version. Theoretically it could have a frame rate four times higher than the colour version. In that case, it would compute the same number of elements per second. This theoretical limit is not possible to achieve, and Table 4.7 shows the efficiency achieved by the mono version in 16 and 32 bits. An entry of 100% would mean that it is as effective as the colour version.

| Algorithm \ N | $128^2$ | $256^2$ | $512^2$ | $1024^2$ | $2048^2$ |
|---|---|---|---|---|---|
| gpu-mono-16 | 70.5 % | 67.5 % | 75.1 % | 97.5 % | 74.6 % |
| gpu-mono-32 | 71.4 % | 66.4 % | 78.4 % | 73.7 % | 62.0 % |

Table 4.7: The efficiency of the mono versions using the colour versions as reference.

All the previous numbers were benchmarked using the GeForce 8800 GTX graphics card. In Table 4.8, and Table 4.9, the raw numbers and speedup for the GeForce 7600 GS are shown. This is interesting from a computation/price. The GeForce 7600 was not able to compute $2048^2$ in 32 bits because of its small amount of memory, and those results are marked NA in the tables.

In Table 4.10, the raw performance, adjusted for price, is depicted. The prices used were $275 for the CPU, $850 for G8800, and $87 for GeForce 7600. [2]

[2]Prices gathered from [38]

| Algorithm \ N | $128^2$ | $256^2$ | $512^2$ | $1024^2$ | $2048^2$ |
|---|---|---|---|---|---|
| gpu-mono-16 | 1917 | **535** | **174** | **42** | **10** |
| gpu-mono-32 | 1470 | 387 | 103 | 25 | 6 |
| gpu-colour-16 | 937 | 266 | 89 | 22 | 5 |
| gpu-colour-32 | 565 | 182 | 59 | 15 | NA |
| cpu-mono-32-single | 1081 | 270 | 69 | 17 | 4 |
| cpu-mono-32-dual | **2133** | 533 | 133 | 33 | 8 |

Table 4.8: Performance comparison of different wavelet algorithms, measuring the number of elements processed pr. micro second. GPU used: GeForce 7600GS

| Algorithm \ N | $128^2$ | $256^2$ | $512^2$ | $1024^2$ | $2048^2$ |
|---|---|---|---|---|---|
| gpu-mono-16-7600gs | 1.8 | **2.0** | **2.5** | **2.5** | **2.4** |
| gpu-mono-32-7600gs | 1.4 | 1.4 | 1.5 | 1.5 | 1.5 |
| gpu-colour-16-7600gs | 0.9 | 1.0 | 1.3 | 1.3 | 1.2 |
| gpu-colour-32-7600gs | 0.5 | 0.7 | 0.9 | 0.9 | NA |
| cpu-mono-32-single | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| cpu-mono-32-dual | **2.0** | **2.0** | 1.9 | 2.0 | 2.0 |

Table 4.9: Relative speedups using GeForce 7600 GS

The GeForce 7600 GS is low-end board from the previous generation graphic boards. At the time of writing it was available for less than 100$ [38]. As the results show, it is quite a bit slower than the GeForce 8800, but still faster than the CPU for all data sizes bigger than $128^2$. Considering that the desktop version of the processor used costs approximately 270$ this is a very good result. For ultrasound scanners the price/performance aspect is not very important, but still not irrelevant. But since the scanner will contain both a high-end CPU and GPU, the offloading of the CPU discussed earlier is more important. For other applications however, Figure 4.4 should be interesting. It shows that for small sizes (less than $1024^2$) the GeForce 7600 gives most computation for each dollar. On $1024^2$, and higher, the GeForce 8800 is the most economic. Considering that the 8800 is 10 times as expensive as the 7600 GS, this result really shows how much the GPUs develop each hardware generation.

The visual quality achieved is comparable to the ones achieved in [19] and [31], and example renderings are shown in Figure 4.5. In Figure 4.6, the difference between 16 and 32 bits on an otherwise identical rendering is shown.

## 4.3 Discussion

As we can see from 4.2 and 4.3, our GPU method is much faster than the previous methods, both CPU and GPU. We achieve a speedup of 30 compared to the CPU and speedup of three compared to the previous GPU implementation. In addition, we are able to offload the CPU from 29% to 1% when computing the DWT filter on $512^2$ data elements.

The numbers also give some other interesting findings. First, the GPU method speedups get higher when the data size increases. This coincide with findings in [1]. The main reason lies in

| Algorithm \ N | $128^2$ | $256^2$ | $512^2$ | $1024^2$ | $2048^2$ |
|---|---|---|---|---|---|
| gpu-mono-16 | 5.73 | 18.40 | 50.15 | 83.42 | 77.38 |
| gpu-colour-16 | 8.13 | 27.26 | 66.76 | **85.52** | **103.70** |
| gpu-mono-16-7600gs | **59.83** | **66.81** | **87.06** | 83.40 | 81.55 |
| gpu-colour-16-7600gs | 29.23 | 33.23 | 44.53 | 43.94 | 41.41 |
| cpu-mono-32-single | 10.73 | 10.73 | 10.96 | 10.73 | 10.73 |

Table 4.10: Performance from Table 4.2 and 4.8 adjusted for price.

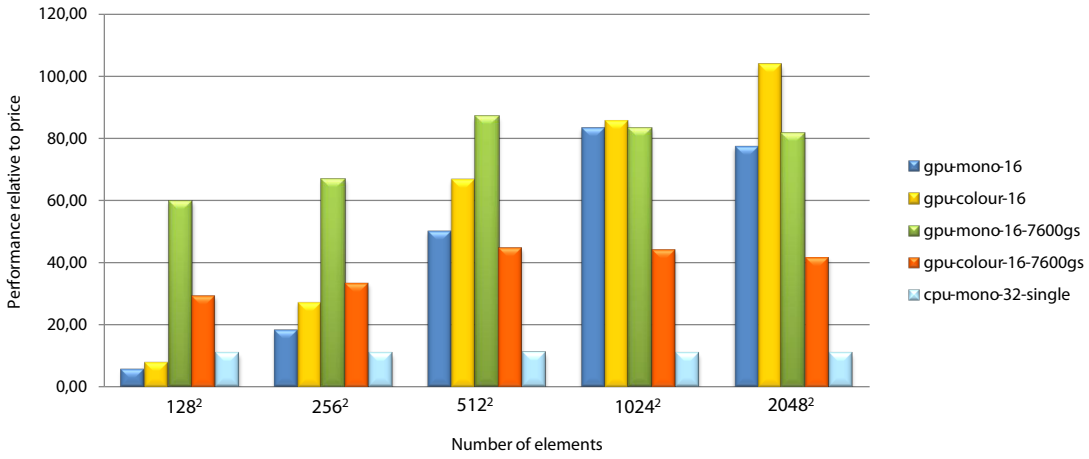



Figure 4.4: The performance of 16-bit computation relative to size.

the overhead in executing a single rendering. This is constant regardless of the data size and is amortised in the larger data sets. As can be seen in Table 4.5 and in Figure 4.2, the analysis and filtering time are sublinear compared to the data size because the 128 processors of the GPU are used more efficiently on large data.

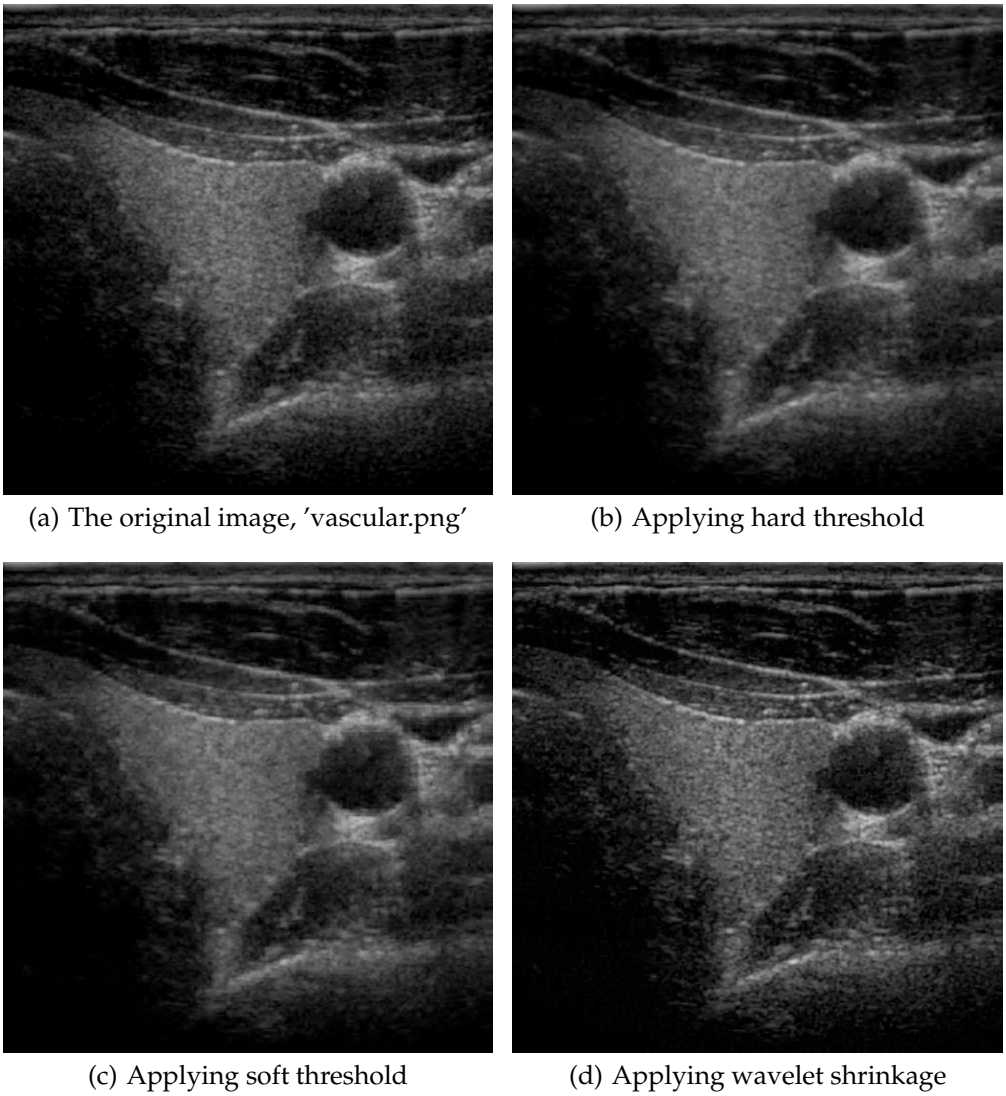
16 and 32 bits precision seem to be irrelevant when working with small data sets, but on larger sets there is a measurable difference. For data sets of size $2048^2$ 16 bits precision is $\sim$20% and $\sim$44% faster than 32 bits for the colour and mono version respectively. As can be seen in Figure 4.6 it is very hard to see a visual difference between the two renderings.

The mono version is less efficient than the coloured version in terms of raw processing power, and varies quite a bit depending on data size. On $1024^2$ elements in 16 bits the efficiency is as high as 97.5% which is very good. On other data such as $2048^2$ using 32 bits, the efficiency is as low as 62%. For these data sizes and types other approaches than ours should probably be tested. This is left as future work.

In Table 4.11, we have compared our estimated results with the actual results. For both CPU and GPU timings we were off by almost an order of magnitude. This is expected since we ignored overhead and latency in our calculations. The speedup estimate, however, is almost on target.

| Variable | Predicted | Actual | Error |
|---|---|---|---|
| CPU | 3.15ms | 29ms | 9.2 |
| GPU | 0.5ms | 3.8ms | 7.6 |
| Speedup | 6.3 | 7.3 | 1.2 |

Table 4.11: Comparison between theoretical model and actual results.



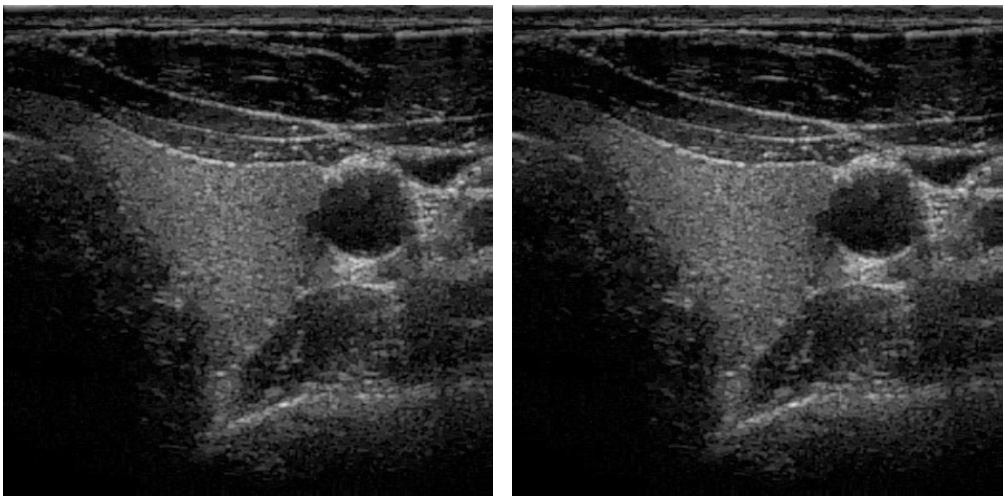
(a) The original image, 'vascular.png'

(b) Applying hard threshold

(c) Applying soft threshold

(d) Applying wavelet shrinkage

Figure 4.5: Comparison of two different enchancement techniques



(a) 16 bits (b) 32 bits

Figure 4.6: Comparison of 16bit and 32bit precision

Chapter 5

# Conclusions and Future Work

*I may not have gone where I intended to go, but I think I have ended up where I needed to be*

**-Douglas Adams**

## 5.1 Conclusion

In this thesis, we have looked at wavelet algorithm development target at the GPU. Our motivation was the increased need for speeding up the calculations needed in 3D ultrasound, where presently the wavelet calculations take a significant amount of processing time.

By implementing a program that both outperforms and offloads the CPU we have shown that performing wavelet image enhancement on the GPU is a viable solution. The techniques developed in this thesis were deemed so successful by GE Healthcare that they will be included in their next generation of cardiac ultrasound scanners.

### Main contributions

This thesis has three main contributions:

*First*, we showed that it is possible to implement a very fast wavelet transform on consumer-level graphics hardware. The wavelet transform is the final step of of the "baking chain" – a term used in ultrasound imaging to describe all the methods that are applied to the data after it is collected by the transducer, but before they are displayed on a screen. Our novel wavelet methods were developed, implemented and tested with such a baking chain. The techniques implemented were directly applicable to a modern cardiac ultrasound scanner.

*Second*, we showed that our fast GPU wavelet methods can significantly outperform the current state-of-the-art CPU-based implementations. Our implementation achieved a speedup of 29.8 compared to a SIMD-optimised CPU version.

Porting algorithms from the CPU to the GPU usually results in a speedup of about 30%. Howwever, we were able to show a three-fold speedup over previous wavelet GPU implementations. We achieved this by analysing the underlying hardware to find and remove the

performance bottlenecks. The main performance bottleneck identified, was the use of dependent, and too many, texture reads. These bottlenecks were removed by creating a novel GPU method which used advanced GPU techniques to avoid texture fetches.

*Third*, we have shown that it is possible to offload the CPU so that it on realistically sized data reduces its load from 29% to 1%. The main implication is that the CPU can be used to do other tasks like furher refining the data before they are fed to the wavelet technique. This opens up possibilities for more advanced uses like High Definition Television (HDTV) denoising and other exiting applications.

## 5.2 Future work

Unfortunately, the time allocated for a master thesis is very limited. The following list tries to summarise some of the ideas that we would like to explore further.

- **Improve the backward DWT**: As was seen in Table 4.5 the backward DWT used a big portion of the time. In this thesis, we used quads to do static branch resolution. This cause us to render each synthesis twice. It might be possible to use the 'stencil test' instead. This technique have been successfully applied to sorting, [1].
- **Integrate the code with JASPER**: Wavelets can be used for compression, and the JPEG2000 standard is based on this. JASPER is a open source package that enables reading and writing of JPEG2000. It should be possible to integrate the wavelet component from this thesis into JASPER.
- **Port more algorithms to the GPU**: We have ported the last part of the *baking chain* of a ultrasound scanner to the GPU. As shown in the results chapter the GPU still has much horsepower left for other tasks. It should be possible to add more of the baking chain to the GPU.
- **DirectX10 and CUDA**: During the development of this thesis Microsoft released their new graphics framework called DirectX10. It has many new possibilities that further generalises the programming on a GPU. This can result in higher performance or easier programming. NVIDIA also shipped a new framework called CUDA early in 2007. It is a framework for easy development of GPGPU. Unfortunately it is only for NVIDIA, but it would be interesting to see how the algorithms could be expressed in the CUDA framework.
- **3D GPU processing**: The algorithms work on 2D data. The latest industry transducers produce 3D data. The newer GPU has native support for 3D data via *3D textures*: and it would be interesting to transfer the methods described in this thesis to 3D on a GPU with 3D texturing features.
- **HDTV and other applications**: Our new methods allows computation in real time of much more data than previously. Denoising of content from High-definition television (HDTV) cameras in real time is one exiting possibility.

# Bibliography


[1] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[2] Matt Pharr. *Mapping Computational Concepts to GPUs*. Addison Wesley, 2005.

[3] John Owens. Streaming architectures and technology trends. In Matt Pharr, editor, *GPU Gems 2*, pages 457–470. Addison Wesley, 2005.

[4] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.

[5] Nvidia. Available on: `http://developer.download.nvidia.com/compute/cuda/0_81/NVIDIA_CUDA_Programming_Guide_0.8.2.pdf`, accessed on 1.april 2007, 2007.

[6] Nvidia. Available on: `http://www.nvidia.com`, accessed on 1.april 2007, 2007.

[7] Intel. Available on: `http://www.intel.com/intel/finance/pricelist/processor_price_list.pdf`, accessed on 1.april 2007, 2007.

[8] I Buck, K Fatahalian, and P Hanrahan. Gpubench: Evaluating gpu performance for numerical and scientific applications. *2004 ACM Workshop on General-Purpose Computing on Graphics Processors*, pages pp. C–20, 2004.

[9] Intel. Available on: `http://www.intel.com/products/processor/`, accessed on 1.april 2007, 2007.

[10] Magnus Wahrenberg. Volume rendeirng for 3d display. Master's thesis, Royal Institute of Technology (KTH), School of Computer Science and Communication, 2006.

[11] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[12] Leif Chrisitan Larsen. Utilizing gpus on cluster computers. Master's thesis, Norwegian University of Science and Technology (NTNU), Department of Computer and Information Science, 2006.

[13] Mark Harris. Mapping computational concepts to gpus. In Matt Pharr, editor, *GPU Gems 2*, pages 493–508. Addison Wesley, 2005.

[14] Ian Buck. Taking the plunge into gpu computing. In Matt Pharr, editor, *GPU Gems 2*, pages 493–508. Addison Wesley, 2005.

[15] Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, David Tuft, and Dinesh Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. *Tech. Rep. TR05-016*, 2005.

[16] T. R. Nelson and D. H. Pretorius. Three-dimensional ultrasound imaging. *Ultrasound Med Biol.*, 1198.

[17] J. W. Goodman. Some fundamental properties of speckle. *J. Opt. Soc. Am.*, (66):1145–, 1976.

[18] A. K. Jain. *Fundamentals of digital image processing*. Prentice Hall., Englewood Cliffs, NJ, 1989, 1989.

[19] Xuli Zong, A.F. Laine, and E.A. Geiser. Speckle reduction and contrast enhancement of echocardiograms viamultiscale nonlinear processing. *IEEE Transactions on Medical Imaging*, 17(4):pp. 532–540, 1998.

[20] E. R. Ritenour, T. R.Nelson, and U. Raff. Application of median filter to digital readiographic images. *Proc. 7th Int. Conf. Acoust. Speech, Signal Processing*, pages 23.1.1–23.1.4, 1984.

[21] A. Loannidis, D. Kazakos, and D. D. Watson. Application of median filtering on nuclear medicine scintigram images. *Proc. 7th Int. Conf. Pattern Recognition*, pages pp. 33–36, 1984.

[22] T. Loupas, W. N. McDicken, and P. L. Allen. Noise reduction in ultrasonic images by digital filtering. *Br. J. Radiol.*, pages pp. 389–392, 1987.

[23] Dr.ing. Erik Steen. correspondence, 2007.

[24] J. Cooley, R. Garwin, C. Rader, B. Bogert, and T. Stockham. The 1968 arden house workshop on fast fourier transform processing. *Audio and Electroacoustics, IEEE Transactions on*, 17(2):66–76, 1969.

[25] Ingrid Daubechies. *Ten lectures on wavelets*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[26] Gerald. Kaiser. *A Friendly Guide to Wavelets*. Springer, 1994.

[27] Claudia Kerstin Schremmer. *Multimedia Applications of the Wavelet Transform*. PhD thesis, University of Mannheim, 2002.

[28] Aleksandra Pizurica. *Image Denoising Using Wavelets and Spatial Context Modeling*. PhD thesis, Universiteit Gent, 2002.

[29] S. G. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(7):674–693, 1989.

[30] Richard E. Woods Rafael C. Gonzales. *Digital Image Processing, Second Edition*. Prentice Hall., 2002.

[31] David L. Donoho. De-noising by soft-thresholding. *IEEE Transactions on Information Theory*, 41(3):613–627, 1995.

[32] Jason L. Mitchell, Marwan Y. Ansari, and Evan Hart. Advanced image processing with directx 9 pixel shaders. In Wolfgang F Engel, editor, *ShaderX*$^2$, pages 439–464. Wordware Publishing Inc, 2003.

[33] Tien-Tsin Wong, Chi-Sing Leung, Pheng-Ann Heng, and Jianqing Wang. Discrete wavelet transform on consumer-level graphics hardware. *IEEE Transactions On Multimedia*, 2007.

[34] W. Sweldens. The lifting scheme: A construction of second generation wavelets. *SIAM J. Math. Anal.*, 29(2):511–546, 1997.

[35] Asadollah Shahbahrami, Ben Juurlink, and Stamatis Vassiliadis. Performance comparison of simd implementations of the discrete wavelet transform. In *ASAP '05: Proceedings of the 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*, pages 393–398, Washington, DC, USA, 2005. IEEE Computer Society.

[36] David J. Lilja. *Measuring Computer Performance A Practitioner's Guide*. Cambridge University Press, 2005.

[37] Nvidia. Available on: `http://forum.stanford.edu/events/2007/plenary/slides/hanrahan%20slides.pdf`, accessed on 1.april 2007, 2007.

[38] Komplett.no. Available on: `http://www.komplett.no`, accessed on 29.april 2007, 2007.

Appendix A

# Implementation

In this chapter, we will dicuss the programs that were made during this thesis. The code is split into three main parts. *gpuwaveletlib* is a C++ library that exposes a library for the filters and techniques we have presented in chapter 3. It used by the two front end programs. One command-line (`cmdgpuwavelet`), and one GUI based (`GPUVideoProcessor`).

## A.1 GPUVideoProcessor

`GPUVideoProcessor` is a front end for *gpuwaveletlib* that exposes the different filters and techniques for a user in an easy to use GUI. The user can select input file(s), a filter to use, parameters to the filters and desired output. This GUI has made it easier to test and implement different filters. It should also be useful for other authors of wavelet and spatial based filters. In the following text, an upper case bold letter (eg **A**) will refer to a UI element in Figure A.2.

`GPUVideoProcessor` can process still images, movies and even process input from the webcamera. PNG, Bitmap-files, GIF and JPEG are supported still image formats, and uncompressed AVI, MPEG-1 and WMV (Windows Media Files) are supported video formats. The input can be chosen from the drop-down box marked **B**. The drop-down lists all the media files in the same directory as the program. In addition, there is a option to browse for more files and a webcamera option if one is available.

The program has two main modes, one for the spatial filters and one for the wavelet filters. This is almost invisible for the user since it it chosen automatically when the user choses a filter. But in the wavelet mode there are a couple of more options. The normal spatial filters include Gaussian filtering (Section 3.3.1), and Sharpening Filtering (3.3.2). In addition, a Pass Through options is available which just shows the picture without any manipulation. The wavelet based filters that are available include Soft Thresholding (3.3.3), Hard Thresholding (3.3.4), Wavelet Shrinkage and HERMAADETSTAANOE. All these filters are chosen from the drop-down box **C**. When the user chooses a filter one or more sliders appear (**F**) which control the parameters of the filter. One example is the *threshold* parameter of the Hard Thresholding filter. Only the sliders for the parameters that the current filter actually support is shown.

When the user choose a wavelet filter two more drop-down boxes appear to let the user choose visual output format (**D**) and which wavelet to use (**E**). The visual output controls which

part of the wavelet filtering is shown. The user can choose between Synthesis, Analysis and Filtered Analysis. In Figure A.1, the three options and which part they visualize is shown. Synthesis shows the final result after analysing, applying the filters and synthesising. This

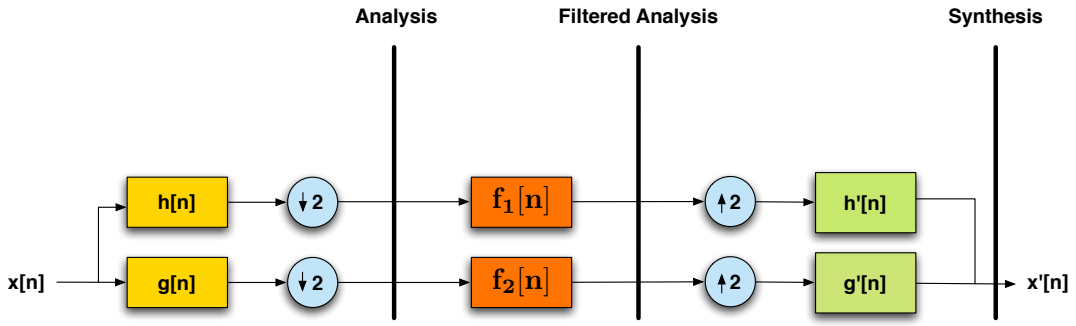



Figure A.1: The different wavelet visual output options

is the real end result. The Analysis option shows the detail and approximation coefficients layed out in the same fashion as Figure 2.13. The details are usually very close to zero so 0.5 has been added to them to make them more visible in the visualisation. The Filtered Analysis option shows the coefficients after applying the filters. This is of great use when designing the filters and especially when debugging them.

The other wavelet specific option is which wavelet to use. The program supports the use of Daub 9/7 and Daub 5/3.

At the top of the window some diagnostic text is outputted (**A**). It shows the hardware used and the size of the picture, but the most interesting part is the Frames Pr Second (FPS) counter. This can be used to check the speed of different methods.

For demonstration purposes a full screen option has been implemented as well and can be chosen by pressing button **G**.

In addition to the options available in the GUI, there are a number of options tunable by editing a config file `settings.txt`. The most important one is probably the ability to set the number of levels of the wavelet analysis. The other options mainly control optimisation options for benchmarking purposes.

## A.2 Cmdgpuwavelet

`cmdgpuwavelet` is a command line utility that supports all the same options as the GUI version. There are two main benefits of having a command line utility in addition to the GUI. The command line utility makes it easy to set up tests for benchmarking and it makes it easy to set up a chain of filters for the more complex filter setups.

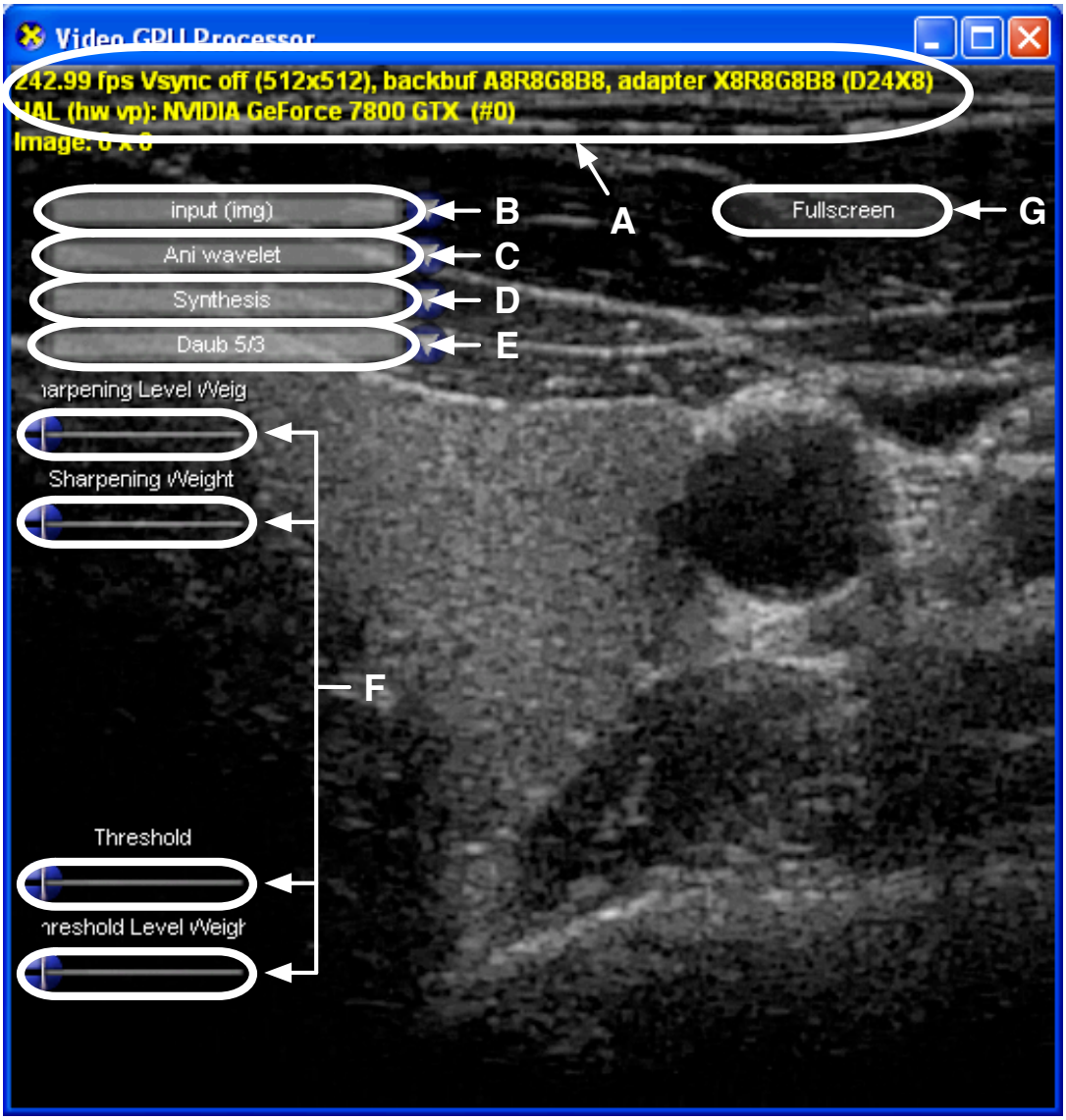



Figure A.2: Screenshot of image processor

Appendix B

# GPU microprograms

In this section, the most important shader microcode is listed.

## B.1 Daubechies 9/7 Mono DWT

```
float4x4 mvpMatrix : WorldViewProjection;
float texelWidth;
float texelHeight;


// The 9/7 Version
float analysisLowKernel[] = { 0.026748757411f, -0.016864118443f,
    -0.078223266529f, 0.266864118443f, 0.602949018236f, 0.266864118443f,
    -0.078223266529f, -0.016864118443f, 0.026748757411f };
float analysisHighKernel[] = { 0.0f,    0.091271763114f,  -0.057543526229f,
     -0.591271763114f, 1.11508705f, -0.591271763114f, -0.057543526229f,
    0.091271763114f, 0.0f };
float synthesisLowKernel[] = { 0,    -0.091271763114,  -0.057543526229,
    0.591271763114, 1.11508705,  0.591271763114, -0.057543526229,
    0.091271763114f, 0};
float synthesisHighKernel[] = { 0.026748757411f, 0.016864118443f,
    -0.078223266529f, -0.266864118443f, 0.602949018236f,-0.266864118443f,
    -0.078223266529f, 0.016864118443f, 0.026748757411f };
float evenSynthesisFilter[] = { 0, 0.016864118443,
    -0.057543526229,-0.266864118443,1.11508705,-0.266864118443,
    -0.057543526229, 0.016864118443,0};
float oddSynthesisFilter[] = {  0.026748757411,  -0.091271763114,
    -0.078223266529,  0.591271763114,0.602949018236, 0.591271763114,
    -0.078223266529, -0.091271763114, 0.026748757411 };

texture SourceTexture;

sampler2D SourceSampler = sampler_state {
   Texture = (SourceTexture);
   MipFilter = NONE;
```

```
    MinFilter = Point;
  MagFilter = Point;
    AddressU = Mirror;
    AddressV = Mirror;
    MaxANISOTROPY = 1;
};

struct PS_OUT {
  float4 Color : COLOR0;
};

struct TEX_VS_IN {
  float4 Position : POSITION;
    float2 Tex : TEXCOORD0;
};


TEX_VS_IN NormalVS(TEX_VS_IN input) {
  TEX_VS_IN output = (TEX_VS_IN)0;
  output.Position = mul(input.Position, mvpMatrix);
  output.Tex  = input.Tex;
  return output;
}

PS_OUT HorisontalAnalysis(TEX_VS_IN input) {
  PS_OUT output = (PS_OUT)0;

  float3 vals[10];

  for(int i=0;i<10;i++) {
    vals[i] = tex2D(SourceSampler, float2(input.Tex.x + (i-4) * texelWidth,
        input.Tex.y));
  }

  for(int i=0;i<9;i++) {
    output.Color.r += vals[i].r * analysisLowKernel[i];
    output.Color.g += vals[i+1].r * analysisHighKernel[i];
  }


  output.Color.a = 1.0f;
  return output;
}

PS_OUT VerticalAnalysis(TEX_VS_IN input) {
  PS_OUT output = (PS_OUT)0;

  float3 vals[10];

  for(int i=0;i<10;i++) {
    vals[i] = tex2D(SourceSampler, float2(input.Tex.x, input.Tex.y + (i-4) *
         texelHeight));
  }

  for(int i=0;i<9;i++) {
```

```
    output.Color.rg += vals[i].rg * analysisLowKernel[i];
    output.Color.ba += vals[i+1].rg * analysisHighKernel[i];
  }

  return output;
}


technique WaveletAnalysis {
  pass P0 {
    PixelShader = compile ps_2_0 HorisontalAnalysis();
    VertexShader = compile vs_2_0 NormalVS();
    AlphaBlendEnable = false;
  }
  pass P1 {
    PixelShader = compile ps_2_0 VerticalAnalysis();
    VertexShader = compile vs_2_0 NormalVS();
    AlphaBlendEnable = false;
  }
}

// The synthesis uses 5 taps
struct SYNTHESIS_TEX {
  float4 Position : POSITION;
    float2 Tap1 : TEXCOORD0; // Min two texels
    float2 Tap2 : TEXCOORD1; // Min one texels
    float2 Tap3 : TEXCOORD2; // Base position
    float2 Tap4 : TEXCOORD3; // Pluss one texels
    float2 Tap5 : TEXCOORD4; // Pluss two texels
};



SYNTHESIS_TEX HorisontalSynthesisVS(TEX_VS_IN input) {
  SYNTHESIS_TEX output = (SYNTHESIS_TEX)0;

  output.Position = mul(input.Position, mvpMatrix);
  output.Tap1 = float2(input.Tex.x - 2 * texelWidth, input.Tex.y);
  output.Tap2 = float2(input.Tex.x - texelWidth, input.Tex.y );
  output.Tap3 = input.Tex;
  output.Tap4 = float2(input.Tex.x + texelWidth, input.Tex.y );
  output.Tap5 = float2(input.Tex.x + 2 * texelWidth, input.Tex.y );

  return output;
}

SYNTHESIS_TEX VerticalSynthesisVS(TEX_VS_IN input) {
  SYNTHESIS_TEX output = (SYNTHESIS_TEX)0;

  output.Position = mul(input.Position, mvpMatrix);
  output.Tap1 = float2(input.Tex.x, input.Tex.y - 2 * texelHeight);
  output.Tap2 = float2(input.Tex.x, input.Tex.y - texelHeight);
  output.Tap3 = input.Tex;
  output.Tap4 = float2(input.Tex.x, input.Tex.y + texelHeight);
  output.Tap5 = float2(input.Tex.x, input.Tex.y + 2 * texelHeight);
```

```
  return output;
}

PS_OUT SynthesisEven(SYNTHESIS_TEX input) {
  PS_OUT output = (PS_OUT)0;

  float4 vals[5];
  vals[0] = tex2D(SourceSampler, input.Tap1);
  vals[1] = tex2D(SourceSampler, input.Tap2);
  vals[2] = tex2D(SourceSampler, input.Tap3);
  vals[3] = tex2D(SourceSampler, input.Tap4);
  vals[4] = tex2D(SourceSampler, input.Tap5);

  // Even starts on low
  output.Color.rb = vals[0].rg * evenSynthesisFilter[0]
          + vals[0].ba * evenSynthesisFilter[1]
          + vals[1].rg * evenSynthesisFilter[2]
          + vals[1].ba * evenSynthesisFilter[3]
          + vals[2].rg * evenSynthesisFilter[4]
          + vals[2].ba * evenSynthesisFilter[5]
          + vals[3].rg * evenSynthesisFilter[6]
          + vals[3].ba * evenSynthesisFilter[7]
          + vals[4].rg * evenSynthesisFilter[8];
  return output;
}

PS_OUT SynthesisOdd(SYNTHESIS_TEX input) {
  PS_OUT output = (PS_OUT)0;

  float4 vals[5];
  vals[0] = tex2D(SourceSampler, input.Tap1);
  vals[1] = tex2D(SourceSampler, input.Tap2);
  vals[2] = tex2D(SourceSampler, input.Tap3);
  vals[3] = tex2D(SourceSampler, input.Tap4);
  vals[4] = tex2D(SourceSampler, input.Tap5);

  // Odd starts on high
  output.Color.rb = vals[0].ba * oddSynthesisFilter[0]
          + vals[1].rg * oddSynthesisFilter[1]
          + vals[1].ba * oddSynthesisFilter[2]
          + vals[2].rg * oddSynthesisFilter[3]
          + vals[2].ba * oddSynthesisFilter[4]
          + vals[3].rg * oddSynthesisFilter[5]
          + vals[3].ba * oddSynthesisFilter[6]
          + vals[4].rg * oddSynthesisFilter[7]
          + vals[4].ba * oddSynthesisFilter[8];
  return output;
}

technique WaveletSynthesis {
  pass P0 {
    PixelShader = compile ps_2_0 SynthesisEven();
    VertexShader = compile vs_2_0 VerticalSynthesisVS();
    AlphaBlendEnable = false;
```

```
  }
  pass P1 {
    PixelShader = compile ps_2_0 SynthesisOdd();
    VertexShader = compile vs_2_0 VerticalSynthesisVS();
    AlphaBlendEnable = false;
  }
  pass P2 {
    PixelShader = compile ps_2_0 SynthesisEven();
    VertexShader = compile vs_2_0 HorisontalSynthesisVS();
    AlphaBlendEnable = false;
  }
  pass P3 {
    PixelShader = compile ps_2_0 SynthesisOdd();
    VertexShader = compile vs_2_0 HorisontalSynthesisVS();
    AlphaBlendEnable = false;
  }
}

float4 selector;
float addValue; // to visualize high pass we often add 0.5.

PS_OUT DisplayMonoWaveletAnalysisPS(TEX_VS_IN input) {
  PS_OUT output = (PS_OUT)0;

  // Select which component we want to show this pass
  output.Color.rgb = length(tex2D(SourceSampler, input.Tex) * selector);
  // And add a value in case we are displaying the high pass
  output.Color.rgb += addValue;
  output.Color.a = 1.0f;

  return output;
}

technique DisplayMonoWaveletAnalysisTechnique {
  pass P0 {
    PixelShader = compile ps_2_0 DisplayMonoWaveletAnalysisPS();
    VertexShader = compile vs_2_0 NormalVS();
  }
}
```

Listing B.1: "Daub 9/7 Mono"

## B.2 Daubechies 9/7 Colour DWT

```
float4x4 mvpMatrix;

float texelWidth;
float texelHeight;

// From wikipedia
// Daubechies 9/7-CDF-wavelet
float analysisLowKernel[] = { 0.026748757411f, -0.016864118443f,
    -0.078223266529f, 0.266864118443f, 0.602949018236f, 0.266864118443f,
```

```
    -0.078223266529f, -0.016864118443f, 0.026748757411f };
float analysisHighKernel[] = { 0.0f,    0.091271763114f,  -0.057543526229f,
     -0.591271763114f, 1.11508705f, -0.591271763114f, -0.057543526229f,
    0.091271763114f, 0.0f };

// From wikipedia
// NOTE: These are "interleaved".
float evenSynthesisFilter[] = { 0, 0.016864118443,
    -0.057543526229,-0.266864118443,1.11508705,-0.266864118443,
    -0.057543526229, 0.016864118443,0};
float oddSynthesisFilter[] = {  0.026748757411,  -0.091271763114,
    -0.078223266529,  0.591271763114,0.602949018236, 0.591271763114,
    -0.078223266529, -0.091271763114, 0.026748757411 };


texture SourceTexture;
sampler2D SourceSampler = sampler_state {
   Texture = (SourceTexture);
   MipFilter = NONE;
   MinFilter = Point;
  MagFilter = Point;
   AddressU = Mirror;
   AddressV = Mirror;
   MaxANISOTROPY = 1;
};

texture LowPassSourceTexture;
sampler2D LowPassSource = sampler_state {
   Texture = (LowPassSourceTexture);
   MipFilter = NONE;
   MinFilter = Point;
  MagFilter = Point;
   AddressU = Mirror;
   AddressV = Mirror;
   MaxANISOTROPY = 1;
};

texture HighPassSourceTexture;
sampler2D HighPassSource = sampler_state {
   Texture = (HighPassSourceTexture);
   MipFilter = NONE;
   MinFilter = Point;
  MagFilter = Point;
   AddressU = Mirror;
   AddressV = Mirror;
   MaxANISOTROPY = 1;
};

struct PS_OUT {
  float4 Color : COLOR0;
};

struct PS_DUAL_OUT {
  float4 Output1 : COLOR0;
  float4 Output2 : COLOR1;
```

```
};

struct TEX_VS_IN {
  float4 Position : POSITION;
    float2 Tex : TEXCOORD0;
};

// The synthesis uses 5 taps
struct SYNTHESIS_TEX {
  float4 Position : POSITION;
    float2 Tap1 : TEXCOORD0; // Min two texels
    float2 Tap2 : TEXCOORD1; // Min one texels
    float2 Tap3 : TEXCOORD2; // Base position
    float2 Tap4 : TEXCOORD3; // Pluss one texels
    float2 Tap5 : TEXCOORD4; // Pluss two texels
};

SYNTHESIS_TEX HorisontalSynthesisVS(TEX_VS_IN input) {
  SYNTHESIS_TEX output = (SYNTHESIS_TEX)0;

  output.Position = mul(input.Position, mvpMatrix);
  output.Tap1 = float2(input.Tex.x - 2 * texelWidth, input.Tex.y);
  output.Tap2 = float2(input.Tex.x - texelWidth, input.Tex.y );
  output.Tap3 = input.Tex;
  output.Tap4 = float2(input.Tex.x + texelWidth, input.Tex.y );
  output.Tap5 = float2(input.Tex.x + 2 * texelWidth, input.Tex.y );

  return output;
}

SYNTHESIS_TEX VerticalSynthesisVS(TEX_VS_IN input) {
  SYNTHESIS_TEX output = (SYNTHESIS_TEX)0;

  output.Position = mul(input.Position, mvpMatrix);
  output.Tap1 = float2(input.Tex.x, input.Tex.y - 2 * texelHeight);
  output.Tap2 = float2(input.Tex.x, input.Tex.y - texelHeight);
  output.Tap3 = input.Tex;
  output.Tap4 = float2(input.Tex.x, input.Tex.y + texelHeight);
  output.Tap5 = float2(input.Tex.x, input.Tex.y + 2 * texelHeight);

  return output;
}

TEX_VS_IN NormalVS(TEX_VS_IN input) {
  TEX_VS_IN output = (TEX_VS_IN)0;
  output.Position = mul(input.Position, mvpMatrix);
  output.Tex  = input.Tex;
  return output;
}

PS_DUAL_OUT HorizontalAnalysis(TEX_VS_IN input) {
  PS_DUAL_OUT output = (PS_DUAL_OUT)0;

  float3 vals[10];
```

```
  for(int i=0;i<10;i++) {
    vals[i] = tex2D(SourceSampler, float2(input.Tex.x + (i-4) * texelWidth,
        input.Tex.y));
  }
  for(int i=0;i<9;i++) {
    output.Output1.rgb += vals[i] * analysisLowKernel[i];
    output.Output2.rgb += (vals[i+1] * analysisHighKernel[i]);
  }
  output.Output1.a = 1.0f;
  output.Output2.a = 1.0f;

  return output;
}

PS_DUAL_OUT VerticalAnalysis(TEX_VS_IN input) {
  PS_DUAL_OUT output = (PS_DUAL_OUT)0;

  float3 vals[10];

  for(int i=0;i<10;i++) {
    vals[i] = tex2D(SourceSampler, float2(input.Tex.x, input.Tex.y + (i-4) *
        texelHeight));
  }
  for(int i=0;i<9;i++) {
    output.Output1.rgb += vals[i] * analysisLowKernel[i];
    output.Output2.rgb += (vals[i+1] * analysisHighKernel[i]);
  }
  output.Output1.a = 1.0f;
  output.Output2.a = 1.0f;

  return output;
}

// This uses approx 9 texture and 10 arithmetic
PS_OUT SynthesisEven(SYNTHESIS_TEX input) {
  PS_OUT output = (PS_OUT)0;

  output.Color.rgb = evenSynthesisFilter[0] * tex2D(LowPassSource, input.
      Tap1)
         + evenSynthesisFilter[1] * tex2D(HighPassSource, input.Tap1)
         + evenSynthesisFilter[2] * tex2D(LowPassSource, input.Tap2)
         + evenSynthesisFilter[3] * tex2D(HighPassSource, input.Tap2)
         + evenSynthesisFilter[4] * tex2D(LowPassSource, input.Tap3)
         + evenSynthesisFilter[5] * tex2D(HighPassSource, input.Tap3)
         + evenSynthesisFilter[6] * tex2D(LowPassSource, input.Tap4)
         + evenSynthesisFilter[7] * tex2D(HighPassSource, input.Tap4)
         + evenSynthesisFilter[8] * tex2D(LowPassSource, input.Tap5);
  output.Color.a = 1;
  return output;
}

PS_OUT SynthesisOdd(SYNTHESIS_TEX input) {
  PS_OUT output = (PS_OUT)0;
```

```
  output.Color.rgb = oddSynthesisFilter[0] * tex2D(HighPassSource, input.
      Tap1);
  output.Color.rgb += oddSynthesisFilter[1] * tex2D(LowPassSource, input.
      Tap2);
  output.Color.rgb += oddSynthesisFilter[2] * tex2D(HighPassSource, input.
      Tap2);
  output.Color.rgb += oddSynthesisFilter[3] * tex2D(LowPassSource, input.
      Tap3);
  output.Color.rgb += oddSynthesisFilter[4] * tex2D(HighPassSource, input.
      Tap3);
  output.Color.rgb += oddSynthesisFilter[5] * tex2D(LowPassSource, input.
      Tap4);
  output.Color.rgb += oddSynthesisFilter[6] * tex2D(HighPassSource, input.
      Tap4);
  output.Color.rgb += oddSynthesisFilter[7] * tex2D(LowPassSource, input.
      Tap5);
  output.Color.rgb += oddSynthesisFilter[8] * tex2D(HighPassSource, input.
      Tap5);
  output.Color.a = 1;
  return output;
}



// These methods are used to visualize the result of a wavelet analysis.
PS_OUT WriteHigh(TEX_VS_IN input) {
  PS_OUT output = (PS_OUT)0;
  output.Color.rgb = tex2D(HighPassSource, input.Tex);
  output.Color.a = 1.0f;
  return output;
}

PS_OUT WriteLow(TEX_VS_IN input) {
  PS_OUT output = (PS_OUT)0;
  output.Color.rgb = tex2D(LowPassSource, input.Tex);
  output.Color.a = 1.0f;
  return output;
}

// Does back and forth but nothing more Can be used for testing speed
technique FastWavelet {
  pass P0 {
    PixelShader = compile ps_2_0 HorizontalAnalysis();
    VertexShader = compile vs_2_0 NormalVS();
  }

  pass P1 {
    PixelShader = compile ps_2_0 VerticalAnalysis();
    VertexShader = compile vs_2_0 NormalVS();
  }
  pass P2 {
    PixelShader = compile ps_2_0 VerticalAnalysis();
    VertexShader = compile vs_2_0 NormalVS();
  }
```

```
  pass P3 {
    PixelShader = compile ps_2_0 SynthesisEven();
    VertexShader = compile vs_2_0 VerticalSynthesisVS();
  }

  pass P4 {
    PixelShader = compile ps_2_0 SynthesisOdd();
    VertexShader = compile vs_2_0 VerticalSynthesisVS();
  }

  pass P5 {
    PixelShader = compile ps_2_0 SynthesisEven();
    VertexShader = compile vs_2_0 VerticalSynthesisVS();
  }

  pass P6 {
    PixelShader = compile ps_2_0 SynthesisOdd();
    VertexShader = compile vs_2_0 VerticalSynthesisVS();
  }

  pass P7 {
    PixelShader = compile ps_2_0 SynthesisEven();
    VertexShader = compile vs_2_0 HorisontalSynthesisVS();
  }

  pass P8 {
    PixelShader = compile ps_2_0 SynthesisOdd();
    VertexShader = compile vs_2_0 HorisontalSynthesisVS();
  }
}

technique FastWaveletAnalysis {
  pass P0 {
    PixelShader = compile ps_2_0 HorizontalAnalysis();
    VertexShader = compile vs_2_0 NormalVS();
  }

  pass P1 {
    PixelShader = compile ps_2_0 VerticalAnalysis();
    VertexShader = compile vs_2_0 NormalVS();
  }
  pass P2 {
    PixelShader = compile ps_2_0 VerticalAnalysis();
    VertexShader = compile vs_2_0 NormalVS();
  }
}

technique FastWaveletSynthesis {
  pass P0 {
    PixelShader = compile ps_2_0 SynthesisEven();
    VertexShader = compile vs_2_0 VerticalSynthesisVS();
  }

  pass P1 {
    PixelShader = compile ps_2_0 SynthesisOdd();
```

```
    VertexShader = compile vs_2_0 VerticalSynthesisVS();
  }

  pass P2 {
    PixelShader = compile ps_2_0 SynthesisEven();
    VertexShader = compile vs_2_0 VerticalSynthesisVS();
  }

  pass P3 {
    PixelShader = compile ps_2_0 SynthesisOdd();
    VertexShader = compile vs_2_0 VerticalSynthesisVS();
  }

  pass P4 {
    PixelShader = compile ps_2_0 SynthesisEven();
    VertexShader = compile vs_2_0 HorisontalSynthesisVS();
  }

  pass P5 {
    PixelShader = compile ps_2_0 SynthesisOdd();
    VertexShader = compile vs_2_0 HorisontalSynthesisVS();
  }
}

PS_OUT VisualizeHighPassPS(TEX_VS_IN input) {
  PS_OUT output = (PS_OUT)0;
  output.Color.rgb = tex2D(SourceSampler, input.Tex) + 0.5;
  output.Color.a = 1.0f;
  return output;
}


PS_OUT VisualizeHighLowPS(TEX_VS_IN input) {
  PS_OUT output = (PS_OUT)0;
  output.Color.rgb = tex2D(SourceSampler, input.Tex);
  output.Color.a = 1.0f;
  return output;
}

technique VisualizeHighPassTechnique {
  pass P0 {
    PixelShader = compile ps_2_0 VisualizeHighPassPS();
    VertexShader = compile vs_2_0 NormalVS();
  }
}

technique VisualizeLowPassTechnique {
  pass P0 {
    PixelShader = compile ps_2_0 VisualizeHighLowPS();
    VertexShader = compile vs_2_0 NormalVS();
  }
}
```

Listing B.2: "Daub 9/7 Colour"

## B.3 Spatial filters

```
float4x4 mvpMatrix;
float luminance = 0.5;
float epsilon = 0.01;

float sourceWidth = 512;
float sourceHeight = 512;

float texelWidth;
float texelHeight;

float texelWidthOffset;
float texelHeightOffset;

texture SourceTexture;
sampler2D SourceSampler = sampler_state {
   Texture = (SourceTexture);
   MipFilter = NONE;
   MinFilter = Point;
  MagFilter = Point;
   AddressU = Clamp;
   AddressV = Clamp;
   MaxANISOTROPY = 1;
};

struct PS_OUT {
  float4 Color : COLOR0;
};

struct TEX_VS_IN {
  float4 Position : POSITION;
   float2 Tex : TEXCOORD0;
};

// A tex structure for 7 on a row. (Used for seperable kernels)
struct TEX_VS_OUT_7x1 {
  float4 Position : POSITION;
   float2 TapZero : TEXCOORD0;
   float2 Tap1  : TEXCOORD1;
   float2 TapMin1 : TEXCOORD2;
   float2 Tap2  : TEXCOORD3;
   float2 TapMin2 : TEXCOORD4;
   float2 Tap3  : TEXCOORD5;
   float2 TapMin3 : TEXCOORD6;
};

// A tex structure for 3x3 kernels excluding middle tap
// Is nice because it has 8 texcoords which fits fine in Shader model 2.0
// |1 2 3|
// |4 5|
// |6 7 8|
struct TEX_VS_OUT_3x3_NO_MIDDLE {
  float4 Position : POSITION;
   float2 Tap1  : TEXCOORD0;
```

```
    float2 Tap2  : TEXCOORD1;
    float2 Tap3  : TEXCOORD2;
    float2 Tap4  : TEXCOORD3;
    float2 Tap5  : TEXCOORD4;
    float2 Tap6  : TEXCOORD5;
    float2 Tap7  : TEXCOORD6;
    float2 Tap8  : TEXCOORD7;
};

TEX_VS_IN NormalVS(TEX_VS_IN input) {
  TEX_VS_IN output = (TEX_VS_IN)0;
  output.Position = mul(input.Position, mvpMatrix);
  output.Tex = input.Tex;

  return output;
}


TEX_VS_OUT_5x1 HorisontalVS(TEX_VS_IN input) {
  TEX_VS_OUT_5x1 output = (TEX_VS_OUT_5x1)0;
  output.Position = mul(input.Position, mvpMatrix);
  output.TapZero = input.Tex + float2(texelHeightOffset, texelWidthOffset);
  output.Tap1  = input.Tex + float2(texelHeightOffset, texelWidthOffset +
      texelWidth);
  output.TapMin1 = input.Tex + float2(texelHeightOffset, texelWidthOffset -
      texelWidth);
  output.Tap2  = input.Tex + float2(texelHeightOffset, texelWidthOffset + 2
      * texelWidth);
  output.TapMin2 = input.Tex + float2(texelHeightOffset, texelWidthOffset -
      2 *texelWidth);
  output.Tap3  = input.Tex + float2(texelHeightOffset, texelWidthOffset + 3
      * texelWidth);
  output.TapMin3 = input.Tex + float2(texelHeightOffset, texelWidthOffset -
      3 * texelWidth);

  return output;
}


TEX_VS_OUT_5x1 VerticalVS(TEX_VS_IN input) {
  TEX_VS_OUT_5x1 output = (TEX_VS_OUT_5x1)0;
  output.Position = mul(input.Position, mvpMatrix);
  output.TapZero = input.Tex + float3(texelHeightOffset,    texelWidthOffset
      , 0.0);
  output.Tap1  = input.Tex + float3(texelHeightOffset + texelHeight,
      texelWidthOffset, 0.0);
  output.TapMin1 = input.Tex + float3(texelHeightOffset - texelHeight,
      texelWidthOffset, 0.0);
  output.Tap2  = input.Tex + float3(texelHeightOffset + 2 * texelHeight,
      texelWidthOffset, 0.0);
  output.TapMin2 = input.Tex + float3(texelHeightOffset - 2 * texelHeight,
      texelWidthOffset, 0.0);
  output.Tap3  = input.Tex + float3(texelHeightOffset + 3 * texelHeight,
      texelWidthOffset, 0.0);
```

```
  output.TapMin3 = input.Tex + float3(texelHeightOffset - 3 * texelHeight,
      texelWidthOffset, 0.0);

  return output;
}

TEX_VS_OUT_3x3_NO_MIDDLE ThreeXThreeNoMiddleVS(TEX_VS_IN input) {
  TEX_VS_OUT_3x3_NO_MIDDLE output = (TEX_VS_OUT_3x3_NO_MIDDLE)0;

  output.Position = mul(input.Position, mvpMatrix);
  output.Tap1 = input.Tex + float2(texelHeightOffset - texelHeight,
      texelWidthOffset - texelWidth);
  output.Tap2 = input.Tex + float2(texelHeightOffset - texelHeight,
      texelWidthOffset);
  output.Tap3 = input.Tex + float2(texelHeightOffset - texelHeight,
      texelWidthOffset + texelWidth);
  output.Tap4 = input.Tex + float2(texelHeightOffset,     texelWidthOffset -
       texelWidth);
  output.Tap5 = input.Tex + float2(texelHeightOffset,     texelWidthOffset +
       texelWidth);
  output.Tap6 = input.Tex + float2(texelHeightOffset + texelHeight,
      texelWidthOffset - texelWidth);
  output.Tap7 = input.Tex + float2(texelHeightOffset + texelHeight,
      texelWidthOffset);
  output.Tap8 = input.Tex + float2(texelHeightOffset + texelHeight,
      texelWidthOffset + texelWidth);
  return output;
}

float blurWeight[] = {0.006, 0.061, 0.242, 0.383, 0.242, 0.061, 0.006 };

// Separated gaussian blur
PS_OUT GaussianBlurPS(TEX_VS_OUT_5x1 input) {
  PS_OUT output = (PS_OUT)0;

  float3 val = float3(0,0,0);
  val += tex2D(SourceSampler, input.TapMin3) * blurWeight[0];
  val += tex2D(SourceSampler, input.TapMin2) * blurWeight[1];
  val += tex2D(SourceSampler, input.TapMin1) * blurWeight[2];
  val += tex2D(SourceSampler, input.TapZero) * blurWeight[3];
  val += tex2D(SourceSampler, input.Tap1) * blurWeight[4];
  val += tex2D(SourceSampler, input.Tap2) * blurWeight[5];
  val += tex2D(SourceSampler, input.Tap3) * blurWeight[6];

  output.Color.rgb = val;

  //output.Color.rgb = float3(1,0,0);

  output.Color.a = 1;
  return output;
}

technique GaussianBlur {
  pass P0 {
    PixelShader = compile ps_2_0 GaussianBlurPS();
```

```
    VertexShader = compile vs_2_0 HorisontalVS();
  }
  pass P1 {
    PixelShader = compile ps_2_0 GaussianBlurPS();
    VertexShader = compile vs_2_0 VerticalVS();
  }
  pass P2 {
    PixelShader = compile ps_2_0 GaussianBlurPS();
    VertexShader = compile vs_2_0 VerticalVS();
  }
  pass P3 {
    PixelShader = compile ps_2_0 GaussianBlurPS();
    VertexShader = compile vs_2_0 VerticalVS();
  }
  pass P4 {
    PixelShader = compile ps_2_0 GaussianBlurPS();
    VertexShader = compile vs_2_0 VerticalVS();
  }
}



texture LowPassSourceTexture;
sampler2D LowPassSampler = sampler_state {
    Texture = (LowPassSourceTexture);
    MipFilter = NONE;
    MinFilter = Point;
  MagFilter = Point;
    AddressU = Mirror;
    AddressV = Mirror;
    MaxANISOTROPY = 1;
};

texture HighPassSourceTexture;
sampler2D HighPassSampler = sampler_state {
    Texture = (HighPassSourceTexture);
    MipFilter = NONE;
    MinFilter = Point;
  MagFilter = Point;
    AddressU = Mirror;
    AddressV = Mirror;
    MaxANISOTROPY = 1;
};


PS_OUT CollectHighAndLow(TEX_VS_IN input) {
  PS_OUT output = (PS_OUT)0;
  output.Color.r = tex2D(LowPassSampler, input.Tex).r;
  output.Color.gba = tex2D(HighPassSampler, input.Tex).gba;
  return output;
}

technique Passthrough {
  pass P0 {
    PixelShader = compile ps_2_0 CollectHighAndLow();
```

```
    VertexShader = compile vs_2_0 NormalVS();
    AlphaBlendEnable = false;
  }
}



float threshold;
float thresholdLevelWeight;

PS_OUT SoftThresholdingPS(TEX_VS_IN input) {
  PS_OUT output = (PS_OUT)0;
  // Low pass is not affected
  output.Color.r = tex2D(LowPassSampler, input.Tex).r;
  // Threshold high pass
  float3 val = tex2D(HighPassSampler, input.Tex).gba;
  output.Color.gba = sign(val) * max(abs(val)-threshold, 0);

  return output;
}

float3 hardThreshold(float3 input, float threshold) {
  // s will be 0 if we are within the threshold value, else 1
  float3 s = sign(max(abs(input)-threshold, 0));
  return input * s;
}

PS_OUT HardThresholdingPS(TEX_VS_IN input) {
  PS_OUT output = (PS_OUT)0;
  // Low pass is not affected
  output.Color.r = tex2D(LowPassSampler, input.Tex).r;
  // Hard threshold on the high pass
  float3 val = tex2D(HighPassSampler, input.Tex).gba;
  output.Color.gba = hardThreshold(val, threshold);
  return output;
}

technique SoftThreshold {
  pass P0 {
    PixelShader = compile ps_2_0 SoftThresholdingPS();
    VertexShader = compile vs_2_0 NormalVS();
  }
}

technique HardThreshold {
  pass P0 {
    PixelShader = compile ps_2_0 HardThresholdingPS();
    VertexShader = compile vs_2_0 NormalVS();
  }
}
```

Listing B.3: "Spatial filters"