

## MASTER

### Advanced ultrasound beam forming using GPGPU technology

van Bavel, Y.

*Award date:*  
2013

[Link to publication](#)

#### **Disclaimer**

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

#### **General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain



# Advanced ultrasound beam forming using GPGPU technology

*Master Thesis*

Yannick van Bavel

prof.dr. H. Corporaal  
ir. G.J.W. van den Braak  
prof.dr. J.J. Lukkien  
dr.ing. P.J. Brands

Committee:  
Eindhoven University of Technology  
Eindhoven University of Technology  
Eindhoven University of Technology  
Esaote Europe

Eindhoven, October 31, 2013

# Abstract

Ultrasound scanners are often used in medical diagnostics for visualising body parts without entering the body. An image is created by visualising reflections from an ultrasound pulse, transmitted into the body. Current scanners use a scanning which creates an image line by line, using focused pulses on each line separately. This method results in high quality images, but it limits the frame rate. In order to increase the frame rate, a different scanning method, called plane wave scanning, has to be used. With plane wave scanning a complete frame is acquired using a single ultrasound pulse on all channels.

However, plane wave scanning increases the computational load, because more data needs to be processed after each transmission. Therefore, more compute performance is needed to pave the way for high frame rate ultrasound imaging in to the kHz range, while a maximum frame rate of only 100 Hz is common today. GPGPU technology can deliver the needed performance requirements.

Esaote created a plane wave ultrasound research platform, allowing researchers to create their own applications for control of the scanner, receiving data, and for processing. In order to support researchers, with processing on a GPU, a high performance computing framework is created, which manages a compute pipeline on a GPU. The framework allows researchers to focus on the GPU implementations of their algorithms, instead of application development. The first pipeline implemented with the framework shows a 67 times improvement, compared to a naive CPU implementation, reaching a frame rate of 6.8 kHz at 8 mm image depth. The improvement gets bigger for larger image depths, because the GPU's peak performance is not reached at small depths.

When designing a system it is important to select a GPU, which meets the frame rate requirements. In order to assist system designers, with the selection of a GPU, a performance model is introduced. The model divides a kernel in parts and estimates the running time of each part. The running time of the entire kernel is predicted by taking the sum of all kernel parts. The results in this thesis show an error of 10% or less for the NVidia Fermi architecture.

# Contents

<b>Contents</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Related work . . . . .	2
1.2 Contributions . . . . .	3
1.3 Outline . . . . .	3
<b>2 Ultrasound Imaging</b>	<b>4</b>
2.1 Ultrasound scanning . . . . .	5
2.1.1 Line by line scanning . . . . .	5
2.1.2 Plane wave scanning . . . . .	6
2.2 Imaging pipeline . . . . .	6
2.2.1 DC removal . . . . .	7
2.2.2 Bandpass filter . . . . .	7
2.2.3 Delay and Sum reconstruction . . . . .	8
2.2.4 Hilbert transform and envelope detection . . . . .	9
2.2.5 Colour map . . . . .	9
<b>3 CUDA platform</b>	<b>11</b>
3.1 Hardware architecture . . . . .	11
3.1.1 Kepler architecture . . . . .	12
3.2 Programming and execution model . . . . .	14
3.3 Difficulties in CUDA programming . . . . .	15
<b>4 High Performance Computing Framework</b>	<b>17</b>
4.1 Overview . . . . .	17
4.2 HPC framework features . . . . .	19
4.3 Pipeline configuration . . . . .	19
4.4 Filter implementations . . . . .	20
4.5 Results . . . . .	22
<b>5 Performance Model</b>	<b>24</b>
5.1 Utilisation roofline . . . . .	24
5.2 Performance of SFU and FP . . . . .	27
5.3 Divide & Conquer . . . . .	29
5.3.1 Discarding shared memory accesses . . . . .	29
5.3.2 Prediction . . . . .	30
5.3.3 Results . . . . .	31
5.4 Discussion . . . . .	32
<b>6 Conclusion &amp; Future work</b>	<b>33</b>
6.1 Future work . . . . .	33

<b>Bibliography</b>	<b>35</b>
---------------------	-----------

# Chapter 1

## Introduction



**Figure 1.1:** Example of an ultrasound scanner of Esaote.

An ultrasound scanner, see figure 1.1, is a commonly used device in medical diagnosis. It can visualise parts inside the human body by using ultrasound, without entering the body. Ultrasound waves are transmitted into the body and will reflect on body parts. The reflections, or echoes, are received by the scanner and transformed into an image. These images are used to gather information about someone's health and can help to diagnose diseases.

In the past years ultrasound processing has shifted from dedicated hardware to software running on a PC, allowing ultrasound scanners to become faster, real-time, and smaller. Current implementations on a general Central Processing Unit (CPU) reach frame rates of 100 Hz. The next step is ultrafast (kHz range) ultrasound scanning, enabling more accurate tracking of movements and blood flow measurements. This step again increases the computational needs for ultrasound processing. The goal is to reach the current scanner's frame rate of 2 kHz at an image depth of 8 mm. The CPU cannot reach that throughput, therefore the usage of a Graphical Processing Unit (GPU) is investigated.

A GPU is a processor specialized in graphical operations. The hardware is specifically designed for altering graphical data with the purpose of displaying them on a screen. When GPUs with programmable pipelines were released, people started to become attracted in using the GPU for

general purpose computing. At first they had to convert their calculations to graphical operations with all the constraints of a graphics API. In 2006 NVidia released the Compute Unified Device Architecture (CUDA) [8], enabling general purpose computing directly on their GPUs. The work presented in this thesis uses the CUDA platform to enable ultrafast ultrasound processing.

A system designer has to take the system's requirements into account when designing a system. One requirement can be the performance of the system. In order to select a GPU he should know the performance it can deliver for a specific application. Therefore, a new performance model for CUDA GPUs is introduced, which takes specifications of the GPU and the algorithms into account. It can assist designers in choosing the right GPU for a system, which will meet the requirements.

## 1.1 Related work

GPUs have been used for ultrasound processing, since GPUs with programmable pipelines are available. In [12] a frequency domain image reconstruction algorithm for plane wave imaging is discussed using the OpenGL graphics API. At that time the speed up on the GPU was a factor two compared to a CPU, but the expectation was that the performance of GPUs would increase significantly in the near future.

Multiple discussion have been started, now GPU programming is available more easily. Eklund et al.[1] and So et al.[10], both conclude that a GPU is suited for many examples of ultrasound processing. The GPU has also other advantages, beyond high performance. The cost and energy efficiency of GPUs might make it appropriate for the portable, battery powered ultrasound scanners. However, the discussions notice the increased complexity in GPU programming.

A lot of research is performed on performance modelling for (GPU) architectures. The roofline model [13] states that an application's performance is either bounded by the memory bandwidth or by the computational performance. It uses the computational intensity to determine which bound is applicable for a kernel. It also introduces ceilings for suboptimal implementations, uncoalesced memory accesses for example. The roofline model gives a very rough estimation, because only floating point operations are taken into account.

In [11] the roofline model was extended by creating new rooflines: memory utilisation roofline and computational utilisation roofline. Both rooflines are calculated by taking the operations and memory accesses of a kernel into account. After the utilisation rooflines are calculated, they are applied in the same as the rooflines of the original model.

The analytical model described in [3] uses memory and computation warp parallelism to analyse the amount of execution overlap between warps. Although this model gives accurate results, it is hard to apply. Many parameters have to be put into the model and also benchmarking is required to gather some GPU parameters.

The Boat Hull model [5] adapts the roofline model by adding algorithmic classes. The usage of classes enables performance prediction before code development. One main difference with the roofline model is that the result of the model is the running time of an algorithm, instead of the performance. Also, additional instructions, offset, are introduced. They incorporate instructions, like address calculations, which were not taken into account by the original roofline model.

In order to verify a model one can use kernels from applications, but it can also be good to use microbenchmarks [14]. These are small programs, usually created for doing specific measurements. A second reason for microbenchmarking is to verify the theoretical performance of a GPU.

## 1.2 Contributions

This thesis shows that a GPU is a better candidate for ultrasound processing than a CPU. In order to utilize a GPU for ultrasound processing a high performance computing framework is created. The framework creates a computing pipeline at run time based on a configuration file. A developer only has to focus on the implementation of a filter step and not on pipeline management and connecting filters to each other.

The main contributions of this work are:

1. An advanced beamforming pipeline has been developed on the GPU, which can reach frame rate of at least 2 kHz at an image depth of 8 mm.
2. A new high performance computing framework is developed, in which all algorithms are implemented.
3. A new performance model is introduced for NVidia GPUs, which enables device selection based on performance requirements.

## 1.3 Outline

Chapter 2 gives an introduction in ultrasound imaging and the needed algorithms. The CUDA platform is explained in chapter 3. In chapter 4 the high performance computing framework is introduced, including performance results of the algorithms and a comparison between running times on GPU and CPU. The performance model is described in chapter 5. The conclusions and future work are given in chapter 6.

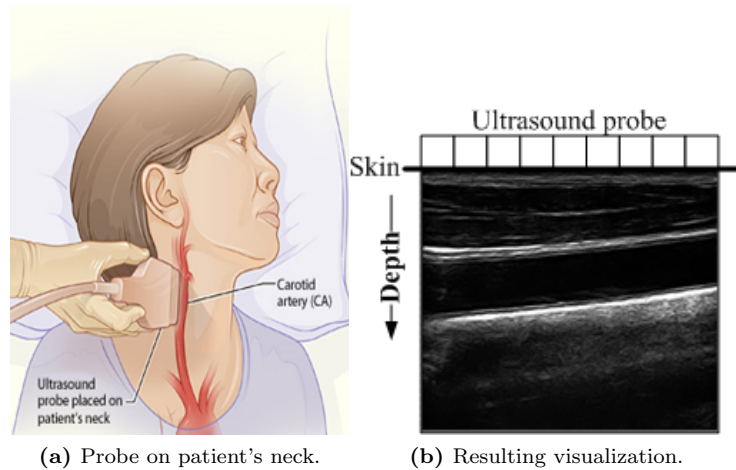


## Chapter 2

# Ultrasound Imaging

Ultrasound (US) is a sound wave with a frequency above the human hearing range of 20 kHz. One application of ultrasound is medical ultrasonography, which is a non invasive method for visualizing body structures as shown in figure 2.1.

An ultrasound probe, or transducer, contains multiple piezoelectric elements, called channels. Each channel can be used for transmitting and receiving of ultrasound waves. The probe is placed on the patient and an US pulse is sent into the body. The channels acquire the returning echoes by sampling at multiple time instances. Each time instance will correspond to a certain depth, because a sound wave has a certain velocity. The depth of a sample is an important feature, because it can be used to locate objects and to measure distances between them. An echographic image with a larger depth can be created when more samples are taken. The sequence of samples acquired by a single channel is called radio frequency (RF) signal or vector and will result in a single vertical image line. The combination of vectors from all channels is called a frame. Section 2.1 explains how ultrasound scanners acquire data and section 2.2 defines the steps needed to create an echographic image.



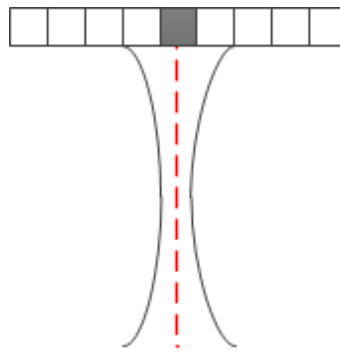
**Figure 2.1:** Medical ultrasonography applied on the carotid artery.

## 2.1 Ultrasound scanning

The way of acquiring US signals is called a scanning protocol. This section will explain a traditional scanning protocol (line by line scanning) and a protocol for ultrafast imaging (plane wave scanning).

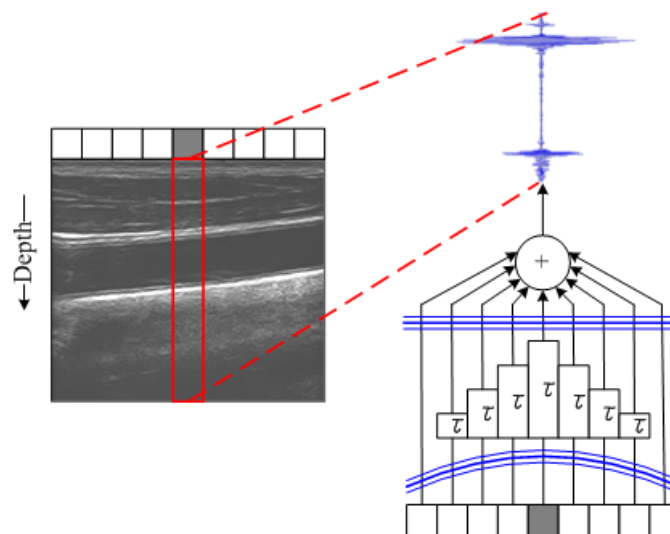
### 2.1.1 Line by line scanning

The traditional scanning method is line by line scanning. With line by line scanning the ultrasound pulses are focused on a single vertical image line, and often also focused at a certain depth. A small group of the channels is used to create a beam focused on a single vertical line, as shown in figure 2.2.



**Figure 2.2:** Focused ultrasound transmission.

All reflections from that pulse are delayed and added to form a single beam, figure 2.3. A frame is formed by transmitting a pulse for each line, one after each other. Although this gives high resolution images, it limits the frame rate because multiple firings are needed to construct a single frame. Moreover, a difference in time exists between the lines of a frame, because lines are formed after each other.

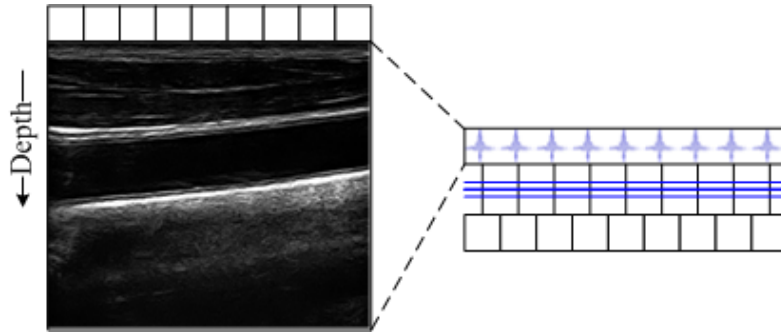


**Figure 2.3:** Vertical line by line beamforming.

### 2.1.2 Plane wave scanning

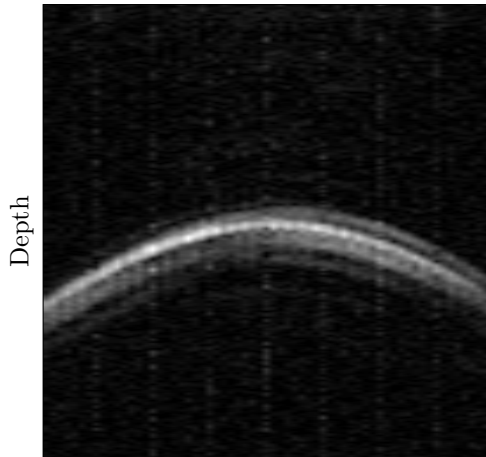
The scanning method with the highest frame rate is plane wave scanning. A higher frame rate allows tracking of fast moving objects. With plane wave scanning all channels are used for the transmission to form a planar US wave. This plane wave is not focused around a vertical line in contrast to line by line scanning.

Also all channels are used, at the same time, to receive the echoes. In figure 2.4 this is illustrated because every probe element creates an US signal. In theory the frame rate of this method is only limited by the sound travel time. The spatial resolution of the frames, however, is low, due to the unfocused transmission.



**Figure 2.4:** Plane wave scanning.

Another drawback of plane wave scanning is the need for image reconstruction algorithms in software, introducing additional computational needs. An echo is received by multiple channels, this causes point reflectors to show up as an arc as shown in figure 2.5.



**Figure 2.5:** Time of flight image of a single point reflector using plane wave scanning.

## 2.2 Imaging pipeline

Several steps are needed to reconstruct an echographic image from the plane wave scanning results:

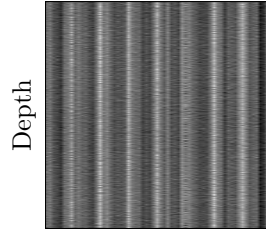
1. DC removal.

2. Bandpass filter.
3. Delay and Sum reconstruction.
4. Hilbert transform & envelope detection.
5. Colour map.

The processing steps are applied in a sequential, frame based pipeline. This section will explain these steps in more detail. Later on, in chapter 4, these steps are implemented on a GPU to meet the high throughput constraints of the plane wave scanning method.

### 2.2.1 DC removal

Each vector is sampled by a different receive channel, each introducing a different offset (DC). Figure 2.6 shows an unreconstructed echographic image without the DC removed, it is created by applying an envelope detection and a colour map. The different offsets of each channel cause vertical bars with different colours in the image. Also, the bars will spread over the image during the reconstruction stage, resulting in a white haze on the image. A DC removal applied on each column will remove these unwanted effects.



**Figure 2.6:** A unreconstructed echographic image without the DC removed.

The DC is removed by subtracting the average  $a_x$  of each vertical image line  $x$  from each sample  $r(x, y)$ :

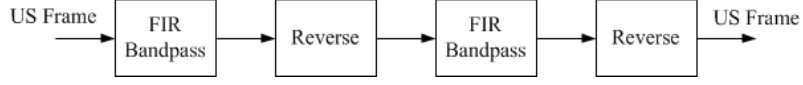
$$a_x = \frac{\sum_{y=0}^{M-1} r(x, y)}{M} \quad \text{with } M \text{ the number of samples in depth.} \quad (2.1)$$

$$r_{\text{DC removed}}(x, y) = r(x, y) - a_x \quad (2.2)$$

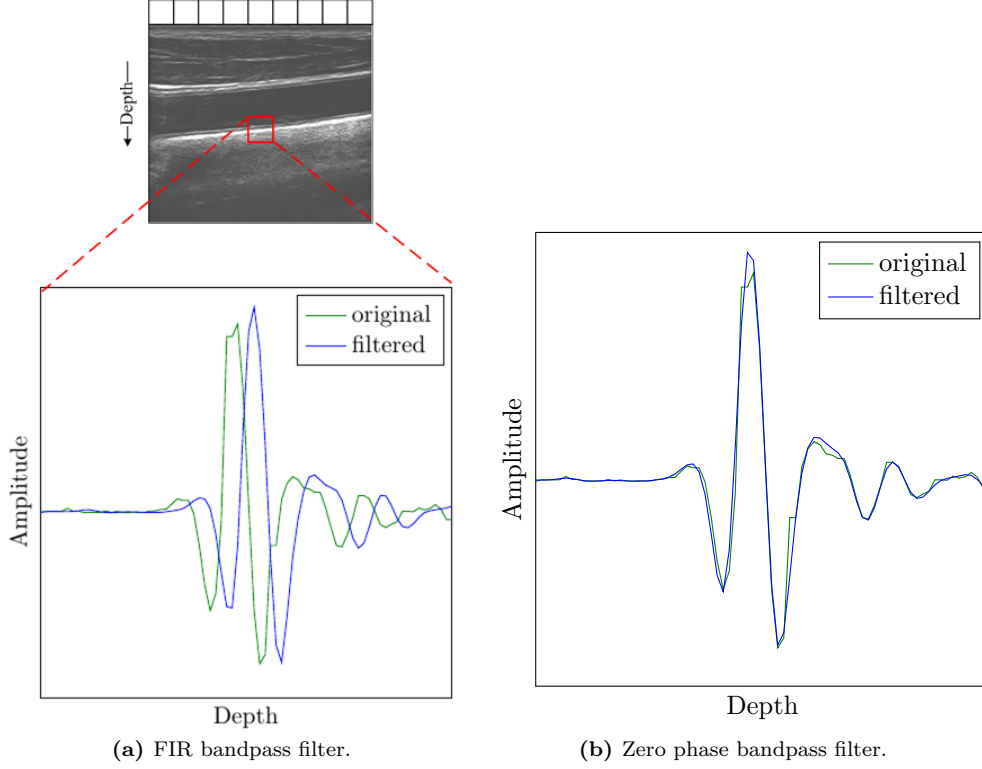
### 2.2.2 Bandpass filter

The received signals can contain noise from different sources, but only the returning echoes should be visualised. These echoes will have a frequency similar to the frequency which was used for the transmission. A FIR bandpass filter could be applied on each line of a frame, to remove the noise. However, a regular FIR bandpass filter will change the phase of signal, see figure 2.8a. A phase shift of the signal will change the depth of samples. In figure 2.8a the phase shift causes a change in the location of the peaks of the signal. The location of a peak corresponds to a certain depth of a strong reflecting object. So, a phase shift will change the location of object.

In order to retain the phase of the signal, the FIR filter is applied in the forward and reverse direction [9]. This means that the FIR filter is applied twice, while reversing the signal in between, see figure 2.7. As shown in figure 2.8b this method does not change the phase of signal. A second advantage is that the filter order is doubled, increasing the strength of the filter.



**Figure 2.7:** The zero phase bandpass filter pipeline.



**Figure 2.8:** The difference in phase of a regular bandpass filter and the zero phase bandpass filter.

### 2.2.3 Delay and Sum reconstruction

In figure 2.9 the travel path of a sound wave and one of its reflections is shown. However, the sound wave will reflect in a lot of directions. So, the reflections from a point  $s(x, y)$  are not only received by the channel directly above the point, but by all channels  $x'$ . The reflection  $r(x', y')$  is the sample acquired by a channel  $x'$  at a depth  $y'$ . In order to reconstruct the signal at  $s(x, y)$ , samples from all channels should be combined.

The reflection coming from  $s(x, y)$  will be acquired by a channel  $x'$  at depth  $d(x, x', y)$ :

$$d(x, x', y) = \sqrt{y^2 + (x - x')^2} \quad (2.3)$$

The signal at  $s(x, y)$  can be reconstructed by taking the sum of all reflections coming from that point  $s(x, y)$ :

$$s(x, y) = \sum_{x'=0}^{N-1} r(x', d(x, x', y)) \text{ with } N \text{ being the number of channels.} \quad (2.4)$$

### 2.2.4 Hilbert transform and envelope detection

The colour map is applied on the envelope of the signal, in order to improve the image quality. As with the bandpass filtering, it is necessary to maintain the phase of the signal. The amplitude envelope of an analytical signal has exactly that property.

First, the Hilbert transform is applied to derive the analytical representation of a real-valued signal  $x(t)$ . If  $X(f)$  is the Fourier transform of  $x(t)$ , then the Hilbert transform  $H(f)$  is given by:

$$H(f) = \begin{cases} 2X(f) & \text{if } f > 0, \\ X(f) & \text{if } f = 0, \\ 0 & \text{if } f < 0. \end{cases} \quad (2.5)$$

Equation (2.5) removes the negative frequencies from the spectrum of  $x(t)$ . This is allowed for real-valued signals, due to the Hermitian symmetry of the spectrum, as long as the complex-valued signal  $x(t) + j\hat{x}(t)$  is used in the time domain after removing the negative frequencies.

In the time domain  $\hat{x}(d)$  can be approximated by applying a phase shifting all-pass filter, see equation (2.6). The accuracy of this method can be tuned by changing the Hilbert transform window  $H_w$ . The application, presented here, uses a window size of 15.

$$\hat{x}(d) = \frac{2}{\pi} \sum_{m=1}^{H_w} \frac{x(t+m) - x(t-m)}{m} \text{ where } m = 1, 3, 5 \dots H_w \quad (2.6)$$

From the analytical signal  $x(d) + j\hat{x}(d)$  it is easy to calculate the amplitude envelope  $A(d)$  of the signal, equation (2.7). The envelope, obtained with this method, has no phase shift with respect to the original signal  $x(d)$ , as shown in figure 2.10, preserving the location of features in the signal.

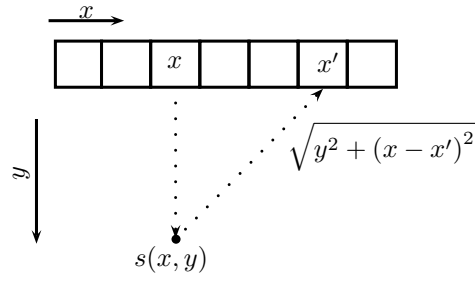
$$A(d) = \sqrt{x^2(d) + \hat{x}^2(d)} \quad (2.7)$$

### 2.2.5 Colour map

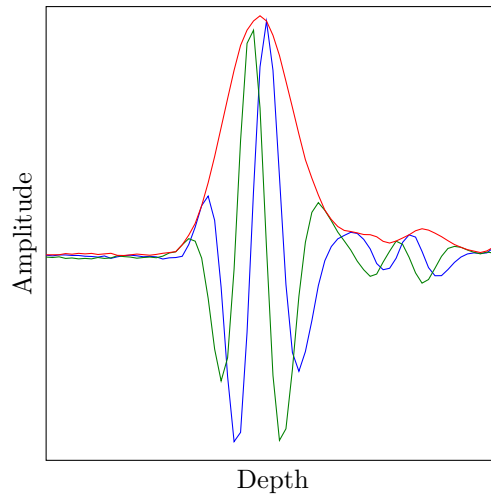
After steps 1-4 we have the reconstructed envelope of each signal in the frame. For the creation of the image a linear gray colour map is applied on the signal's amplitude. The amplitudes are clipped to the range of the colour map. Equation (2.8) shows how the gray scale value  $v(x)$  is calculated from a value  $x$ .

$$v(x) = \begin{cases} 255 & \text{if } x > \max, \\ 255 \cdot \frac{x}{\max} & \text{otherwise.} \end{cases} \quad (2.8)$$

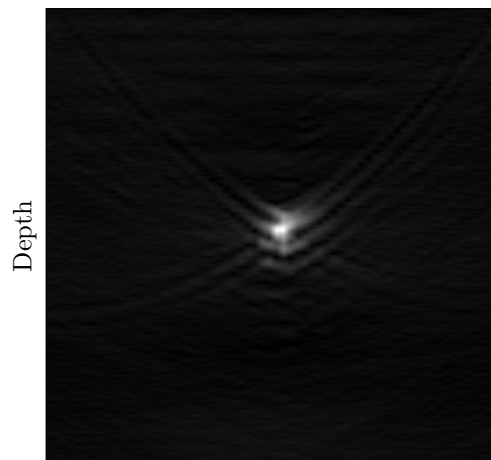
The colour map is applied on the envelope of the signal, see equation (2.7). So, values smaller than 0 do not occur in the data set. The result after colour mapping is shown in figure 2.11. The figure also shows the result of the reconstruction step.



**Figure 2.9:** Travel path of a sound wave, coming from  $x$  and one of its reflections, to  $x'$ .



**Figure 2.10:** A plot showing the real signal (green), its Hilbert transformed imaginary part (blue) and the envelope (red).



**Figure 2.11:** Reconstructed image of a single point reflector.

## Chapter 3

# CUDA platform

For general purpose computing on the GPU NVidia's Compute Unified Device Architecture (CUDA) platform is used. NVidia created the CUDA platform especially to ease general purpose computing on GPUs. It consists of a programming and execution model and a hardware architecture. Section 3.1 starts with the CUDA hardware architecture. In section 3.2 the programming and execution model are explained. The difficulties of programming with CUDA are explained in section 3.3.

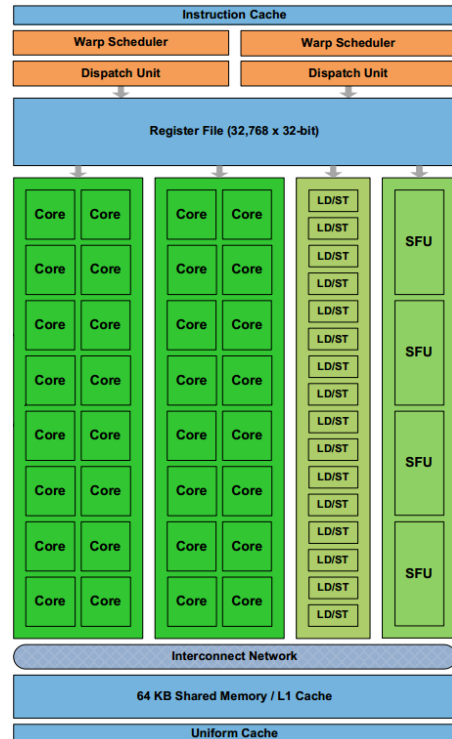
### 3.1 Hardware architecture

A CUDA GPU contains multiple independent streaming multiprocessors (SM). Figure 3.1 shows the streaming multiprocessor of the Fermi architecture [7]. The SMs have 4 execution blocks:

- 2 Computational blocks containing 16 cores.
- 1 Memory instruction blocks with 16 Load/Store units.
- A special function block with 4 special function units.

Each of these blocks operate in a Single Instruction Multiple Data (SIMD) fashion. All units inside a block execute the same instruction, but they apply it on different data elements. This type of parallelism is called Data-Level Parallelism. An algorithm should contain enough data-level parallelism in order to gain from this type of architecture.

The *Core* blocks are called Processing Element (PE) or CUDA core. The PEs execute floating point (FP) and integer instructions. The Special Function Units (SFU) calculate single precision FP transcendental functions, like square root, sine and cosine.



**Figure 3.1:** Fermi streaming Multiprocessor (SM) architecture.



Another important part of the SM is a memory space, called shared memory. CUDA defines multiple memory spaces. In this work two of them are used: global and shared memory. The global memory is the GPU's off chip memory, which is accessible by all SMs on the chip. The shared memory is a private, on chip memory inside each SM. It has a higher bandwidth and a lower latency than global memory. The high bandwidth is achieved by dividing the memory in modules, called banks. The banks can be accessed in parallel, if accesses go to different banks.

### 3.1.1 Kepler architecture

The previous section explained the CUDA hardware based on the Fermi architecture. The newer Kepler architecture [6] is quite similar to Fermi, but has an important difference. The number of processing units per streaming multiprocessor, on Kepler called SMX, increased a lot, as shown in figure 3.2. The number of CUDA cores, executing regular FP and integer instructions, increased with a factor 6 to 192. The warp schedulers also changed. They can now schedule an entire warp, 32 threads, every cycle, while on Fermi a scheduler only scheduled 16 threads.

A Kepler GPU has 4 warp schedulers. So, 128 threads can be scheduled every cycle, but there are 192 cores. In order to keep all cores busy, warp schedulers have to issue 2 independent instructions. However, a kernel should supply enough independent instructions to exploit this parallelism.

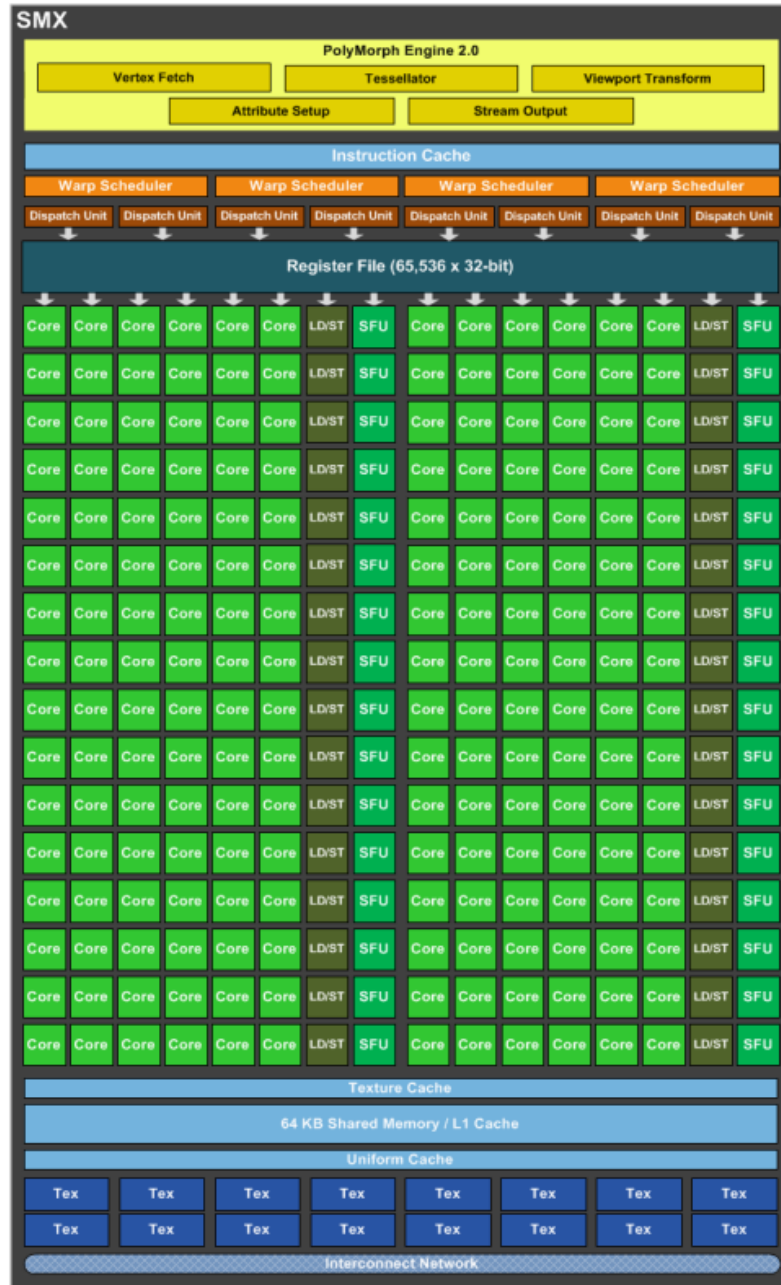


Figure 3.2: Kepler Streaming Multiprocessor (SMX) architecture.

## 3.2 Programming and execution model

The CUDA programming and execution model defines the notions kernel, thread, thread block and warp. In this section all these notions and their meanings are introduced.

CUDA C is an extension of the C programming language. It allows programmers to create functions, describing the functionality of a single thread, which is called a kernel. An example of a kernel is shown in listing 3.1. This example shows how a vector addition is implemented with CUDA. At run time multiple, parallel threads are created, each applying the programmed function.

```
--global__ void vectorAdd(float* a, float* b, float* c)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    c[index] = a[index] + b[index];
}
```

**Listing 3.1:** Vector addition kernel.

CUDA introduces three variables for identification of a thread: `blockIdx`, `blockDim` and `threadIdx`. The `blockIdx` and `blockDim` variables are used to identify the block, a thread belongs to, and to get the dimensions of a block. The identifier of a thread inside a block is stored in `threadIdx`. Each variable is a 3-dimensional vector type. So, a programmer has the possibility to create thread blocks that match the dimension of the input data. The combination of the three variables gives a unique identifier to each thread. The thread identifier is often used to select the data elements a thread should use. This is also shown in listing 3.1.

The programmer does not only define the number of threads, but also divides the threads in independent thread blocks. The thread blocks are divided over the SMs of the GPU and this assignment will not change, while a kernel is executing. Therefore, threads within the same thread block can share data using the shared memory of the SM. Figure 3.3 shows how blocks are divided on two different GPUs. The CUDA software distributes the thread blocks over the available SMs automatically. So, a program will gain from increasing the number of SMs, without interference of the program, as long as there are enough thread blocks.

The SIMD width is hidden for the programmer. The GPU divides the threads in groups, called warps, at runtime. The programmer has no influence on the creation of warps, they are formed by the hardware. A multiprocessor partitions consecutive threads into warps, with the first warp containing thread 0. All threads in a warp are scheduled by the warp scheduler at the same time. Although the programmer does not have to pay attention to the SIMD width, for reaching peak performance he should take the warp size, currently 32 threads, into account.

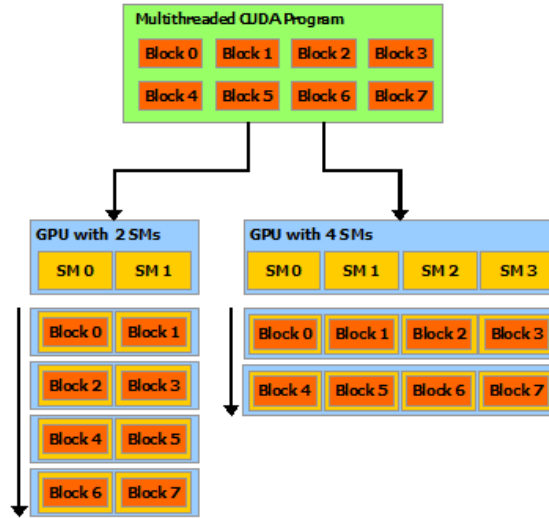


Figure 3.3: The assignment of blocks on the GPU.

### 3.3 Difficulties in CUDA programming

The GPU has a high memory bandwidth and high performance. However, the programmer has to follow some rules to reach them. This section explains some topics which are important when programming a GPU with CUDA.

#### Divergence

Control flow statements can cause threads to follow different execution paths. If this happens between threads inside a single warp, the entire warp will execute for each path. This is called warp divergence. A warp scheduler selects an instruction from a warp and schedules it for every thread in a warp. The SMs of a GPU are SIMD processors. Therefore, a warp can not execute two different instructions at the same time. The problem can become larger if a divergent path contains a synchronisation statement. Some threads will block until the rest of the warp is finished execution. The programmer should limit the number of execution paths within warps, to reach the peak performance.

#### Global memory access

A SM can combine the memory requests from a threads in a warp into a single, coalesced memory request. If the SM is not able to do so, than the memory requests will be replayed until all all threads performed their request. Off course this will increase the execution time and lowers the achieved global memory bandwidth. The programmer should take care that global memory accesses are aligned and sequential. Figure 3.4 shows two examples of uncoalesced memory accesses on different hardware versions (compute capability).

#### Shared memory banks

The shared memory of each SM has a very high bandwidth, because the memory is divided in parallel accessible banks. The number of banks is equal to the number of threads in a warp. So, each thread should access a different bank to reach peak performance. If multiple threads access the same bank, bank conflicts occur. This will cause replays of the memory requests, lowering the achieved bandwidth.

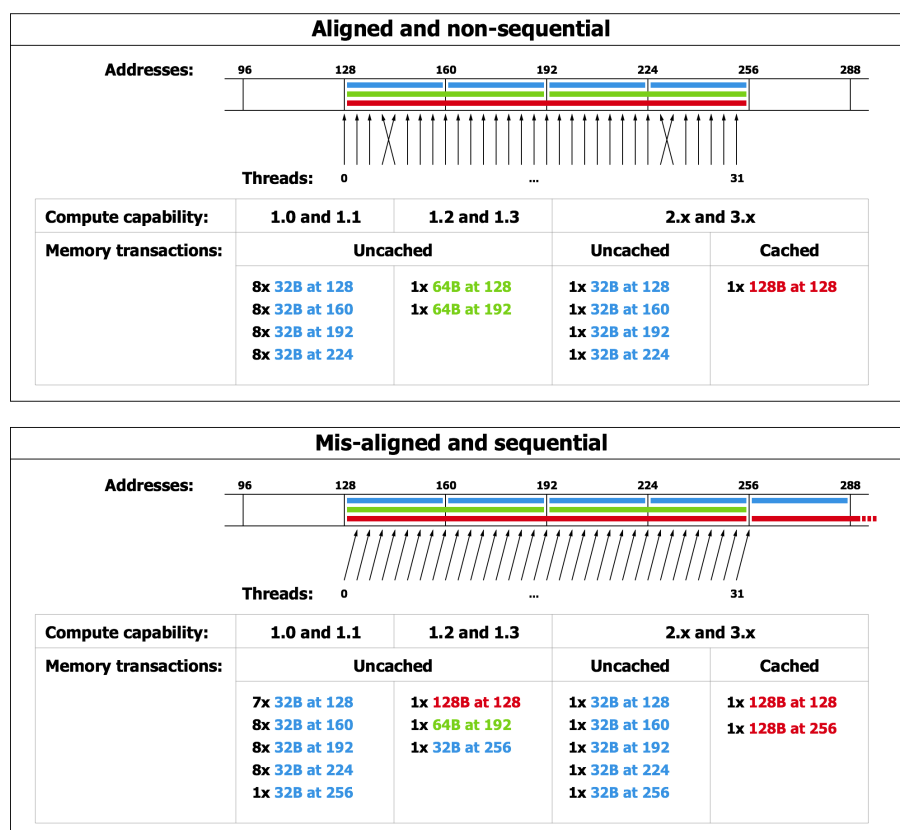


Figure 3.4: Example of uncoalesced global memory accesses.

## Chapter 4

# High Performance Computing Framework

A lot of research in medical ultrasonography is done off-line, using large datasets and applications like MATLAB. The step from off-line processing to a real-time application is often big. An application has to contain additional code, like controlling data acquisition, which has nothing to do with the algorithms. Furthermore, with increasing computational load, more processing power is required. A GPU is one solution to supply the required processing power, but this introduces more complexity for the programmer. In order to support researchers in utilizing the GPU a high performance computing (HPC) framework is created which aides them in the creation of real-time ultrasound processing applications.

In section 4.1 an overview of the HPC framework is given. An example of a pipeline configuration is given in section 4.3. The CUDA implementations of the algorithms from section 2.2 are described in section 4.4. The performance results of the GPU are given in section 4.5.

### 4.1 Overview



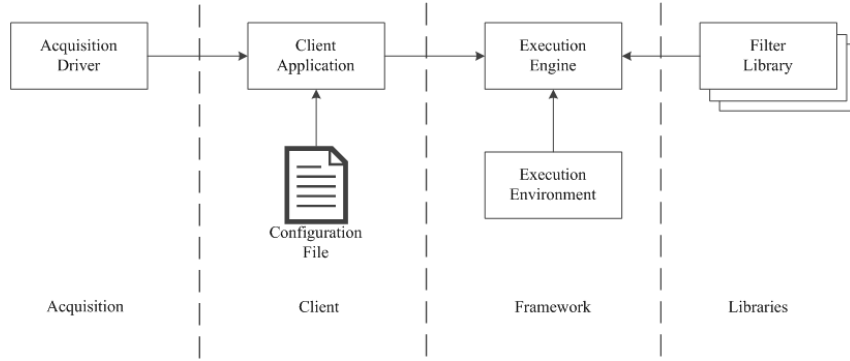
**Figure 4.1:** An Esaote portable ultrasound scanner.

Esaote, a company developing US scanners for medical diagnostics, created a plane wave ultrasound research system using a portable scanner (Figure 4.1). The scanner can be connected to a computer using a USB interface. Researchers can take full control of the scanner using a software library. The software also supplies the unprocessed US data to the researchers. With this system it is possible to develop reconstruction algorithms and other techniques to increase the resolution of plane wave imaging. Also other applications, object tracking or flow measurements, can be

investigated.

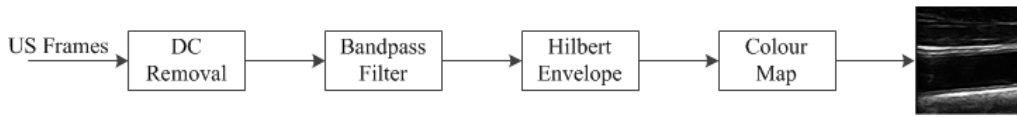
The Esaote research software is split into four parts, as shown in figure 4.2:

- Acquisition driver: controls the ultrasound scanner and the data acquisition.
- Client application: sends data from the acquisition driver to the framework and serves as a user interface.
- Framework: the high performance computing framework.
- Filter libraries: contain filter implementations.



**Figure 4.2:** Overview of the high performance computing framework.

The HPC framework consists of two modules: Execution Engine and Execution Environment. The Execution Engine is the main part of the framework. It controls the computing pipeline and provides the features described in section 4.2. The Execution Environment is an abstraction of the used computing device. Currently, only an environment for CUDA GPUs has been built, enabling the support of all CUDA GPUs. Currently, no environment is created for processing on the CPU. However, filters can execute code on the CPU.



**Figure 4.3:** An example of a computing pipeline.

The main task of the HPC framework is to create and manage a computing pipeline on a GPU. A pipeline is a sequence of filters which are applied on ultrasound (US) frames. An example is shown in figure 4.3. The user can create any pipeline, as long as it does not contain cycles, using a configuration file. The configuration file contains the desired filters, the connections between those filters and filter parameters. Section 4.3 discusses the configuration in more detail and also gives an example. The execution engine will dynamically load the filter libraries at runtime, create filter instances and configures them. The execution engine will do two things based on the definition of the connections:

1. Determine the execution schedule using topological sort [4].
2. Connect the outputs of the preceding filters to the current filter, just before execution.

The filter libraries are an important part of the software. It allows researchers to create, share and reuse filter implementations. The framework is able to load libraries which are not created by Esaote. Researchers can implement their own algorithms in CUDA and combine them in a pipeline with filters from Esaote and other universities. During filter implementation a filter developer only has to focus on the implementation of a filter. Other features, like synchronization, scheduling and data transfer, are done by the framework.

## 4.2 HPC framework features

The most important task of the HPC framework is the management of the computing pipeline. Besides that, the framework also has some additional features:

- Synchronization: between code running on the CPU and GPU. The user is able to synchronize when he wants to, but the framework will also synchronize automatically when necessary.
- Memory copies: the GPU has its own memory space which is not accessible by the host directly. The HPC framework maintains an input queue for the pipeline. The memory copies from and to the GPU are done by the framework. It also takes care of synchronization between execution and memory copy, if needed.
- Running time profiling: the framework measures the running times of the entire pipeline and of each filter separately. The filters are measured using the environment's timer, measuring only the kernel's execution time on the GPU.
- Bypass: allows disabling of selected filters at runtime as shown in figure 4.4. In this way the effect of a filter step can be checked without restarting the program.

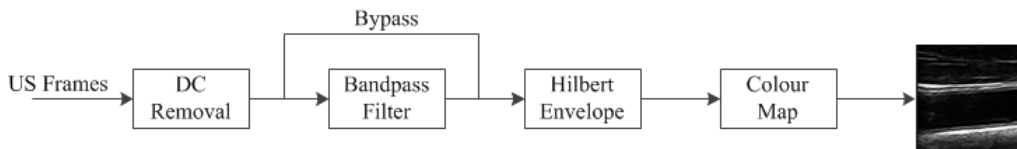


Figure 4.4: Example of filter bypassing.

## 4.3 Pipeline configuration

The framework uses a configuration file to create the computing pipeline at run-time. A scripting language is used to define the entire pipeline. Filter libraries are imported using the `import` keyword. Filter libraries can be distributed apart from the framework and enable sharing of filter implementations. With the `filter` and `view` keywords instantiations are created. It is allowed to create multiple instances of the same filter type. The name of the filter can be chosen by the creator of the configuration, but he should take care that filter names are unique. The name is used to identify filters when the connection between the filters are defined with the `link` keyword. The configuration shown in listing 4.1 will create the pipeline of figure 4.3.



```
environment cuda

import esaote_cuda
import esaote_recon_cuda

filter type dc_removal name dc_removal
filter type hilbert name envelope
filter type filtfilt name bandpass

view type colormap name echoview

link source frame@environment dest in@dc_removal
link source out@dc_removal dest in@bandpass
link source out@bandpass dest in@envelope
link source out@envelope dest in@echoview
```

**Listing 4.1:** Configuration example.

The configuration file does not only contain the filters and the connections between filters. It also has the possibility to contain filter specific parameters. This is shown in listing 4.2. The framework parses the parameter data and sends it to the filter instance, which is selected by the name of the filter.

```
set param coeffs@bandpass value float 3 [-0.0049 0.1007 0.4761]
set param window@envelope value int 1 [15]
set param amplitude@envelope value bool 1 [true]
```

**Listing 4.2:** Filter parameters in configuration file.

An application can perform different processing steps by changing the configuration file. There is no compilation step required to change the pipeline.

## 4.4 Filter implementations

All processing steps, mentioned in section 2.2, are implemented with CUDA. This section will discuss all these implementations.

### DC removal

DC removal consists of three steps:

1. Take the sum of each vertical vector.
2. Divide the sum by the number of samples in a vector. This results in the average (DC) of each vector.
3. Subtract the average of each sample.

The first step is implemented using the parallel reduction as explained in [2]. With reduction the problem size is reduced with each step, like a tree. Figure 4.5 explains this in more detail. The reduction approach needs synchronisation between each step, to make sure that each step is completely finished. If a thread would be created for each reduction step, then synchronisation would be necessary between blocks. The CUDA hardware does not support this and it would require multiple, slow, kernel starts. Therefore, a thread block is created for each vector as illustrated in figure 4.6, which will apply the reduction step multiple times.

Each iteration a thread will read a sample from global memory and add it to the previous result in shared memory. The data in global memory is stored in column major order and the memory is allocated using an aligned pitch. So, threads read the data aligned and sequentially.

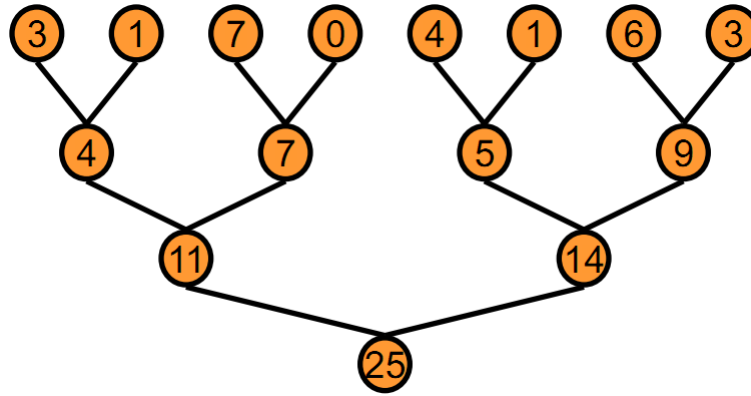


Figure 4.5: Tree based approach of reduction.

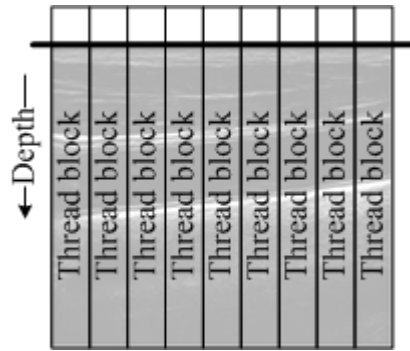


Figure 4.6: Thread block division.

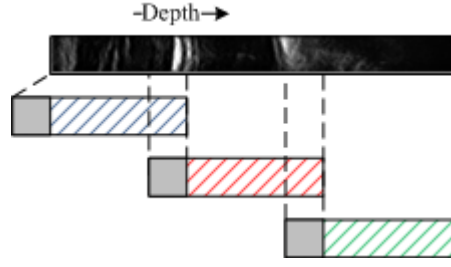
This ensures coalesced memory access.

When all data from global memory is read, the reduction process continues on shared memory. During that stage the number of active threads reduces each step by two. At the last stage of the summation only one thread is active, which produces the average of the vector. The last step (3) is to update all samples and write back the result to global memory. During this step all threads are active again, each updating multiple samples.

### Bandpass filter and Hilbert envelope

The bandpass filter and the Hilbert envelope have a similar implementation and will be discussed together in this section. Both algorithms compute a sum over a one dimensional window of neighbouring samples. They are applied on each vector separately, just like the DC removal. However, the vectors are divided over multiple thread blocks.

The windows of consecutive samples overlap each other. So, threads have input data in common. On GPUs without cache, this would decrease the performance of the kernel. By using shared memory this can be optimized. It has a higher throughput and it will reduce the number of requests to global memory. Each thread reads a single value from global memory and stores it in shared memory. It performs the computation using shared memory, after synchronization with other threads in a block. Fermi and Kepler GPUs will not benefit from this optimisation, because global memory accesses are cached on those architectures.



**Figure 4.7:** Thread block division of a single vertical image line for the bandpass kernel.

Figure 4.7 illustrates the thread block division of a single image line for the bandpass kernel. The threads in the gray part of a thread block are only used for data fetching from global memory to shared memory, but do not compute a result. This is needed because each thread only reads a single sample and needs samples from neighbouring threads. At the borders of a thread block there are not enough neighbours, so border threads cannot compute a result. The amount of border threads is a multiple of the warp size. This avoids divergence inside the warps, increasing the performance. Also, the additional threads terminate after the data fetch. So, an entire warp terminates and will not be scheduled any more.

The bandpass filter is applied in the forward and reverse direction. This means that the filter is applied twice, each time reversing the vector while writing the output. A vector is divided over multiple thread blocks and synchronisation is needed between these blocks. Therefore, the kernel is started twice.

#### Delay and Sum reconstruction

The calculations of the sound waves' travel time need a square root, see figure 2.3. Unfortunately, the throughput of a square root operation is not very high as it requires many instructions and uses the special function units. However, the travel times are the same for each frame. So, they can be pre-calculated and stored in global memory. Now a thread only needs to read the travelling time from memory and use it to address the input data. Pre-calculating the travel times improved the execution time of this filter 1.8 times.

#### Colour map

The colour map is implemented using the CUDA graphics interoperability with OpenGL. With OpenGL a frame buffer is created, which can be filled by a CUDA kernel. For each output pixel a thread is created, which reads the envelope of the signal and creates a gray scale colour value using equation (4.1). A colour is stored using vector type containing four values, allowing the output data to be sequentially written to the buffer.

$$r(x) = g(x) = b(x) = \begin{cases} 255 & \text{if } x > \max, \\ 255 \cdot \frac{x}{\max} & \text{otherwise.} \end{cases} \quad (4.1)$$

## 4.5 Results

The processing steps, mentioned in section 2.2, are implemented using the HPC framework on an NVidia Geforce GTX660Ti. The results, together with running times on an Intel Core i5-2300, are shown in table 4.1. The first remark is that the goal, a frame rate of 2kHz, is reached. The GPU can process the pipeline at 6.8kHz, while the CPU processes only 101 frames per second.

The running time of the reconstruction step is the largest. This step performs the most operations per thread and also produces the most global memory traffic. In order to reconstruct a single point a thread must read, for every column, a pre-calculated travel time and a sample from the input data.

A large speed up is obtained from CPU to GPU. One reason is that the CPU implementations were not parallelised. A theoretical increase in speed for the CPU would be a factor of 32, 4 cores and SIMD width of 8 single floating point values. Of course the real speed up will be less and the peak memory bandwidth might already be reached before the peak performance is reached, but the GPU would outperform the CPU even if a 32 times speed up on the CPU was realised.

Furthermore, it can be noticed that the algorithms scale better on a GPU than on a CPU. Table 4.1 also contains results from a larger data set, 128 columns with 128 mm depth. This effectively increases the number of samples in the data set with a factor of 32. The running time on the GPU increases 29 times, while on the CPU an increase of 240 times can be seen.

Frame dimension (columns x depth)	Kernel	GPU ( $\mu$ s)	CPU ( $\mu$ s)	Speed up
64 $\times$ 8 mm	DC Removal	9	126	14
	Bandpass	15	238	16
	Reconstruction	93	8525	92
	Hilbert envelope	9	629	70
	Colour map	20	328	16
	<b>Total</b>	<b>146</b>	<b>9846</b>	<b>67</b>
128 $\times$ 128 mm	DC Removal	119	3986	33
	Bandpass	196	7888	40
	Reconstruction	3672	2332597	635
	Hilbert envelope	136	22451	165
	Colour map	60	9273	155
	<b>Total</b>	<b>4183</b>	<b>2376195</b>	<b>568</b>

**Table 4.1:** Comparison of running times on GPU and single core CPU implementation, without use of SIMD instructions.

## Chapter 5

# Performance Model

Until now the work focused on using the GPU as a computing platform, which gave good performance improvements. However, it is not clear whether or not the best possible performance is reached. A performance model can be used to answer this question. For that reason this thesis introduces a new performance model. A second reason for performance modelling is platform selection. With a performance model the execution times on a GPU can be estimated without actually running the kernels on the GPU.

Multiple thread blocks can be active on a streaming multiprocessor (SM) of a GPU. However, the performance model presented here, does not take this into account. The measurements in this chapter are all performed using a single active thread block per SM.

Section 5.1 starts with an explanation of the utilisation roofline model. In section 5.2 of a kernel combining SFU and FP instructions is modelled. A new performance model is introduced in section 5.3. Section 5.4 will discuss the effect on the performance model of multiple active thread blocks.

### 5.1 Utilisation roofline

The utilisation roofline is an addition to the original roofline model introduced in [11]. Instead of using a single memory bandwidth and a single operation throughput, it introduces new bounds based on the instruction mix of a kernel and the used data sources. The utilisation roofline consists of two roofs: Computational Utilisation Roof (*cur*) and Memory Utilisation Roof (*mur*). The Computational Utilisation Roof is calculated by dividing the total execution time  $tc$  by the total number of operations  $CQ_{total}$ , see (5.3). The total execution time is the sum of the execution times of  $n$  operation types (5.1).  $CQ_i$  is the number of operations of type  $i$  with performance  $C_{ceiling}^i$ .

$$tc = \sum_{i=0}^{n-1} \frac{CQ_i}{C_{ceiling}^i} \quad (5.1)$$

$$CQ_{total} = \sum_{i=0}^{n-1} CQ_i \quad (5.2)$$

$$cur = \frac{CQ_{total}}{tc} \quad (5.3)$$

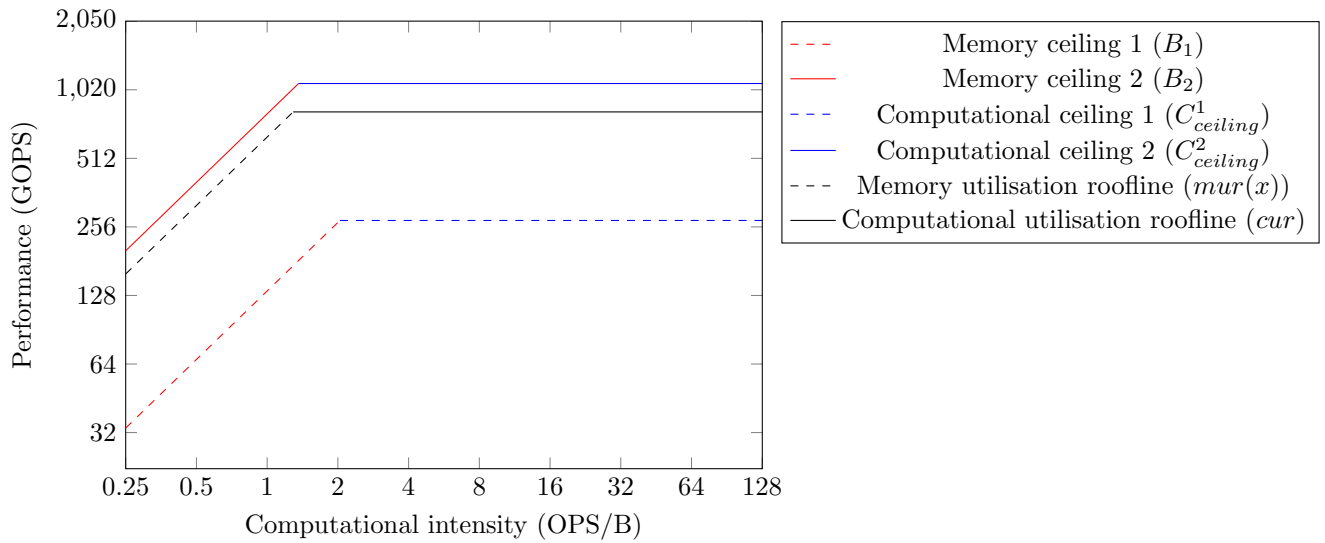
Equations (5.4) show that a similar approach is taken for the memory utilisation roofline.

$$\begin{aligned}
 tm &= \sum_{j=0}^{m-1} \frac{MQ_j}{B_j} \\
 MQ_{total} &= \sum_{j=0}^{m-1} MQ_j \\
 mur(x) &= \frac{MQ_{total}}{tm} \text{ where } x \text{ is the computational intensity.}
 \end{aligned} \tag{5.4}$$

The memory and computational utilisation roofline are combined into a single roofline by equation (5.5). It states that the performance is either bounded by computations or by memory transfers. Therefore, the performance of the kernel is the minimum of the two.

$$\text{utilisation-roofline}(x) = \min(mur(x), cur) \tag{5.5}$$

In figure 5.1 the roofline model and the utilisation roofline model are shown. The solid blue and red lines form the roofline for a NVidia GeForce GTX 470. The memory roofline is the bandwidth of shared memory. The dashed blue and red lines form a ceiling for operations, with a lower throughput, on global memory. By applying the equations of the utilisation roofline model a single roofline is created (black lines in figure 5.1).



**Figure 5.1:** Utilisation roofline model.

```

template <int OPS>
__global__ kernelMultiply(float* input, float* output)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    float result = input[index];

    #pragma unroll
    for(int i=0; i < OPS; i++)
    {
        result *= result;
    }

    output[index] = result;
}

```

**Listing 5.1:** CUDA microbenchmark example.

With microbenchmarks both the theoretical performance and the computational utilisation roofline are verified on a NVidia Geforce GTX470. An example of a microbenchmark is shown in listing 5.1. The results in table 5.1 show that the micro benchmarks comes very close to the theoretical performance. In all cases the error is less than 1%. The results in table 5.2 show that the cur is correct for operations handled by the same hardware pipeline. The fused multiply-add (FFMA), multiplication (FMUL) and addition (FADD) operations are processed by the same PEs, but with different throughput. The reciprocal square root instruction (RSQRT) is executed by the special function unit (SFU). The calculated performance (cur) of the instruction mixes is within the error ranges of the single instruction performance in table 5.1.

Operation	Theoretical performance (GFLOPS)	Measured performance (GFLOPS)	Error
FFMA	1088.6	1080.0	<1%
FMUL	544.3	540.0	<1%
FADD	544.3	540.0	<1%
RSQRT	68.0	67.5	<1%

**Table 5.1:** Theoretical and measured performance of a single instruction type on the NVidia Geforce GTX470 GPU.

Operations per thread			cur (GFLOPS)	Measured performance (GFLOPS)	Error
FFMA	FMUL	FADD			
10	2	8	725.8	717.9	1%
2	1	0	816.5	804.4	1%
0	2	8	544.0	539.8	1%

**Table 5.2:** Computational Utilisation Roof applied on the NVidia Geforce GTX470 GPU.

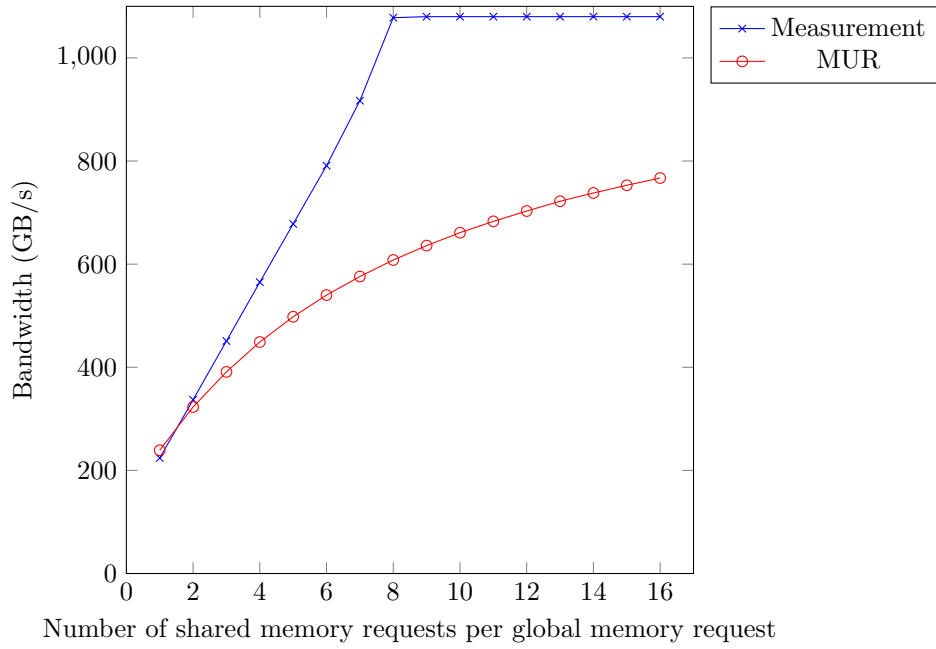
When the Computational Utilisation roofline is applied on instructions which can execute in parallel the model is not correct. Table 5.3 shows results of a kernel containing independent FFMA and RSQRT operations. These instructions are executed by different hardware pipelines and could run in parallel. The utilisation roofline ignores this and takes the sum of the execution times. Therefore the cur is too pessimistic as these two pipelines execute in parallel.

The memory utilisation roofline (MUR) is also verified using microbenchmarking. A kernel is created which contains independent global and shared memory requests. These requests will overlap, because the global and shared memory operate in parallel. The results in figure 5.2 show

Operation ratio (FFMA:RSQRT)	Prediction (GFLOPS)	Measurement (GFLOPS)	Error
1:2	99.0	101.2	2%
1:1	128.1	134.9	5%
2:1	181.4	202.3	10%
8:1	408.2	544.3	25%

**Table 5.3:** Computational Utilisation Roof applied on a kernel combining SFU and FP operations.

that the memory utilisation roofline is too pessimistic, because it ignores this parallelism.



**Figure 5.2:** The memory bandwidth combining global and shared memory accesses.

## 5.2 Performance of SFU and FP

Special functions and general floating point instructions are executed by different hardware pipelines, as stated in the previous section. These pipelines can run in parallel. So, the combined performance is bounded by the slowest pipeline.

Lets assume two pipelines  $M_0$  and  $M_1$  with throughputs  $P_0$  and  $P_1$ . If a kernel has  $O_0$  and  $O_1$  amount of operations executed by pipelines  $M_0$  and  $M_1$  respectively. The time spent on each pipeline,  $T_0$  and  $T_1$ , are calculated using equations (5.6) and (5.7). The combined throughput is calculated using the maximum of the running times of the pipelines, equation (5.8)



$$T_0 = \frac{O_0}{P_0} \quad (5.6)$$

$$T_1 = \frac{O_1}{P_1} \quad (5.7)$$

$$P_{0+1} = \frac{O_0 + O_1}{\max(T_0, T_1)} \quad (5.8)$$

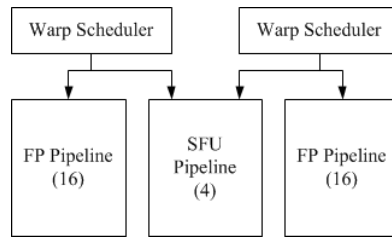
$$(5.9)$$

Equation (5.8) is applied on the SFU and FP pipelines of the GeForce GTX 470. The results in table 5.4 show that this model is too optimistic, because a larger error arises at a 8:1 ratio. So, there must be some bottleneck, preventing full parallel execution.

The SFU and FP pipelines are fed with instructions by the same schedulers, as illustrated in figure 5.3. These schedulers can either issue an FP instruction or an SFU instruction, but not both at the same time. This causes a bottleneck for the hardware. The three pipelines cannot be started at the same time.

Operation ratio (FFMA:RSQRT)	Prediction (GFLOPS)	Measurement (GFLOPS)	Error
1:2	102.1	101.2	<1%
1:1	136.1	134.9	<1%
2:1	204.1	202.3	<1%
4:1	340.2	335.2	2%
6:1	476.3	465.7	2%
8:1	612.4	544.3	13%

**Table 5.4:** Prediction of SFU and FFMA mixture, throughput based (5.8).



**Figure 5.3:** The CUDA architecture showing the dependency between the FP and SFU pipeline.

From the results in table 5.4 it can be noticed that upto a ratio of 6:1 the error is very small. This suggest that 6 FFMA operations are hidden by a single RSQRT instruction. A FFMA instruction is counted as 2 operations. So, effectively 3 FP instructions can be hidden by a SFU instruction.

In equations (5.10-5.12) this overlap is accounted for. First the time spent on RSQRT instruction is calculated ( $T_{\text{RSQRT}}$ ). The same is done for FFMA instructions ( $T_{\text{FMA}}$ ), only for each RSQRT instructions 6 FFMA operations are subtracted. The performance of the combination  $P_{\text{RSQRT} + \text{FMA}}$  is calculated by dividing the total number of operations by the total running time.

$$T_{\text{RSQRT}} = \frac{O_{\text{RSQRT}}}{P_{\text{RSQRT}}} \quad (5.10)$$

$$T_{\text{FMA}} = \frac{\max(O_{\text{FMA}} - 6 \cdot O_{\text{RSQRT}}, 0)}{P_{\text{FMA}}} \quad (5.11)$$

$$P_{\text{RSQRT} + \text{FMA}} = \frac{O_{\text{RSQRT}} + O_{\text{FMA}}}{T_{\text{RSQRT}} + T_{\text{FMA}}} \quad (5.12)$$

The prediction error decreases when equation (5.12) is applied on the same microbenchmark, as shown in table 5.5.

Operation ratio (FFMA:RSQRT)	Prediction (GFLOPS)	Measurement (GFLOPS)	Error
1:2	102.0	101.2	<1%
1:1	136.0	134.9	<1%
2:1	204.0	202.3	<1%
4:1	340.0	335.2	1%
6:1	476.0	465.7	2%
8:1	544.4	544.3	<1%
12:1	643.1	654.4	2%

**Table 5.5:** Prediction of SFU and FFMA mixture, based on an overlap factor of 6 (equation (5.12)).

## 5.3 Divide & Conquer

A CUDA kernel can often benefit from the high bandwidth of shared memory. Especially when threads have input values in common or need to share results. The following pattern is common for a CUDA kernel, when shared memory is used:

1. Data fetch: each thread reads data from global memory and stores it in shared memory. The number of shared stores is usually close to the number of global loads.
2. Computation: the kernel applies its computations on the data in shared memory.
3. Store result: the result of the computation is written to global memory.

The first two steps are separated by a synchronization instruction in most cases. Between steps 2 and 3 there is not always synchronization required, but there is always a data dependency. The model proposed in this thesis will divide a kernel based on the above observation, and will predict the running time of each part separately.

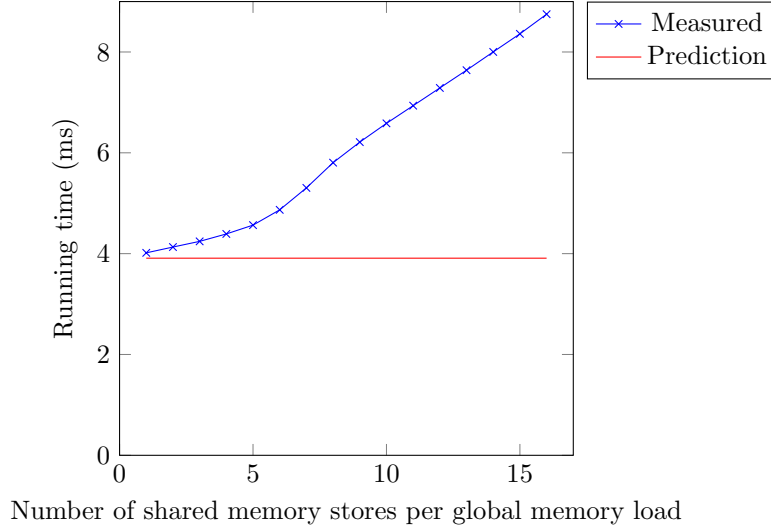
The model will only use the access time of global memory, when a step accesses both global and stored memory. The next section will show that it does not influence the divide and conquer (D&Q) model a lot. Section 5.3.2 will give more details about the prediction method. The benchmarks and results are discussed in section 5.3.3.

### 5.3.1 Discarding shared memory accesses

The model discards shared memory accesses when, in the same kernel block, also the global memory is accessed. Before we proceed with the clarification of the model, we show that this is allowed for small number of shared memory accesses. A microbenchmark was created with a single global memory load and a varying number of succeeding shared memory stores, which store

the data loaded from global memory. The delay, caused by the data dependency between these two steps, is hidden by creating enough threads.

Figure 5.4 shows the running time of the kernel with increasing number of consecutive shared memory stores. Up to 5 or 6 shared memory stores the running time only increases slightly, the increase is probably caused by the added instruction issues. The shared memory stores are hidden by parallel global memory accesses of other warps. So, upto 6 shared memory stores can be discarded without introducing a large error.



**Figure 5.4:** The running time of a kernel with a single global memory load and a varying number of shared memory stores.

### 5.3.2 Prediction

The model uses an approach similar to the Boat Hull model for the prediction of each kernel part, it is either compute or memory bounded. So, the running time is the maximum of the time spend on computations and the time spend on memory transactions. The computation time  $T_c$  is calculated using the cur and the overlap of SFU and FP. The overlap factor  $X$  is based on the mix of FP instructions. For FFMA instructions  $X = 6$ , otherwise  $X = 3$ , because a FFMA instructions counts as two FP operations.

$$T_{\text{SFU}} = \frac{O_{\text{SFU}}}{P_{\text{SFU}}} \quad (5.13)$$

$$T_{\text{FP}} = \frac{\max(O_{\text{FP}} - X \cdot O_{\text{SFU}}, 0)}{\text{cur}} \quad (5.14)$$

$$T_c = T_{\text{FP}} + T_{\text{SFU}} \quad (5.15)$$

The memory transaction time is calculated based on the total number of bytes accessed and theoretical bandwidth. The shared memory bytes are discarded, in case global memory is also accessed in the same block.

$$T_m = \frac{\text{Number of bytes}}{\text{Memory bandwidth}} \quad (5.16)$$

Up to now only the kernel's calculations and memory access time are taken into account. But a kernel contains additional instructions, like address calculations. This is modelled by adding a number of offset instructions to each kernel block. The type of instructions and the throughput of these instructions differ a lot. The offset time  $T_{\text{offset}}$  for  $O_{\text{offset}}$  is calculated using the lowest performance  $P_{\text{offset,low}}$  and the highest performance  $P_{\text{offset,high}}$ :

$$T_{\text{offset,low}} = \frac{O_{\text{offset}}}{P_{\text{offset,low}}} \quad (5.17)$$

$$T_{\text{offset,high}} = \frac{O_{\text{offset}}}{P_{\text{offset,high}}} \quad (5.18)$$

$$T_{\text{offset}} = [T_{\text{offset,low}}, T_{\text{offset,high}}] \quad (5.19)$$

The running time for a part of a kernel  $T_{\text{part}}$  then becomes:

$$T_{\text{part}} = \max(T_m, T_c) + T_{\text{offset}} \quad (5.20)$$

### 5.3.3 Results

The model is verified using four benchmarks on a GeForce GTX 470 and GTX 660 Ti. Table 5.6 contains the specifications of these GPUs. The GTX 660 Ti has the larger computational performance of the two, but the memory bandwidths are close too each other. This will cause predictions of memory bounded kernel parts to lie close to each other. Furthermore, the difference between the offset high and low performance is a factor 2 for the GTX 470, but a factor 5 on the GTX 660 Ti. The factor is higher for the GTX 660 Ti, because there is a larger difference in throughput of different instructions.

	GeForce GTX 470	GeForce GTX 660 Ti
FP performance (GFLOPS)	1089	3056
SFU performance (GOPS)	68	255
Offset high performance (GOPS)	544	1273
Offset low performance (GOPS)	272	255
Global memory bandwidth (GB/s)	134	144
Shared memory bandwidth (GB/s)	1089	1019

**Table 5.6:** Specifications of the NVidia GeForce GTX 470 and GTX 660 Ti.

Tables 5.7 and 5.8 give the prediction results for the NVidia GeForce GTX 470 and GTX 660 Ti of the divide & conquer (D&Q) model and the utilisation roofline model. The relative error of the D&Q model is calculated based on the average of the prediction interval. The predictions for the GeForce GTX 470 have an error of 10% or less and are more accurate than the utilisation roofline (UR) model.

Kernel	Measurement ( $\mu\text{s}$ )	D&Q Prediction ( $\mu\text{s}$ )	D&Q Error	UR Error
Band pass filter	189	161 - 213	1%	40%
Hilbert envelope	241	219 - 309	10%	50%
Tiled Matrix Multiplication (256×256)	201	173 - 190	10%	20%
Tiled Matrix Multiplication (2048×2048)	91008	85950 - 92616	2%	11%

**Table 5.7:** Prediction results for the NVidia GeForce GTX 470.

The predictions for the GTX 660 Ti have a higher error and it does not perform well on the band pass filter and Hilbert envelope benchmarks, when compared to the GTX 470. The GTX

660 Ti is not fully occupied, when only one block is active per SM. So, it is not able to reach peak performance. However, the prediction error for the D&Q model is less than for the UR model.

Kernel	Measurement ( $\mu$ s)	D&Q Prediction ( $\mu$ s)	D&Q Error	UR Error
Band pass filter	187	101 - 190	29%	40%
Hilbert envelope	260	125 - 280	28%	54%
Tiled Matrix Multiplication ( $256 \times 256$ )	176	170 - 198	4%	5%
Tiled Matrix Multiplication ( $2048 \times 2048$ )	72307	85313 - 96709	21%	17%

**Table 5.8:** Prediction results for the NVidia GeForce GTX 660 Ti.

## 5.4 Discussion

The measurements in this chapter only allowed kernels to run a single active block per SM, because the divide & conquer model does not take multiple active blocks per SM into account. In this section the effect of multiple active blocks on the model is shown.

A kernel was programmed and configured to execute with 768 threads on a GeForce GTX 470. The number of active blocks per SM was configured, by changing the shared memory configuration. In table 5.9 the results are shown. When two blocks are active on a SM, more latencies can be hidden which decreases the kernel's running time. The model does not take this into account, causing the error to increase.

Total blocks	D&Q Prediction ( $\mu$ s)	1 Active block per SM		2 Active blocks per SM	
		Measurement ( $\mu$ s)	Error	Measurement ( $\mu$ s)	Error ( $\mu$ s)
14	212 - 234	239	7%	239	7%
28	425 - 468	476	6%	391	14%
56	850 - 936	950	6%	751	19%
112	1699 - 1873	1896	6%	1458	22%

**Table 5.9:** The effect of multiple active blocks per SM on a GeForce GTX470.

The number of active blocks per SM can be calculated using the available resources of a SM and the resource request of a kernel. The effect of the amount of blocks per SM is harder to determine. More research is needed in order to add this to the divide & conquer model.

## Chapter 6

# Conclusion & Future work

This thesis has shown that the GPU is a good candidate for ultrasound processing. The performance goal of 2 kHz was reached with a frame rate of 6.8 kHz at 8 mm. Which exceeds the frame rate of the current scanner. The high frame rate enables the tracking of moving parts in the body. Also flow measurements will benefit from this high frame rate.

A framework was introduced which can create processing pipelines on the GPU dynamically. A filter developer will only have to focus on the filter implementation and the framework will take care of the rest. By using this framework an application can change the processing pipeline without compilation. Furthermore, the framework is able to import filter libraries from multiple sources, also from the research community, besides Esaote. So, the transfer from research to product is easier with the presented framework.

Also, a performance model was introduced which enables system designers to select a CUDA GPU based on the performance requirements. The accuracy is higher than the previous introduced utilisation roofline model, while it is still easy to apply. The prediction error for Fermi GPUs is lower than 15% and below 30% for the Kepler architecture. The model, together with the high performance computing framework, will ease the usage of a GPU for ultrasound processing.

### 6.1 Future work

The high performance computing framework currently only supports a single CUDA GPU. This makes it only suited for GPUs of NVidia. In order to support GPUs from different vendors, like AMD, it should also support OpenCL. A second change for the framework would be multi-GPU support. A system can have multiple GPUs available for computing. In order to increase the performance the framework could use multiple GPUs. It could run multiple instances of the same pipeline on the available GPUs or it could divide the filters over the available GPUs. The task of the framework would be to transfer the data between the GPUs and to start the filter execution on the right GPU.

For smaller systems an Accelerated Processing Unit (APU), combining a CPU and GPU on the same chip, might be more suitable. It can deliver a higher performance than a CPU, but consumes less power than a GPU. In case the framework is going to be used on smaller systems, than it might be good to look at applying it on APUs.

The running time of a kernel differs when the number of active blocks per SM changes. If the performance prediction model could predict the effect of the number of active blocks per SM, than the accuracy could increase.

Currently the performance prediction model does not take cache hits into account. Global memory accesses benefit a lot, when they hit the cache. The accuracy of the model could be increased, if cache hits are included. However, this will also make the model more difficult to use. The cache hits can be measured by a profiler, but this requires that a GPU is already available. Or a model for predicting the hit rate could be created.

A last issue of the model is, that it is created and tested with CUDA GPUs, just like the framework. When it would also be applicable for other (GPU) architectures, then designers have more options to choose from.

# Bibliography

- [1] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte. Medical Image Processing on the GPU Past, Present and Future . *Medical Image Analysis*, 2013.
- [2] M. Harris. Optimizing Parallel Reduction in CUDA.
- [3] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th annual International Symposium on Computer Architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.
- [4] A.B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.
- [5] C. Nugteren and H. Corporaal. The Boat Hull Model: Enabling Performance Prediction for Parallel Computing Prior to Code Development. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 203–212, New York, NY, USA, 2012. ACM.
- [6] NVidia. Whitepaper: NVIDIA GeForce GTX 680.
- [7] NVidia. Whitepaper: NVIDIA’s Next Generation CUDA Compute Architecture: Fermi.
- [8] NVidia. CUDA C Programming Guide, 2012.
- [9] A.V. Oppenheim and R.W. Schaffer. *Discrete-Time Signal Processing*. Pearson Education, 2006.
- [10] H.K.-H. So, J. Chen, B.Y.S. Yiu, and A.C.H. Yu. Medical Ultrasound Imaging: To GPU or Not to GPU? *Micro, IEEE*, 31(5):54–65, 2011.
- [11] M. Spierings and R. van der Voort. Embedded platform selection based on the Roofline model. Master’s thesis, Eindhoven University of Technology, 2011.
- [12] T. Sumanaweera and D. Liu. *GPU Gems 2*, chapter Chapter 48: Medical Image Reconstruction with the FFT. 2005.
- [13] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, April 2009.
- [14] H. Wong, M.M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, 2010.