NTNU

Norwegian University of
Science and Technology

Parallel Techniques for Estimation and Correction of Aberration in Medical Ultrasound Imaging

Åsmund Herikstad

Master of Science in Computer Science
Submission date: July 2009
Supervisor: Anne Cathrine Elster, IDI
Co-supervisor: Thor Andreas Tangen, ITK

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

Problem Description

Aberration (also known as phasefront aberration) due to spatial variations in the ultrasound propagation velocity in tissue, destroys the focusing of the ultrasound beam and increases the side-lobe level. This leads to a reduction of both the spatial and contrast resolution, blurring the image. Aberration is mainly generated in the human body wall, which is composed of skin, fat, muscle and connective tissue. Typical velocities for tissue in the human body wall is 1448 m/s for fat, 1547 m/s for muscle, and 1613 m/s for skin and connective tissue. These constitute the largest sound speed differences in the human body. The effect of aberration may be compared to smearing petroleum gel on a camera lenses, which totally degrades the focusing quality of the lens.

Dr. Svein-Erik Måsøy has developed and algorithm for estimating and correcting for this difference in sound velocity. This master's thesis project will focus on how to parallelize his method for modern multicore processors and/or modern GPUs. It is of particular interest to look at how applicable the parallelization techniques developed are for today's standard GPU cards.

Assignment given: 26. January 2009
Supervisor: Anne Cathrine Elster, IDI

NTNU

Norwegian University of
Science and Technology

Master Thesis

Parallel Techniques for Estimation and Correction of Aberration in Medical Ultrasound Imaging

Åsmund Herikstad
asmund1@yahoo.com

Department of Computer and Information Science
Norwegian University of Science and Technology, Trondheim
(Norway)

July 2009

Supervisor
Dr. Anne C. Elster

Co-supervisor
Dr. Svein-Erik Måsøy

HPC Research Group

Abstract

Medical ultrasound imaging is a great diagnostic tool for physicians because of its noninvasive nature. It is performed by directing ultrasonic sound into tissue and visualizing the echo signal. Aberration in the reflected signal is caused by inhomogeneous tissue varying the speed of sound, which results in a blurring of the image. Dr. Måsøy and Dr. Varslot at NTNU have developed and algorithm for estimating and correcting ultrasound aberration. This algorithm adaptively estimates the aberration and adjusts the next transmitted signal to account for the aberration, resulting in a clearer image.

This master's thesis focuses on developing a parallelized version of this algorithm. Since NVIDIA CUDA (Compute Unified Device Architecture) is an architecture oriented towards general purpose computations on the GPU (Graphics Processing Unit), it also examines how suitable the parallelization is for modern GPUs. The goal is using the GPU to off-load the CPU with an aim of achieving real-time calculations of the correction filter.

The ultrasound image creation is examined, including how the aberrations come into being. Next, how the algorithm can be implemented efficiently using the GPU is looked at using both NVIDIA's FFT (fast Fourier transform) library as well as developing several computational kernels to run on the GPU.

Our findings show that the algorithm is highly parallelizable and achieves a speedup of over 5x when implemented on the GPU. This is, however, not fast enough for real-time correction, but taking into account suggestions for overcoming the limitations encountered, the study shows great promise for future work.

Acknowledgments

This thesis was completed at the High Performance Computing Group at the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU) in collaborations with the Department of Circulation and Imaging at NTNU's faculty of Medicine.

I Would like to thank Dr. Anne C. Elster for her supervision and help in completing this thesis. I would also like to thank Dr. Svein-Erik Måsøy and Ph. D. student Thor Andreas Tangen at the Department of Circulation and Medical Imaging at the Norwegian University of Science and Technology for their invaluable help and guidance in understanding the technical terms related to ultrasound aberration and discussion on wanted goals of the thesis. The support from, and discussions with Ph. D. student Jan Christian Meyer at the Department of Computer and Information Science at NTNU was also vital for the completion of this thesis. I am also grateful to my girlfriend, Elin, for her patient and her encouragements when I lost motivation. Thanks also goes to my fellow graduate students for their support and good ideas. Finally, I would like to thank NVIDIA for providing several of the graphics cards used to complete this thesis through Dr. Elster's membership in the Professor Affiliates Program.

Åsmund Herikstad

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Medical ultrasound imaging is used to look inside the human body without having to perform surgical operations. The great advantage of this procedure is that it is non-invasive and does not cause any side effects. The equipment needed is also relatively low cost and mobile and therefore widely available to physicians. Ultrasound is hence a good tool to detect and diagnosis patient illnesses. The limitation of the technique is that since the sound waves transmitted by the ultrasonic transducer travel at different speeds in differing tissue, delays and variations in the signal may develop. These aberrations of the signal can cause distortions and blurring in the final image.

The ultrasound signal is transmitted by a transducer which is made of many transducer elements, each transmitting its own signal. An algorithm developed by Dr. Svein-Erik Måsøy and Dr. Trond Varslot [1] attempts to correct aberration of ultrasound images caused by the body wall by estimating the aberration in the ultrasound signal. The algorithm estimates the time-delay and the amplitude variations by comparing each transducer element's signal to its neighbor's and thus computing how much each of them diverge. The result is a correction filter for each transducer element that is applied to the next transmitted signal to cancel the aberration effects of the body wall.

Graphics Processing Units (GPUs) are highly parallel devices primarily developed to process graphical data. However, recent developments in both software and hardware architectures have made computing more general computational problems on the GPUs possible. The NVIDIA Compute Unified Device Architecture (CUDA), enables the users to perform non-graphical computations on the GPU through use of a C language extension. Using the GPU, one is able to utilize the large amount of parallel processing power to

solve algorithms that can be adapted to it at greatly improved speeds.

1.1 Outline

This master thesis has the following structure:

Chapter 2 and 3 presents the background for the thesis. Chapter 2 examines the process of creating ultrasound images and how those images are distorted by aberration of the ultrasound signal. Chapter 3 will present the GPU, its history, and how it may be applied to solve parallel computational problems.

Chapter 4 shows how Måsøy's algorithm proposes to counter the problem of ultrasound aberration and the process taken to estimate and correct the aberrated signal. It also describes and explains the code of the program created to perform the algorithm on the GPU.

Chapter 5 introduces the test case used to benchmark the program as well as the results from the benchmark. The chapter discusses the implications of the results and looks at possible reasons for their outcome. It will also look at what can be done to improve the results.

Chapter 6 concludes this thesis and looks at what have been achieved and how well this meets the goal of the thesis. The chapter also lists some future directions that the work can be take.

Chapter 2

Ultrasound Imaging

The next two chapters highlights the foundations for this master thesis. This chapter presents the basics of ultrasound imaging and associated aberration. The following chapter examines the GPU and its history is examined as well as general computations on the GPU with CUDA architecture and C for CUDA programming language extension.

Ultrasound imaging, also known as medical ultrasonography [16][17], is the process of sending ultrasound waves with a frequency between 2 and 18 MHz, into the human body and measuring the reflected waves, i.e. the echo, from the organs of the body. The echoes are then transformed into an image based on the delay from the time the wave was transmitted until the echo is received. A shorter time period means the echo is from a point closer to the transmitter than a long period.

2.1 Advantages and Disadvantages

Probably the greatest advantage of using ultrasound is its noninvasiveness. One can get good images of the internal structure of a body without having to perform surgery. Ultrasound has no known long-term side effects and also permits real-time imaging. The equipment needed to create ultrasound images is much smaller and more flexible than other currently used imaging techniques.

On the other hand, ultrasound has a distinct disadvantage, the quality of the images can be degraded in some situations. Also, it is virtually impossible to get any image through bones and air cavities. An ultrasound image through the lungs will make the image opaque. Indeed, even skin,

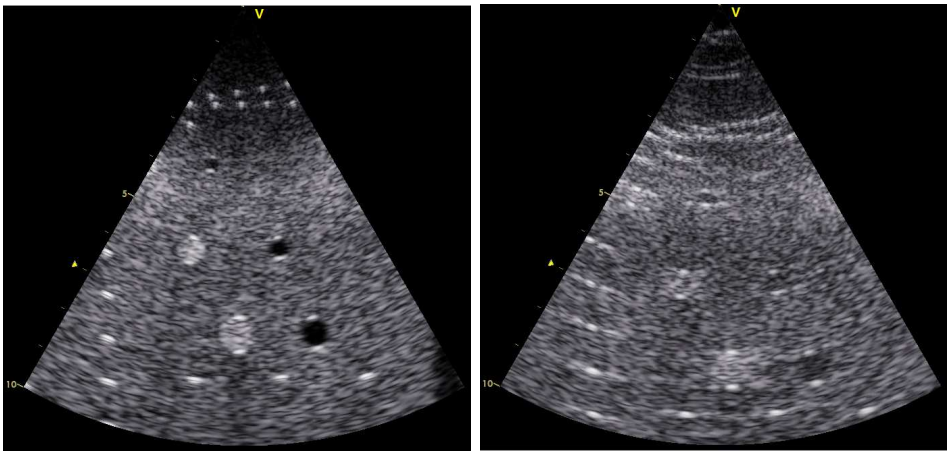


Figure 2.1: On the left is an unaberrated ultrasound image and one the right is an image with aberration. Reprinted with permission from Måsøy [1].

fat and muscle have quite an impact on image quality. This is especially true for obese patients, where the thicker body wall greatly affects the image achieved. This is clearly evident from Figure 2.1, which shows an ultrasound image of a test setup with no aberration on the left, and through a silicon model of the skin and muscles of the abdomen on the right. If the effect of some of these obstacles could be cancelled out, ultrasound imaging would become an even greater tool for imaging the human body.

2.2 Image Formation

The setup for performing ultrasound imaging consists of a ultrasonic transducer, a combined ultrasound transmitter and receiver, and a computer that prepares the signal to be sent and processes the received echo. As can be seen in Figure 2.2, the transducer is actually an array of transducers, which each can transmit and receive a sound wave. This gives the possibility of focusing the sound waves. The waves are focused at the desired depth using phased array techniques, that is the sound waves are delayed differently by different elements of the transducer array. This produces an arc-shaped sound wave from the face of the transducer array, see Figure 2.3. By changing the delay applied to each element of the array one can achieve both variable focus depth and change the direction of the sound wave without moving the transducer itself.

The resulting echoes are collected by the transducer and the shape of

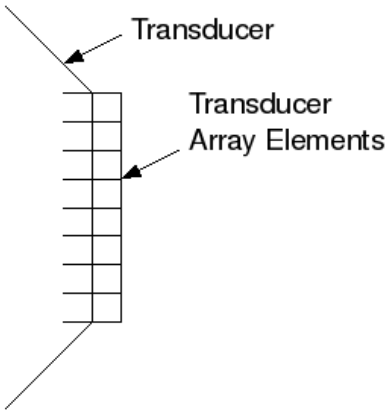


Figure 2.2: A ultrasonic transducer. A transducer is an array of transducer elements with each element transmitting and receiving ultrasound signals for imaging of the human body.

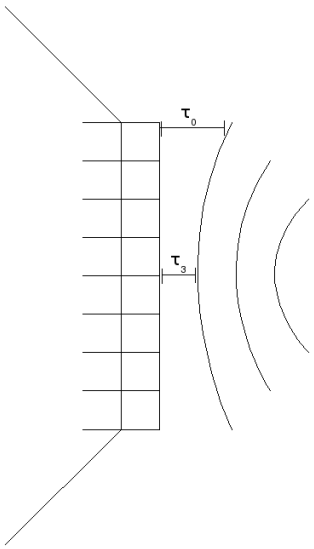


Figure 2.3: Transducer with focused waves. Using different delays (τ) for the different elements focuses the wave.

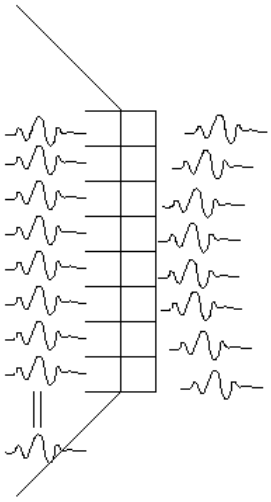


Figure 2.4: Removing the delays introduced when focusing the wave so that the waves can be aligned.

the wave is defocused, removing the delays that one purposely introduced to focus the wave. This aligns all the signals received on each element of the transducer array so that they can be summed, as seen in Figure 2.4.

By looking at when each echo arrives and the strength of the signal, one can plot an image of the body. A simple way of explaining the imaging process is by using a spreadsheet. By placing the transducer at the top of the sheet one can visualize transmitting the signals along the columns. When an echo from the body is received, the time delay is measured. A longer delay means further down the rows. The strength of the echo is translated to a grayscale color with white as the strongest echo and black for a weak echo. Coloring the cells of the spreadsheet with the corresponding colors results in a ultrasound image[16][17].

Figure 2.1 show ultrasound images which are the results of several measurements from the transducer. One image is a compilation of several ultrasound beams. Up to 250 beams may be used create one image. Each beam is one measurement as described above. When the measurement is completed, the focus and direction of the sound wave is changed and another wave is fired. This process is repeated until a complete sweep is recorded, and the final picture can be assembled. The actual sweep is performed electronically, and takes no more than a tiny fraction of a second. By performing multiple sweeps per second one can attain a live image of the body. Figure 2.5 illustrates how the sweep is done.

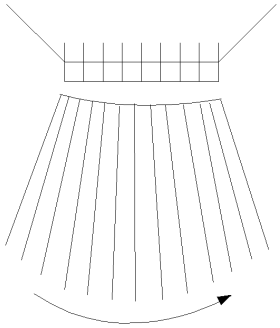


Figure 2.5: Several beams are needed for one ultrasound image.

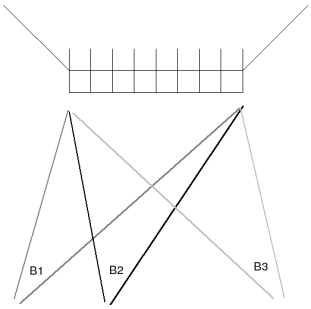


Figure 2.6: Each element sends a wave and the sum of all the waves is the beam.

Each beam is the result of summing the aligned signals from all the elements of the transducer array as seen in Equation 2.1.

$$b(t) = \sum_{k=0}^{n} y_k(t - \tau_k) \tag{2.1}$$

τ_k represents the delay that was introduced in the kth channel to focus the beam and $y_k(t)$ is the signal that is received at time t on the kth channel. $b(t)$ is the resulting beam. See Figure 2.6 for a graphical representation of this.

2.3 Ultrasound Image Quality

There are several factors limiting the quality of ultrasound images. One such limiting factor is absorption where energy from the ultrasound wave is transformed from waves into heat when the tissue is energized and that energy is lost. This effect increases with frequency. There is hence an effective

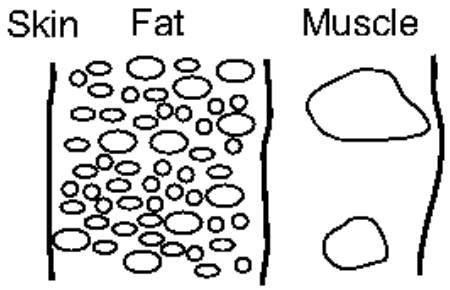


Figure 2.7: A illustration of the body wall. Reprinted with permission from Måsøy [1].

limit to how high frequencies can be used. This is also the reason why it is not possible to simply increase the power of the signal to get stronger echoes, as this would effectively boil the tissue.

Secondly, there are reverberations when waves are reflected internally in the tissue back and forth several times, before finally being picked up by the transducer. Reverberations are evident as a blurred tail in the image.

Finally, there is aberration, also known as phase front aberration [1], which is the distortion of the ultrasound wave caused by the different tissues that the ultrasound wave travels through. The theoretical model for focusing the ultrasound wave assumes that the velocity of the wave is 1540m/s. Any variations in this speed will cause delays of the wave, defocusing the beam and blurring the image. Måsøy [1] makes a claim about aberration, "Aberration is mainly generated in the human body wall, which is composed of skin, fat, muscle and connective tissue. Typical velocities for tissue in the human body wall is 1448 m/s for fat, 1547 m/s for muscle, and 1613 m/s for skin and connective tissue. These constitute the largest sound speed differences in the human body". This is also verified by Hinkelman et al. [19][39].

Aberration is the only focus of investigation and correction in this thesis.

2.4 Aberration

A typical human body wall consists of skin, connective tissue, muscles and fat. The speed of sound is 1448m/s for fat, 1613m/s for connective tissue and skin, 1547m/s for muscle. The fat is usually found as small lobules right beneath the skin, or as larger regions in the muscle layer. These fat lobules are held together by connective tissue. An illustration of the body wall can

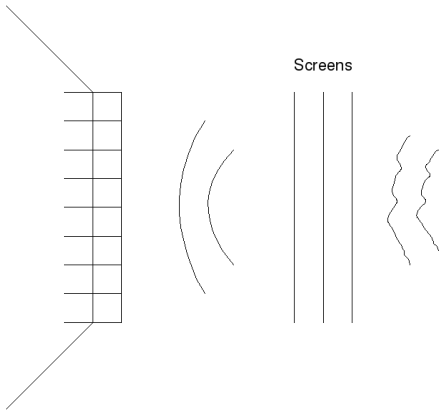


Figure 2.8: Screens as a model for the body wall aberrating the ultrasound signal. Each screen adds aberration to the wave.

be seen in Figure 2.7. As the ultrasound wave travels through the fatty region right beneath the skin, it experiences the largest differences in speed, in the transition from fat to connective tissue. This region creates the larges delays in the wave, and thus the most aberration in the image. Hinkelman et al. [19] examine the different layers of the body wall, and their effect on ultrasound imaging.

To get a better image, one would like to remove this aberration from the signal. To do this one first creates a model of the body wall aberration problem. Figure 2.8 shows a setup where the body wall is modeled by a number of screens. Each of these screens adds distortion to the wave and thus act in the same way as the body wall. If one considers this mathematically, each screen is a function that alters the original wave function, as expressed in Equation 2.2.

$$b_{A1}(t) = A_1 * y(t - \tau_F) \tag{2.2}$$

$b_{A1}(t)$ is the aberrated wave resulting from applying the filter A_1 to the original wave $y(t-\tau_F)$ (τ_F is the focusing delay). Combining more screens, or filters, one can write all these multiplications as a sum of the filters multiplied with the original wave as stated in Equation 2.3.

$$b_A(t) = \sum_{k=0}^{n} A_k * y(t - \tau_F) = A * y(t - \tau_F) \tag{2.3}$$

$b_A(t)$ is the resulting wave aberrated by all the filters. If A could be found, one could simply divide the aberrated signal with the filter and get the unaberrated signal $y(t - \tau_F)$.

2.5 Estimating and Correcting Aberration

Knowing what kind of structures that the waves propagated through, one could simply calculate what aberrations this layer would create and apply the inverse to the signal. However, since there is no way to know the structure of the body wall except by surgically looking at it, one must estimate the effect the body wall has on the signal. Måsøy looks at ways of doing this, and arrives at a method that does several iterations of an estimation which gives good results[1]. This section will examine this method.

Måsøy considers several filtering methods, and concludes that using a time-delay and amplitude filter gives adequate results.

A time-delay filter (*or phase filter*) works by adjusting the delay in arrival time at the transducer caused by aberration. This is similar to what is already done when focusing the wave. The beams from each element of the transducer array are delayed differently to converge on a focus and give a positive interference, resulting in a strong echo. When they arrive back at the transducer, they have to be aligned so that the sum of the signals of each element will create a strong and clear signal. If there is aberration that causes a phase shift, the wave at each element will arrive more or less delayed than it should, based on the delay introduced to focus it. This may cause interference which destroys the final signal. Figure 2.9 shows this effect.

An amplitude filter will remove peaks and troughs in signal strength caused by aberrations. If aberrated, the beams of different elements of the transducer array will have different values for the same samples and thus one of the waves will dominate when all the values are summed. The amplitude filter brings all the waves into the same range, removing this interference. A graphical representation of the amplitude problem is shown in Figure 2.10.

The filters look at neighbors of the element that is being filtered, and adjust their values accordingly. If the value of a sample is much higher or lower for an element than the average of those around it, the value of that sample is adjusted to conform with the other elements.

When these filters are faced with strong aberration, the resulting adjustments are incorrect. One possibility to improve these filters is to use their results in the generation of the next beam [1]. If a ”perfect“ wave is transmitted and returns aberrated, one can estimate how much aberration has happened and then apply the inverse of this aberration to the next wave. The next wave will then already be aberrated, but with the purpose of cancelling the effect of the aberration. When this second wave returns, it will then have cancelled out some of the aberration, permitting a new estimate of

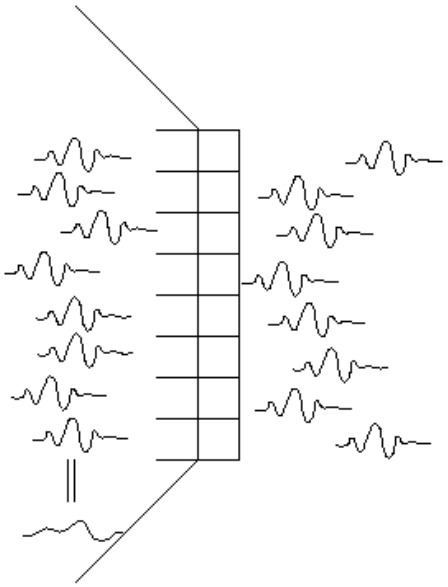


Figure 2.9: If affected by aberration waves at the different transducer elements the signals may have different time-delay and may end up cancelling each other out when summed.

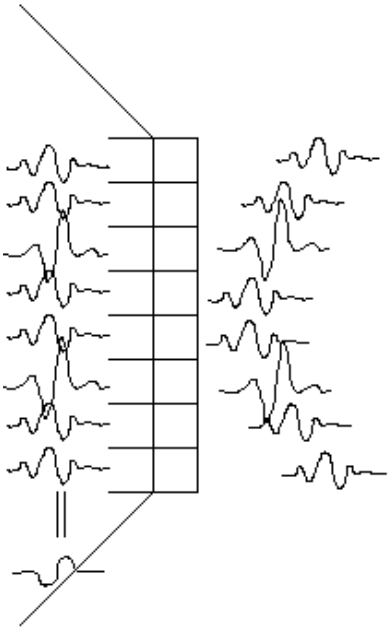


Figure 2.10: Signals with varying amplitude. If large variations in amplitude are present, the sum of the signals will be aberrated.

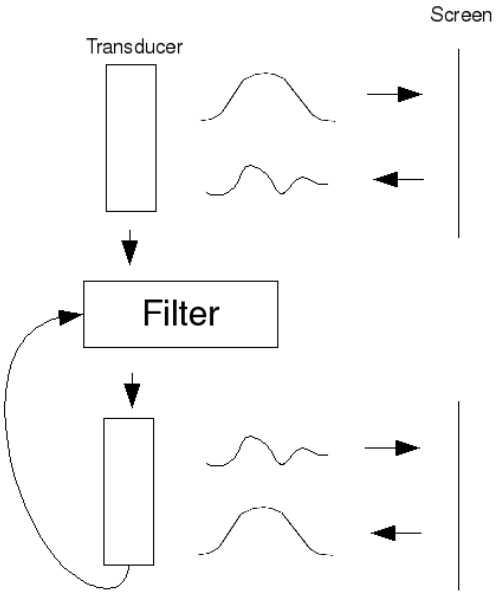


Figure 2.11: The filter is applied to the incoming signal and then the next wave is changed to give a better resulting wave. This iteration continues until some margin of error is reached. (The Gaussian signal is just for illustration and not the actual goal.)

the aberration to be calculated and applied to the next wave. This process is repeated until the wave that returns is unaberrated within certain error margins, improving the results of the filters even for strong aberrations.

The filter works by including information about the aberration from the previous beam. The aberration for one beam is similar to the previous beam as the body wall through which each beam is fired is very similar. The filter is thus able to find an estimate for the aberration for the next beam faster. The estimation of the aberration is an adaptive process, wherein several beams are transmitted in the same direction until the aberration for that beam has been computed to a satisfactory degree. Each step uses the computations from the previous beam, and the data from the final beam is used as the result of that direction. Figure 2.11 shows the iterative process of one beam direction.

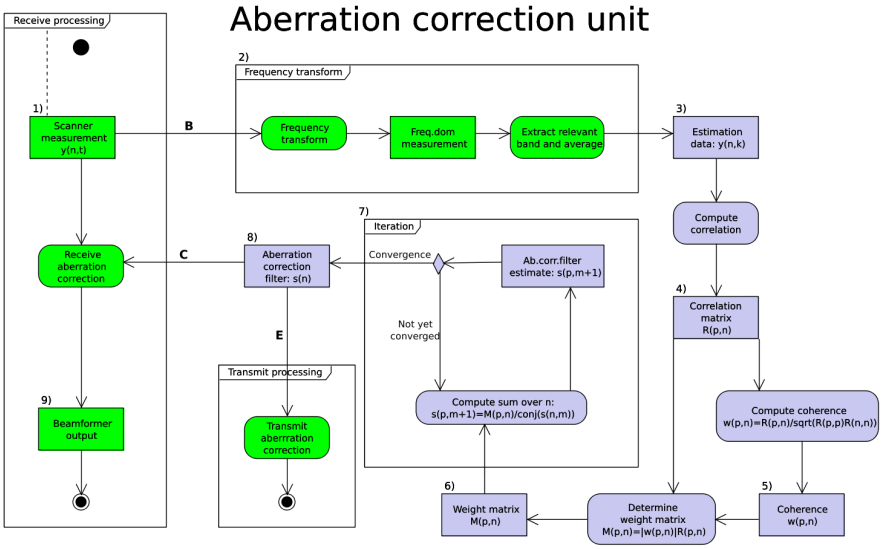


Figure 2.12: Diagram explaining the aberration correction process based on the algorithm by Måsøy [38]

Chapter 3

Moderns GPUs and NVIDIA CUDA

This chapter describes the main features of modern GPUs including the NVIDIA CUDA architecture and associated programming language extension.

3.1 GPGPU

GPGPU is an abbreviation for General-Purpose computation on the Graphical Processing Unit (GPU), that is, utilizing the GPU to complete tasks other than graphical ones. The goal is to increase the speed with which certain tasks can be completed. This has become a field of research because the GPU has a more parallel architecture than the CPU, and to take advantage of this parallel architecture, it is desirable to find new ways of representing problems in a parallel manner.

3.1.1 GPU

The GPU was originally a special function processor, meant to accelerate graphical computations. The GPU on a graphics card contains multiple computation units (cores) in a SIMD architecture [9][23], which performs parallel computations. This approach was developed to enhance image rendering, and create a better graphics experience for the user. An image shown on the screen is a matrix of millions of pixels which must be updated several times a second. Usually, one wants to execute the same calculations on large

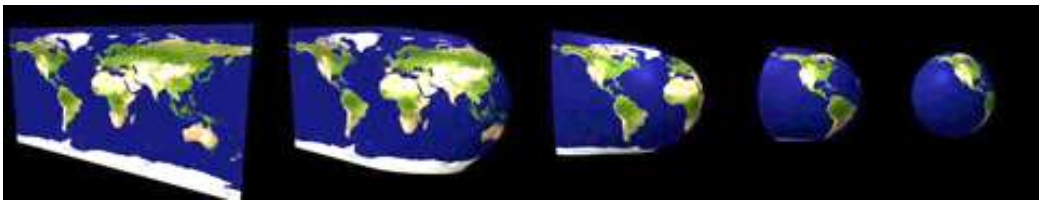


Figure 3.1: An example of using a GPU shader to wrap a texture to a 3D object. Reprinted from [26].

areas of this matrix, and therefore, the GPU is optimized to complete one or more sets of equivalent calculations in parallel. The result is an architecture that achieves high throughput, and has high degree of parallelism.

In the 1980s GPUs were mainly used to implement 2D primitives, in the 1990s they also handled 3D modelling. While GPUs improved and became ever faster, the ability to only use hardcoded instructions, shaders, in the chip became restricting. In the 1990s 3D games became more prevalent and it was of interest to calculate the values of each pixel based on the objects in 3D space. As the games became ever more complex the set of shaders that were available were not enough to express the wanted complexity.

The GPU manufacturers decided to implement programmable shaders, giving the programmer the ability to change the instructions performed on the graphics data. As Luebke and Humpfreys [24] explain, the advent of programmable shaders on the graphics card introduced a new computation model. Many researches started using the graphics card to do parallel computations, feeding data to the GPU as “images“, and rendering them with their own custom built shaders. Instead of rendering them to screen, however, the result was put back into memory and then read as the result of the calculation.

3.1.2 Parallelism

Recently, there has been a shift towards increased parallelism as most modern processors have become multicore. This is according to Asanovic et al. [27], a result of several factors, the main factors being referred to as the brick wall. The factors that make up the brick wall are:

Power wall: with the continuing increase of how many transistors per chip is possible, the power needed and the heat produced when turning on all the transistors is becoming a constraint. Too much heat will wreck the chip and using more power is also more expensive.

Memory wall: accessing memory consumes a larger part of a program runtime than actually executing the computations. When hundreds of computations can be performed in the time it takes to access memory, it is difficult to keep the processor occupied making transistors redundant.

ILP wall: instruction-level parallelism (ILP) is the attempt to extract parts of programs that can be executed simultaneously. An examples of this is out-of-order execution of instructions that are not dependent of each other and branch prediction to compute results of instructions dependent on a branch speculatively. There is however, a limit to how many independent instructions can be found and how many branches on can compute before the number of failed predictions become much larger than the number of correct predictions.

The turn to parallelism attempts to circumvent the brick wall. With many small cores one can have a finer-grained control over which transistors are turned on; this in turn gives less power consumption and less heat. Also, exposing the parallelism of the processor forces the parallel thinking on the programmer, resulting in more parallel programs and better utilization of the processor. Having more instructions that can be executed in parallel, helps hide the memory latency and the time that the processor is stalled waiting for memory accesses is decreased.

As the GPU is already highly parallel and represents several of these features, it is a natural area of interest.

3.1.3 Restrictions

There are several restrictions to the GPU, these are both its strengths and it weaknesses. The most common and important restrictions are presented.

A limitation of the GPU, as can be seen from Figure 3.2, is a much smaller amount of hardware set aside for caching and memory management than the CPU, and thus this is left to the programmer to handle. Efficiently managing memory can be a large part of writing applications for the GPU, but also enables great optimization options. The power of the GPU comes from being able to do many simple computations in parallel, meaning that more complex computations must be broken up into simpler ones.

The main challenge of the GPU is to harness the large number of cores available, and by always having enough computations executing to prevent the GPU from stalling while waiting for memory accesses. When accessing global memory, there is a 200-300 cycle latency, this is significant when compared to executing one instruction. A calculation, for example, takes a

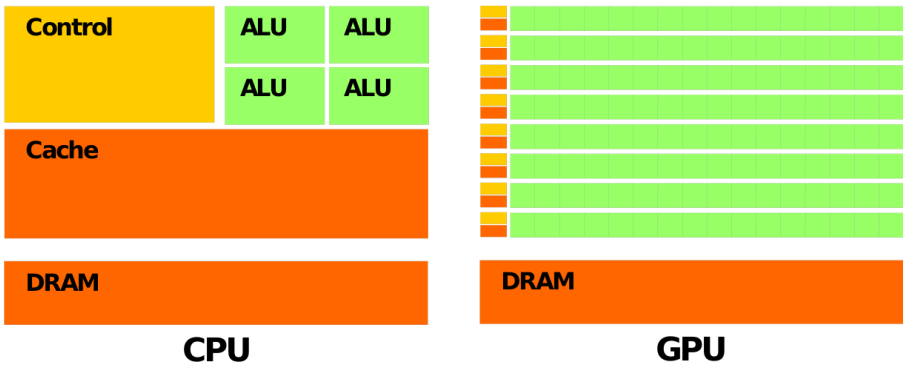


Figure 3.2: CPU uses much more silicon for Control and Cache than GPU which means that one have to handle memory management manually. Reprinted with permission from NVIDIA [10].

maximum of 4 cycles. If the GPU did nothing during these breaks, the efficient use of the GPU would be low. The CPU uses caching and prefetching to prevent this from happening, but the GPU does not. Instead, it compensates by executing other threads and their instructions in this waiting time. Therefore, having a large number of threads gives the GPU the possibility of computating while it waits for a memory load to finish, accomplishing both loading memory and more computations in the same time. If efficiently used, the GPU could use less time executing a program than it would take a CPU to do the same.

As can be seen from Figure 3.3, transfer of data between the host and the GPU is much slower than on-board memory access. It is therefore important to minimize copying between the GPU and the host. Some measures can be taken to optimize memory transfer between host and GPU. For example, aligning data increases efficiency by enabling transfer of sequential data blocks, so that only one memory access is needed per block of data transferred. Also, placing data in page-locked memory increases the speed since the bandwidth between host memory and device memory is higher if host memory is allocated as page-locked [10]. The $cudaMallocHost()$ function is used to allocate page-locked memory.

Another drawback of using the GPU is that branching is very inefficient. Since the GPU is a SIMD processor, it performs the same computations on the dataset regardless of how the code branches. Only after executing the code is the data that the branch would have created selected. For example, if a GPU was instructed to calculate a value for each element of an array and

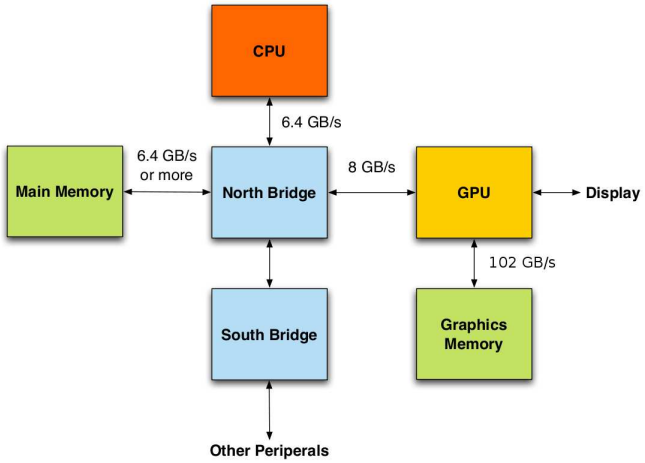


Figure 3.3: The GPU has high speed on-board access to memory, but much slower to the rest of the host system. Reprinted from [32], updated to reflect current values [29].

a special value for a certain element in that array, it would first calculate the values for the whole array and then also calculate the branching value for the whole array. Finally, it would select the branching value from that result set and return that for the single element, along with the non-branching values for the rest of the array. Therefore, branching is discouraged when computing using a GPU.

Finally, The GPU does not have as much special function hardware as the CPU, since these are seldom needed for shader programs. Modulo mathematics, for example, takes more cycles to complete on the GPU than on the CPU. The same is true for square root calculations. On the other hand, the GPU has special function hardware to compute cosine and sine functions, so this is faster than on the CPU.

3.2 CUDA

While GPGPU gives researchers the ability to use the massive parallelism of the GPU, writing shaders and expressing one's algorithms in graphical terms requires the user to be proficient in both his own discipline, graphics, and also quite a lot of advanced programming. This has limited the use of the GPU for general processing. Quite a few attempts at simplifying the process have been invented, such as RapidMind, BrookGPU, and Sh as

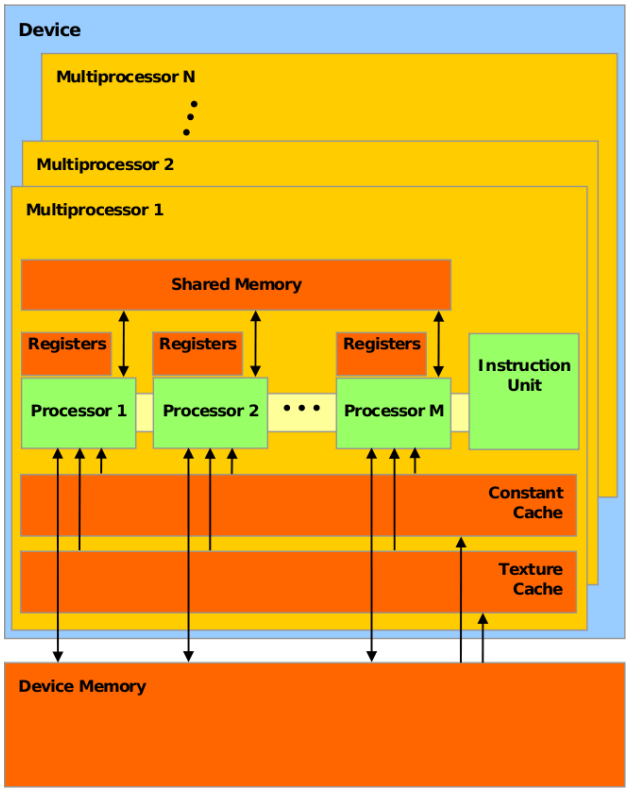


Figure 3.4: The CUDA architecture is divided into several Streaming Multiprocessors, each with 8 cores, a set of registers and shared memory available to executing threads. Reprinted with permission from NVIDIA [10].

well as Microsoft Accelerator [25]. These projects generally remove graphics terms from the programming by translating code written in C or C++ into GPU suitable code. However, they are still restricted to the calls that one can make using the graphics functions and they are therefore in themselves restricting in what one can achieve [5]. Both major graphics card manufacturers, ATI and NVIDIA, started an effort to bring GPGPU to researchers through their hardware. ATI developed Stream SDK which gives the programmer direct access to the native instruction set and memory of the GPU. NVIDIA developed the Compute Unified Device Architecture, CUDA, with the corresponding C for CUDA extension to the C programming language which also gives access to the parallel computational elements [15][14].

The architecture that CUDA describes is divided into sections. Figure 3.4 gives an overview of the CUDA GPU architecture. Each device have a number of Streaming Multiprocessors (SM), for example, the Tesla C1060 card has 30 SMs. The global memory is available to all threads, and is

currently 4 GB for the Tesla C1060. The global memory has a latency of 200-300 cycles, and therefore, common practice is to load data from global memory to shared memory, a faster on-chip memory, do computations and then move it back to global memory [10][22].

The following is a list of the features of a CUDA SM:

- 8 processing units (Stream Processors, SP).
- 16384 registers that are shared between the threads running on the SM.
- 16 KB of shared memory that has approximately the same latency as the registers.
- 8 KB of constant memory which is cached from the global memory.

There are several restrictions on how one can run programs on the CUDA architecture. Some are the result of choosing to devote transistors to processing power over memory management. This results in the programmer having to manually do memory management. There are limits imposed by the architecture: one can have a maximum of 1024 simultaneous active threads contexts per SM, up to 8 thread blocks per SM at one time, and if the total of 16384 available registers is exceeded, the excess will spill to local memory which has the same latency as global memory (200-300 cycles). There are also restrictions on how one may access shared memory. This memory is divided into blocks, and reads from the same blocks by more than one thread may cause collisions which result in the reads being serialized, thus taking more time. This problem has become less evident with newer versions of CUDA, which allow more combinations of reads by several threads. Because of these limitations, it is vital to optimize how one utilizes the GPU, both in memory reads and in kernel size and calculations. Both the CUDA programming guide [10] and Ryoo et al. [22] cover this more fully.

3.2.1 Memory Management

As can be seen in Figure 3.2, very little silicon is used for control and caching, and the CUDA architecture does not include transparent caching of all data that is used. The programmer has to take this into account for each individual program. There are several levels of memory in the architecture, ranging from the zero extra clock cycle access time of the registers, to the several hundred clock cycles access time of the global memory:

- Register - zero clock cycles

- Constant memory - cached, same access time as registers when cache-hit, global access time on cache-miss
- Texture memory - cached, especially good at 2D locality (hardware optimized)
- Shared memory - zero clock cycles when no collisions
- Local memory - part of global memory, 200-300 clock cycles access time
- Global memory - 200-300 clock cycles to access

Since there is such a large variation in speed of the different memory, and since the placement of data is entirely in the hand of the programmer, there is large potential for optimization (and inefficiencies). The most common approach to utilizing the memory of the GPU is to copy data to global memory from host memory and then let each thread copy a portion of the global memory into shared memory to be computed on by threads in a SM. This is usually done by instructing each thread to load a corresponding data value from global into shared memory, and then synchronize all threads so that they can all work on the correct dataset in shared memory. Another approach is to load data into global memory and utilize the texture caching or with constants like convolution kernels, load them directly into constant memory. Varying the configurations to achieve optimal behavior is a research topic in itself, and usually a unique configuration is needed for each program created.

Shared Memory

Shared memory is divided into banks, making it possible to read from each separate bank simultaneously. If two or more threads try to read from the same memory bank, this causes a bank conflict, and the conflicting calls are processes in serial, greatly reducing the effectiveness of the shared memory. It is therefore important to access memory in such a manner as to avoid bank conflicts.

Shared memory banks are organized as 32-bit words, i.e. the size of one float or an integer. A common approach to accessing the shared memory from a kernel is by letting each thread read a 32-bit word at a unique memory position in the data, avoiding bank conflicts. This can be expressed as $array[BaseIndex + tid]$, where tid is the thread ID. However, for large data sets, it is often necessary to have each thread access more than one memory location. This is often done using $array[BaseIndex + s * tid]$, where s is the stride, i.e. the number of elements to the next corresponding location to be

read by the thread (usually the number of threads). In this case, thread tid and thread tid+n will access the same bank whenever s*n is a multiple of the number of blocks. On the current architecture version, this will not happen if s is odd [10]. Therefore, optimizing shared memory is done by making sure threads do not cause bank conflicts.

Global Memory

When accessing memory on the graphics device, an important guideline is to maximize bandwidth usage. This is especially true for global memory, which has two orders of magnitude greater access time. To utilize the whole bandwidth available, a technique called coalesced reads and writes is implemented. The idea is to combine many accesses to global memory into one large access, and thereby utilize the whole bandwidth. An alternative would be to service each access separately, and achieve only a fraction of the available bandwidth. However, there are some preconditions which need to be met before the memory controller will coalesce an access.

Accesses will be coalesced when they fall into the same segment of size [10]:

32 bytes and if all threads access 8-bit words.

64 bytes and if all threads access 16-bit words.

128 bytes and if all threads access 32-bit or 64-bit words.

3.2.2 Thread Configuration

Even though the GPU has hundreds of computational units, supplying these with a fast memory accessible from all of them would require a lot of hardware set aside for connections and read/write collision prevention. This would be expensive, since one would need both more hardware and more advanced hardware. Most importantly, it would leave less space for the computational units [8].

The compromise reached is to limit the amount of fast memory, and make it accessible to only a portion of the computational units. The idea is to have a fast memory on the processor chip that the threads running on a streaming multiprocessor can share, and a slower off-chip global memory accessible for all threads. Ensuring that all threads that should be sharing memory are running on the same SM is important. To ascertain this happens, threads are divided into blocks, which let the GPU group threads together, and assign

groups to SMs. All threads in a block are guaranteed to run on the same multiprocessor and can thus communicate and synchronize with each other. Selecting the size of the blocks is done by the programmer as is described in the next section.

When executed, threads in a block are actually divided into smaller groups called warps [8][10]. This is to facilitate swapping of executing threads on the SM. The SM will switch to a new warp when one warp is stalled, for example while waiting for a memory load. Thus the SP still has threads to schedule even though some of the threads in the block are stalled. There are 32 threads in each warp. The computations are scheduled in half-warps, 16 threads. Warps correspond to the efficient memory access, if all threads in a warp access aligned data from memory, the access will match the requirements for coalesced access.

3.2.3 C for CUDA

C for CUDA extends the C programming language by adding function type qualifiers, function call syntax, some special variables, intrinsic functions, and special external functions through the SDK [18]. The added coded is handled by the NVIDIA compiler *nvcc*, which identifies code meant for the GPU and compiles this separately from the CPU code. Afterwards, these two binaries are combined into one executable file. The most important extensions are explained below. A description of all C for CUDA functionalities used in the thesis can be found in Appendix C.

Function Type Qualifiers

To specify to the compiler that one wants certain parts of the program to be run on the GPU, a specifier (*__global__* or *__device__*), is added to the function declaration, which makes the function callable from the CPU or the GPU respectively. When calling such functions, one needs to specify a set of parameters, the number of blocks to execute the computational kernels on and the number of threads in each block. For example, specifying 500 blocks and 128 threads each would give a total of 64000 threads to run the given code. C for CUDA is explained in more depth in The CUDA Programming Guide [10] and The CUDA Reference Manual [11].

Variable Type Qualifiers

Similar to function type qualifiers, C for CUDA also introduces qualifiers to specify variables that reside on the GPU. $\_\_device\_\_$ specifies that the variable resides on the device, $\_\_constant\_\_$ and $\_\_shared\_\_$ define if the variable is in constant or shared memory respectively. Constant memory is available to all threads of a kernel, while shared is only accessible by threads in the corresponding thread block.

Execution Configuration and Built-in Variables

As explained in Section 3.2.2, one must specify how a kernel is executed. The $<<< nBlocks, nThreads, sharedMemory, stream >>>$ expression enables the programmer to specify the number of blocks, the number of threads in each block, the amount of shared memory that is dynamically allocated per block, and the handle of the stream on which one wants to execute the kernel. The directive is placed between the name of the kernel that will be called and the parenthesized argument list.

The number of blocks ($nBlocks$) and the number of threads per block ($nThreads$) is specified as a dim3, that is a $uint3$ structure that holds 3 integers x,y, and z. Specifying for example $dim3(3, 2, 1)$ means having 6 blocks total (having y set to anything but 1 is not supported on current GPUs) and specifying for example $dim3(8, 8, 1)$ as the number of threads means having a total of $6 \times 8 \times 8 = 384$ threads, 64 per block.

Inside the kernel, one can uniquely identify the current thread by querying the built-in variables that take values from the execution configuration. These built-in variables match the specification variables in that they are $dim3$ as well. $gridDim$ contains the size of the grid, the same as the $nBlocks$ above, likewise $blockDim$ contains the size of the block the querying thread is in, the same as $nThreads$ above. $blockIdx$ gives the location of the block within the grid and $threadIdx$ gives the location of the thread within its block. Combining these one can get the location of the thread within the grid and map each of them to do work on a uniquely specified datapoint.

Asynchronicity

The CUDA architecture allows the programmer to run many different kernels concurrently, so as to utilize all the available cores. This is done by making asynchronous calls to kernels. The CPU part of the code will call the wanted

kernels, and continue separately from the GPU until either *syncthreads*() is called, or a memory transfer, which will block until the kernels are done, is initiated.

Example Code

Use of the above extensions may be seen in Listing 3.5. The program loads data into the GPU memory, increments all the values in the array by one, and copies it back to system memory. The kernel is preceded by $\_\_global\_\_$ to specify that is is a GPU function, and $blockIdx$, $blockDim$, $threadIdx$ are used to access unique data elements per kernel. The values for these variables are passed in the execution configuration of the kernel call.

3.2.4 CUFFT

When releasing the SDK, NVIDIA also released libraries to do Fourier transforms and linear algebra. The CUFFT [12] Fourier transform library, provides an interface that is modeled after the "Fastest Fourier Transform in the West", FFTW [12], and therefore is quite simple to utilize. CUFFT is generally slower than FFTW when computing an FFT for fewer than 8192 elements [21]. However, if one can utilize the batch option which is available for 1 dimensional CUFFT FFTs, the speedup compared to the FFTW is significant. The batch option enables scheduling more than one FFT simultaneously.

3.2.5 Occupancy Calculator

The occupancy calculator is a spreadsheet provided by NVIDIA [33]. After entering the number of threads per block, the number of registers used per thread and the amount of shared memory used per block into the spreadsheet, it will produce three graphs showing the occupancy of the SMs. The occupancy shows how many thread warps are available for each multiprocessor to schedule to hide memory latency. Higher occupancy will often mean faster execution of the whole kernel. The graphs show the current settings and expected gains and losses associated with changing the number of threads, registers, and shared memory usage. Adding the option "$--ptxas-options=-v$" to the nvcc compiler will print the number of threads, registers, and shared memory used.

```
1  __global__ void incrementArrayOnDevice(float *a, int N)
2  {
3    int idx = blockIdx.x*blockDim.x + threadIdx.x;
4    if (idx<N) a[idx] = a[idx]+1.f; //Equivalent of a sequential
         for-loop
5  }
6
7  int main(void)
8  {
9    float *a_h, *b_h;           // pointers to host memory
10   float *a_d;                 // pointer to device memory
11   int i, N = 10;
12   size_t size = N*sizeof(float);
13
14   // allocate arrays on host
15   a_h = (float *)malloc(size);
16   b_h = (float *)malloc(size);
17
18   // allocate array on device
19   cudaMalloc((void **) &a_d, size);
20
21   // copy data from host to device
22   cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
23
24   // do calculation on device:
25   // Part 1 of 2. Compute execution configuration
26   int blockSize = 4;
27   int nBlocks = N/blockSize + (N%blockSize == 0?0:1);
28
29   // Part 2 of 2. Call incrementArrayOnDevice kernel
30   incrementArrayOnDevice <<< nBlocks, blockSize >>> (a_d, N);
31
32   // Retrieve result from device and store in b_h
33   cudaMemcpy(b_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
34 }
```

Figure 3.5: Example CUDA kernel and kernel call, copy data to GPU, do computation, and copy the result back to host.

Chapter 4

Implementation

This chapter will present the algorithm and how it is meant to solve the problem of ultrasound aberration. It will also detail the implementation of the algorithm and the workings of the code of the implementation on the GPU.

4.1 The Algorithm

Several steps are required to estimate and correct the aberration in ultrasound signals. This thesis follows the process described by Måsøy [1], and it can in general terms be stated as follows:

- Select a subsample of data in the time-domain.
- Extract relevant frequencies from dataset.
- Compute correlation matrix.
- Compute coherence matrix.
- Calculate weight matrix.
- Iteratively apply weight matrix to previous correction filter to get new correction filter.

The first step is considered to be already performed in this thesis. However, the second step is relevant, because one has to decide on performing the algorithm in the frequency domain or in the time domain. Both are possible,

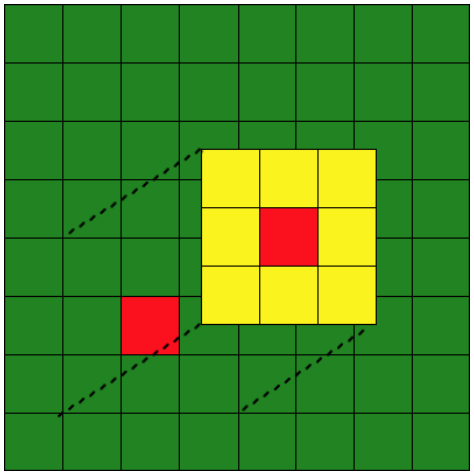


Figure 4.1: Convolution is a filter applied to all elements of an array, weighing all neighbors of the element and saving the sum for the center element.

but the computation of the algorithm is faster in the frequency domain. Because one uses a smaller data set, one selects a set of relevant frequencies and compute an average correction filter per position in the time domain rather than computing a separate correction filter per position. In this thesis, the computations are performed in the frequency domain.

The core steps of the algorithm are dependent on each other: computing the weight matrix uses the coherence matrix together with the correlation matrix. It may therefore be logical to combine these into one process that computes all the steps.

The correlation process as it is applied to the matrix is very similar to image convolution. Convolution is a sliding window that applies a filter to all neighbors of each element in the image and saves the sum of the computation. In this case there are only some more operations applied to the final sum, and the weights being applied are uniform, having the value of the center element. Convolution is illustrated in Figure 4.1.

Finally, applying the weights to the previous correction filter is a iterative process, whereby one calculates new weights and combines these with the previous correction filter. Deciding on the number of iterations to ensure convergence is often done by computing the error between the new value and the previous. Alternatively, one can compute a certain number of iterations that is certain to be enough to converge.

4.2 Process

The code has been implemented to be part of the Abersim 2.0 simulation software, and can be executed as a function call or as a choice when executing the command line version of Abersim. The data to be used in the command line version is currently read from a file, the name of which is specified on execution.

4.2.1 Loading data

The data is received as 64-bit double precision floating point values (double), and converted to 32-bit single precision floating point values (float). Precision is reduced to fit the data inside the smaller memory area. The reason for doing so is that the GPU is primarily used for working with single precision floats, and has only a few computational units available to compute on doubles [29]. The conversion (cast) to floats is therefore beneficial for later computations.

The data is read from MATLAB files using MATIO library [34] which enables reading MATLAB files from C code. The input data is normalized to the maximum value of the set as described in Equation 4.1, to avoid overflows when using floats.

$$u\_z = u\_z/max(u\_z(:)) \tag{4.1}$$

The matrix is copied to the GPU as doubles, and cast to floats, using the computational units set aside for double computations.

4.2.2 Discrete Fourier Transform

When estimating and correcting aberration in ultrasound images, the focus is on the amplitude and phase of the received signal. To attain this information, one must transform the data to a frequency spectrum. This is done by the Fourier transform. The Fourier transform is a mathematical transform from the time domain to the frequency domain. By taking the dataset of the signal received over a given timeframe, one can find the frequencies that are prevalent in the signal received.

To perform the discrete Fourier transform a fast Fourier transform [36][35] is used. The NVIDIA CUDA FFT, CUFFT [12] implementation already available as a library from NVIDIA is used. This library is similar to The

Fastest Fourier Transform in the West, FFTW [37] and has similar functions. The main difference is that CUFFT uses the GPU to perform the FFT, while FFTW uses the CPU.

The 3-dimensional matrix of floats that was copied to the GPU is passed to the CUFFT library, which performs Fourier transforms on each of the data arrays representing data from one transducer element in parallel. The resulting dataset is a 3-dimensional matrix of complex numbers.

4.2.3 Comparison

The next step in the process is to compare values to their neighbors to find the correlation between them and thereby deduce how much aberration there is. This results in a value for each element which describes how divergent the data from that transducer element is. The goal is to compute a new correction filter that can be applied to the next transmitted signal and remove aberrations estimated from the previous signal. Equation 4.2 is the calculations that finds the new correction filter, it is explained more fully by Måsøy [1] and Angelsen et al. [38].

$$S_{k,l} = \sum_m \sum_n \frac{|Y_{k,l} Y^*_{k-m,l-n}|^2}{\sqrt{|Y_{k,l}|^2}\sqrt{|Y_{k-m,l-n}|^2}} \quad Y_{k,l} Y^*_{k-m,l-n} \quad \frac{1}{S_{k-m,l-n}} \tag{4.2}$$

$Y_{k,l}$ represents the transducer element being considered, which is the center of the neighborhood. $Y^*_{k-m,l-n}$ is the complex conjugate of a neighboring transducer element.

The data from the FFT is passed to the comparison kernel, henceforth called the correlation kernel. This kernel incorporates all sections of the algorithm, since they are all computed for each transducer element.

The signal is in practice often different from a theoretical result. The signal contains a lot of noise and other fluctuations, giving a frequency spectrum where it can be hard to find the center frequency. Therefore, instead of extracting one value from the Fourier transform, an average of frequencies in a band of frequencies around the assumed center frequency is used. Figure 4.2 illustrates selecting the different frequencies for each transducer element. This removes noise, as well as making sure the relevant signal is used, as it may have been shifted out of focus. A range of values is loaded for each transducer element then. The previous correction matrix is also loaded into shared memory, as it is needed for the computations.

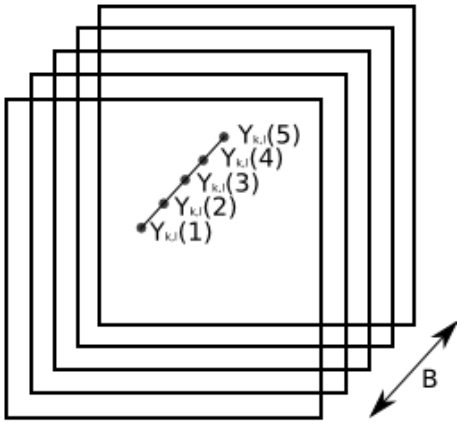


Figure 4.2: Illustration of selecting several values from the 3-dimensional matrix resulting from the Fourier transform. Computation is done on several frequencies around the center frequency of each transducer element $Y_{k,l}$. B is the bandwidth, the number of frequencies.

Each thread loads the data for each respective transducer element, and synchronize to ensure that all threads are done loading data. This gives all threads access to the neighboring data which they need. Subsequently, each thread performs a double loop for vertical and horizontal elements, where for each neighbor within the comparison area the comparison algorithm is performed.

Correlation

Correlation is a measure of similarity of two elements [43]. One wants to look at how similar an element is to its neighbors. Since the signal received by neighboring elements has traveled through almost the same tissue, it should be similar. Measuring to what degree the considered element is similar to its neighbors one can determine if its value is too high or too low. A value that is divergent compared to its neighbors is adjusted.

Each element is multiplied with the complex conjugate of the neighbor. The result is a value giving the similarity of the two elements. A zero value means no correlation while a high positive or negative value means high positive or negative correlation between elements. Positive correlation meaning that when one element varies, the other element varies similarly. Negative correlation means that when one element varies the other element varies in the opposite direction.

The process is repeated on each neighbor pair for each frequency, and the final sum is divided by the number of frequencies as expressed by Equation 4.3.

$$R(k,l,m,n) = \frac{1}{B}\sum^{b} Y_{k,l}Y^{*}_{k-n,l-m}(b) \tag{4.3}$$

Coherence

The coherence calculation is also called the normalized cross-correlation as seen in Equation 4.6. It is found by dividing the correlation by the magnitude of each of its parts. In this case, the parts are magnitudes of the current element and the neighboring element. Equations 4.4 and 4.5 are calculations of the parts. The returned value is normalized between 0 and 1, which ensures that the correlation will not add or subtract any energy from the whole equation.

$$R(k,l) = \frac{1}{B}\sum^{b} \sqrt{|Y_{k,l}(b)|^2} \tag{4.4}$$

$$R(k-m,l-n) = \frac{1}{B}\sum^{b} \sqrt{|Y^{*}_{k-m,l-n}(b)|^2} \tag{4.5}$$

$$W(k,l,m,n) = \frac{R(k,l,m,n)}{\sqrt{R(k,l)}\sqrt{R(k-m,l-n)}} \tag{4.6}$$

Weight

The new weight that is to applied to the correction filter based is calculated on the comparison of the neighboring elements. It is the magnitude of the coherence multiplied with the correlation as seen in Equation 4.7.

$$M(k,l,m,n) = |W(k,l,m,n)|R(k,l,m,n) \tag{4.7}$$

Correction

The weight is applied to the current correction values of the neighboring elements as seen in Equation 4.8.

$$S_{new}(k,l) = \sum_{m,n} M(k,l,m,n)\frac{1}{S_q(k,l,m,n)} \tag{4.8}$$

Reduction

After cross-correlation with the neighbors, the sum of all the correction values from the neighbors is used to calculate the new correction filter value for the element being considered. A convergence factor, μ, is used to speed up the convergence. Equation 4.9 shows the calculation performed.

$$S_{q+1}(k,l) = (1-\mu)S_q(k,l) + \mu S_{new}(k,l) \tag{4.9}$$

Iteration

The value obtained by this kernel is not a complete solution, and thus, one must iterate over the solution to get a converging value. Iteration stops when the change between the newly calculated correction filter and the one from the previous iteration is below a given threshold as shown by Equation 4.10.

$$|S_q|/|S_{q+1}| > \epsilon \tag{4.10}$$

The final S is the correction filter that can be applied to the next transmitted signal, to remove aberration.

4.2.4 Saving Data

After computation completes, the resulting correction matrix is written out to file. The matrix is saved as two separate matrices called *amplitude* and *phase*, containing the real data and the imaginary data respectively..

4.3 Code

This section describes the workings of the code in greater detail. It will follows program execution, and presents code needed to perform the different steps of the process described in the previous section. Excerpts of the program are shown for illustration. Complete code can be found in Appendix F. Appendix C gives more information about the different CUDA functions used.

4.3.1 Using Streams

CUDA enables asynchronous concurrent execution of kernels using streams [10], meaning that all operations executed on a stream will execute in order, but not necessarily in order in relation to other streams that may be executing. This functionality is used because it may improve speed when used on continuous data such as a real-time ultrasound scan, where data is received every time a new scan is completed.

The streams may then take advantage of the computation of the algorithm being in different stages; copy to device, cast to float, compute FFT, compute new correction filter, and copy back to host. Using streams, one may start another stream copying when the first stream has finished the copy to device stage and so on. It is possible to run several computational kernels simultaneously. This will give a higher throughput, by letting more computations be done simultaneously. Too many streams will, however, not result in a speedup, as streams will become queued waiting for each other to finish utilizing the GPU.

Since every stream needs its own separate memory, to take effect of the multiple streams opportunity, the allocated memory that is needed is much larger than with a single stream. One needs a full memory area for each stream. This limits the number of streams admissible, since there is limited memory on the GPU and the host. However, the number of streams needed to concurrently execute all the stages of the algorithm is not high enough that the maximum memory limit should be reached.

4.3.2 Initializing Memory

This step is completed only once, and its timings are therefore not included in any results. The memory is initialized on both the host and on the CUDA device. The following code describes part of the allocation process and the most important calls. The matrices are allocated as 1-dimensional arrays because CUDA does not support pointer-to-pointers style arrays. Listing 4.3 shows some of the allocations done, see Appendix F for complete code.

By using the $cudaMallocHost$ function, one allocates page-locked memory on host, enabling $cudaMemcpy2DAsync$. The allocation itself is slower, but copying data from this memory to device is much faster [10]. $cudaMallocPitch$ allocates aligned memory on the device, padded to allow multiples of 16-bit transfers to be used fully. This is used for 2D arrays as it enables cudaMemcpy2D. $sPitch$ gives the size of padded array in bytes. cudaMalloc allocates

```
1  fftwf_complex *fsrc; //float version of fftw complex numbers.
2  //Allocate page-locked memory for faster transfer
3  cudaMallocHost((void **) &fsrc, streams*totsize
4                 * sizeof(fftwf_complex));
5
6  cufftComplex *elementMatrix, *cufftResultMatrix;
7  //Data to CUFFT
8  cudaMalloc((void**) &elementMatrix,
9                 sizeof(cufftComplex)*streams*totsize);
10 //Result from CUFFT
11 cudaMalloc((void**) &cufftResultMatrix,
12                sizeof(cufftComplex)*streams*totsize);
13
14 cufftComplex *rM_h, *eM_d, *sM_d, *sOM_d, *rM_d;
15 size_t ePitch, sPitch, sOPitch, rPitch;
16 cudaMallocPitch((void **) &sM_d, &sPitch,
17                xsize * sizeof(cufftComplex), streams*ysize);
```

Figure 4.3: In the program initialization, all memory used is allocated, both on the GPU and the host.

unaligned linear memory on the device, used here for the 3D matrix returned from the FFT.

Almost all memory used is for cufftComplex datasets, with elements containing two floats x and y, one used for real values and one for imaginary values.

4.3.3 Loading Data

The code described in Listing 4.4 is the code executed to load the double-precision data matrix from the MATLAB file. The constants, *xsize*, *ysize*, *zsize*, *totsize*, and *streams* are, respectively, the width, height, time dimension, total size of one 3-dimensional matrix, and number of streams. *fnstr* is the filename of the MATLAB file where the matrix is to be found. The matrix is read from file, and transposed to convert from the column-major ordering of MATLAB to row-major ordering.

The doubles are copied to the device, and cast to floats. Both casting on the host and the GPU was tried, and using the GPU was found to be marginally faster.

As described in Listing 4.5, both *cudaSrc* and *elementMatrix* contain both real and imaginary values, thereby casting these to double and float re-

```
1  cudaMallocHost((void **) &src, totsize * streams
2                  * sizeof(fftw_complex));
3  double *u;
4  u = (double*)malloc( xsize * ysize * zsize * sizeof(double) );
5
6  long long int ret;
7  ret = read_genfile(xsize, ysize, zsize, fnstr, "u_z", u);
8  if ( ret ) {
9    write_log(logstr, "Matrix may not be loaded fully");
10 }
11
12 for (x = 0; x < n1; x++) {
13   for (y = 0; y < n2; y++) {
14     for (z = 0; z < n3; z++) {
15       src[z + y * n3 + x * n2 * n3][0] =
16                  u[z * n1 * n2 + y + x * n1];
17     }
18   }
19 }
```

Figure 4.4: Data is loaded from the file specified when calling the aberration estimation and correction function.

spectively enables reading each matrix continuously and not having to extract the real elements and the imaginary elements separately. This simplifies the casting kernel. Because the casting process is just reading data from memory and writing it back again, the maximal number of blocks are used, either the same amount as there are elements in the matrix, or the CUDA specified maximum of 512. The same applies to the number of threads. This spawns many threads, making the huge delay in accessing memory more transparent. The code of the casting kernel is given in Listing 4.6.

4.3.4 CUFFT

Using the CUDA FFT requires little configuration, one selects the size of the transform, N, and the type of transform. Since the data is in complex format already, complex-to-complex is used. Finally, the number of batches to compute simultaneously is selected. The functions called are shown below in Listing 4.7.

```
1  int blocks = iDivUp(totsize*2, 512*30) * 30;
2  if (blocks > 512) blocks = 512;
3  int threads = 512;
4  if (totsize*2 < 512) threads = totsize*2;
5
6  nBlocks = dim3(blocks);
7  blockSize = dim3(threads);
8
9  double* dcudaSrc = (double*) cudaSrc;
10 float* feM = (float*) elementMatrix;
11 cuda_castFromDouble<<<nBlocks, blockSize, 0, stream[s]>>>(
12           (double*)((char*)dcudaSrc + s*sizeof(double)*totsize),
13           (float*)((char*)feM + s*sizeof(float)*totsize),
                totsize*2);
```

Figure 4.5: The data is casted from double to float using a CUDA kernel. The arrays are cast before calling the calling the kernel to allow continuous access.

```
1  __global__ void cuda_castFromDouble(
2                      double* src, float* eM, int size) {
3
4    int idx = threadIdx.x + blockIdx.x * blockDim.x
5              + blockIdx.y * blockDim.x * gridDim.x;
6    for ( int i = idx; i < size;
7          i += blockDim.x * gridDim.x * gridDim.y) {
8      eM[i] = (float) src[i];
9    }
```

Figure 4.6: Each thread loads a part of the 3D matrix and casts elements of it to float, saving the result in a new array.

4.3.5 Comparison

The comparison kernel is executed on parts of the matrix. Before executing the kernel, the number of threads is selected. The maximum of threads that can be run per thread block depends on the amount of shared memory used. The maximal amount of memory available per thread block is 16384 bytes for the current GPUs. Using a bandwidth of for example 15 frequencies around the center frequency, the number of threads one could have would be 121. $121 \times 15 \times 8$ bytes = 14520 bytes, and including the previous correction filter, $14520 + 121 * 8$ bytes = 15488 bytes. The limit of shared memory that can be used by a block is 16384 bytes, the maximum available to a SM. Thus, the maximal number of threads possible using 15 frequencies will be

```
1  cufftPlan1d(&plan, N, CUFFT_C2C, BATCH);
2  cufftExecC2C(plan, eM, rM, CUFFT_FORWARD);
```

Figure 4.7: CUFFT is set up using cufftPlan*() and executed using cufftExec*()

$11 \times 11 = 121$. Dividing the dataset into areas of this maximum size and making space for the filter radius, one finds the number of blocks needed to cover the whole dataset. For example, with a 5×5 kernel, the radius would be 4, 2 on each side of the center, resulting in $7 \times 7 = 49$ blocks needed to cover a 100×100 dataset.

The Kernel

The kernel is the code executed by each of the threads to compute the correction matrix and also the window of neighbors that is placed over each element. The former will be called the computational kernel while the latter is just the kernel. The computational kernel is code in the form of a single program executed by a single thread, but with use of the built-in variables described in Section 3.2.3. This varies the values of variables of the kernel depending on the thread location in the grid and thus accesses different areas of the memory.

More threads are used to load the shared memory than perform the computation, this is because an apron of values is needed. Since the elements being computed on the edge of the kernel needs data from the neighboring values, these need to be loaded, but the correction computation for those values is done by a different computational kernel. Using more threads just to load data seems wasteful, but it is often faster than implementing more branching code for the remaining threads. To disable the redundant loading threads when computing the correction filter a conditional statement is used to select only the threads that should do computation. Listing 4.8 shows the statement disabling the threads.

In addition to speeding up loading, this method removes the need to make sure that the number of threads fit the matrix perfectly, as all threads that would have accessed memory outside the matrix are disabled as well. This is a simplification which wastes threads. However, it is often difficult to specify the number of threads exactly when setting up the execution configuration since the number of threads is a product of 3 values. The threads within the computation area of the kernel are enabled for computation, while the others are disabled.

```
1  bool active = false; //Inside matrix
2  bool write = false; //Inside computed result
3  if( globalx >=  0 && globaly >= 0
4      && globalx <= xsize-1 && globaly <= ysize-1) {
5    active = true;
6    if ((x >= KERNEL_RADIUS && x <= ((xs-1)-KERNEL_RADIUS)
7        && y >= KERNEL_RADIUS && y <= ((ys-1)-KERNEL_RADIUS))) {
8      write = true;
9    }
10 }
```

Figure 4.8: Threads working inside the kernel are activated, while threads only loading the apron are disabled for the computations.

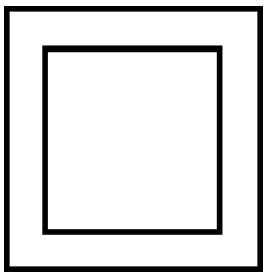


Figure 4.9: The kernel is the inner area, but computations needs data from the whole square which includes the outer area called the apron.

Data that is needed to do the computation but will not be updated itself is called the apron of the kernel, see Figure 4.9. When allocating memory on the device, it is not cleared, and may contain arbitrary values. The matrix on the GPU is padded with a number of elements equal to the width of the apron, to keep all accesses within the allocated memory area. The apron's values are set to zero if they extend outside the unpadded matrix, to prevent random data from the memory.

The shared memory is shared between all the threads of the thread block, and is declared outside the kernel. The size of the memory is dynamically allocated in the execution configuration, and is available as one large memory area. To subdivide the memory into portions used for different matrices, one must explicitly create pointers to the memory area, using offsets. Listing 4.10 is the declaration of the shared variable outside of the kernel, followd by the pointers pointing to locations in shared memory, offset by the portions used. The computations of the correlation, coherence, weight, and correction is done as a double for-loop of kernel width in both directions as showing in

Listing 4.11.

```
1 extern __shared__ cufftComplex shared[];
2
3 cufftComplex *data = shared;
4 cufftComplex *s = &(shared[xs*ys*(bandwidth+1)]);
```

Figure 4.10: Shared memory is allocated as one large memory block, and must be accessed using offsets.

```
1 //for loop vertical k
2   #pragma unroll 5
3   for (int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++) {
4
5     //for loop horizontal l
6     #pragma unroll 5
7     for (int l = -KERNEL_RADIUS; l <= KERNEL_RADIUS; l++) {
```

Figure 4.11: The for-loop covering the whole kernel. Only loop configuration is shown in the figure.

The following sections will follow the same form as Section 4.2.3 and explain the code used to achieve the goals described there.

Correlation

Computing the correlation one multiplies the considered element with the complex conjugate of the neighboring elements for all frequencies in the band as seen in Listing 4.12.

The data is read from shared memory using $data[smemPos+i*B]$, which refers to the considered element at the frequency i in the band. The variable $smemPos$ gives the base address of the considered element in respect to the thread executing. By adding B, which is the width and height of the kernel with apron, one accesses the next frequency, as they are loaded into shared memory as consecutive matrices for each frequency. The k variable is varied by the for-loop to change the vertical position accessed and the l in the horizontal direction. To help the compiler unroll the loops, the *#pragma unroll* is used to tell how many times the loop should be unrolled.

```
1  //R(k,l,m,n)
2  #pragma unroll 15
3  for (int i = 0; i < BANDWIDTH+1; i++) {
4    R_klmn.x += data[smemPos+i*B].x
5                * data[smemPos + k*xs + l + i*B].x
6                + data[smemPos+i*B].y
7                * data[smemPos + k*xs + l + i*B].y;
8    R_klmn.y += data[smemPos+i*B].y
9                * data[smemPos + k*xs + l + i*B].x
10               - data[smemPos+i*B].x
11               * data[smemPos + k*xs + l + i*B].y;
12 }
13 R_klmn.x /= (BANDWIDTH+1);
14 R_klmn.y /= (BANDWIDTH+1);
```

Figure 4.12: The correlation is computed for all frequencies in the frequency band around the center frequency.

Coherence

The computation of $R(k,l)$ is only dependent on the value of the element being considered and can thus be computed before the double for-loop. The computation of $R(k,l)$ and $R(k-m,l-n)$ is also performed over the whole frequency band. The imaginary part is cancelled by the computation of the magnitude. Listing 4.13 shows the computations.

```
1  R_kl += data[smemPos+i*B].x
2          * data[smemPos+i*B].x
3          + data[smemPos+i*B].y
4          * data[smemPos+i*B].y;
5
6  R_mn += data[smemPos + k*xs + l + i*B].x
7          * data[smemPos + k*xs + l + i*B].x
8          + data[smemPos + k*xs + l + i*B].y
9          * data[smemPos + k*xs + l + i*B].y;
```

Figure 4.13: Computations of the magnitude of the parts of the correlation.

The coherence is the normalization of the correlation. The smallest nonzero number representable as a float is added to prevent any divide-by-zero problems. The code for the coherence computation can be seen in Listing 4.14.

```
1 //W(k,l,m,n)
2 W_klmn.x = R_klmn.x / (R_kl * R_mn + 1e-5);
3 W_klmn.y = R_klmn.y / (R_kl * R_mn + 1e-5);
```

Figure 4.14: Computing the coherence is dividing the correlation with the magnitude of its parts.

Weight

The weight applied to the previous correction matrix is computed by applying the absolute value of the coherence to the correlation as seen in Listing 4.15.

```
1 //M(k,l,m,n)
2 M_klmn.x =  sqrtf(W_klmn.x*W_klmn.x
3             + W_klmn.y*W_klmn.y) * R_klmn.x;
4 M_klmn.y =  sqrtf(W_klmn.x*W_klmn.x
5             + W_klmn.y*W_klmn.y) * R_klmn.y;
```

Figure 4.15: To find the weight, the absolute value of the coherence is multiplied with the correlation.

Correction

To compute the updated correction matrix, one applies the weight to the neighborhood around the considered element in the previous iteration. The computation is a division of complex numbers and can be seen in Listing 4.16. The epsilon is again added to counter divide-by-zero problems.

Reduction

Finally, the next iteration of new correction matrix is computed using the previous correction matrix as the guess. The convergence factor is included to ensure convergence. The code can be seen in Listing 4.17.

Iteration

The computation is an iterative process, performed until the values for the correction filter converge. The previous version of the correction filter which will be used when checking if the next iteration has converged is first saved.

```
1  //new_s
2  sum.x += (M_klmn.x * s[smemPos + k*xs + l].x
3          + M_klmn.y * s[smemPos + k * xs + l].y)
4          / (s[smemPos + k*xs + l].x
5          * s[smemPos + k*xs + l].x
6          + s[smemPos + k*xs + l].y
7          * s[smemPos + k*xs + l].y + 1e-5);
8  sum.y += (M_klmn.y * s[smemPos + k*xs + l].x
9          - M_klmn.x * s[smemPos + k * xs + l].y)
10         / (s[smemPos + k*xs + l].x
11         * s[smemPos + k*xs + l].x
12         + s[smemPos + k*xs + l].y
13         * s[smemPos + k*xs + l].y + 1e-5);
```

Figure 4.16: The updated correction matrix is computed, it is a complex division of the weight by the previous correction values of the neighbors.

```
1  sum.x = (1.0f - mu) * s[smemPos].x + mu * sum.x;
2  sum.y = (1.0f - mu) * s[smemPos].y + mu * sum.y;
```

Figure 4.17: The iterative step, combining the previous correction matrix with the updated correction matrix to achieve a convergence.

Thereafter, the new correction filter is copied to the host. The difference between the two matrices is found and the average difference is computed as shown in Listing 4.18. The *minError* variable is compared to the epsilon set as the convergence term, and the iteration ends if it is below the threshold.

4.3.6 CPU Version

A CPU version of the code was also implemented. This computes the algorithm in the same way as the GPU version, but using only one thread. The CPU implementation is used to compare the performance of the parallel implementation to a nonparallel implementation. The code for the CPU version can be found in Appendix F.

```
1  //Calculate Error using CPU
2  float temp = 0;
3  for (j = 0; j < ysize; j++) {
4    for (k = 0; k < xsize; k++) {
5      temp += sqrt( sOLD_h[k+j*xsize].x * sOLD_h[k+j*xsize].x
6                  + sOLD_h[k+j*xsize].y*sOLD_h[k+j*xsize].y)
7                  / sqrt(s_h[k+j*xsize].x*s_h[k+j*xsize].x
8                  + s_h[k+j*xsize].y*s_h[k+j*xsize].y);
9    }
10 }
11 temp /= ysize*xsize;
12 temp = abs(temp - 1.0f);
13 if (temp < minError) {
14   minError = temp;
15 }
```

Figure 4.18: The difference between the new correction matrix and the previous one is computed to check for convergence.

Chapter 5

Evaluation

This chapter presents the test setup as well as the data used to benchmark the program. Section 5.1 describes the benchmark setup for both the CPU and the GPU. Section 5.2 lists the benchmark results. In Section 5.2.2, the performance of the CPU version of the algorithm and the GPU algorithm are compared and in Section 5.3 the overall result is discussed and evaluated.

5.1 Test Setup

This section describes the computer setup used for benchmarking, and how the test dataset was created.

5.1.1 Computer setup

The computer used for the benchmarking of the algorithms was a 2.83GHz Intel Core 2 Quad CPU with a NVIDIA Tesla C1060 graphics device. The operating system was Ubuntu 9.04 Linux using NVIDIA graphics drivers 185.19 and CUDA 2.1. Only one of the CPU cores was used for computation of the CPU portion of the program. The specifications of the GPU can be found in Table 5.1.

5.1.2 Dataset

The dataset for the benchmark is supplied by Kaupang[31]. The dataset is the result of a simulation of a signal fired from the transducer with a center

Table 5.1: The specifications of the NVIDIA Tesla C1060.

Global memory	4 GB
Number of streaming multiprocessors (SM)	30
Number of cores	240
Constant memory	65536 bytes
Shared memory per block	16384 bytes
Registers available per block	16384
Warp size	32
Maximum number of threads per block	512
Maximum sizes of each dimension of a block	512 x 512 x 64
Maximum sizes of each dimension of a grid	65535 x 65535 x 1
Clock rate	1.30 GHz
Concurrent copy and execution	Yes

frequency 3.5 MHz. The body wall used is 20 mm thick and designed to match abdominal wall characteristics. The measurements of the echo are then stored in a 3-dimensional matrix of $100 \times 100 \times 512$ double-precision floating-point values, representing the 512 element signal received from the 100×100 elements of the transducer. The bandwidth is 50 %, given a sampling frequency of 70.248 MHz and an in place Fourier transform, this equals 15 elements for each transducer element: 7 on each side of the assumed center frequency. The simulation setup is described in Figure 5.1.

5.2 Results

The program produces a correction filter, which is a matrix of complex numbers. Each element in the matrix represents the estimated weight that should be applied to the signal produced by the corresponding transducer element to remove aberration caused by the body wall. From each complex value, an amplitude and a phase can be computed and these values are applied to the next signal in accordance with Figure 2.12. The calculations in Equations 5.1, 5.2, and 5.3[42], compute the amplitude and the phase from the complex numbers.

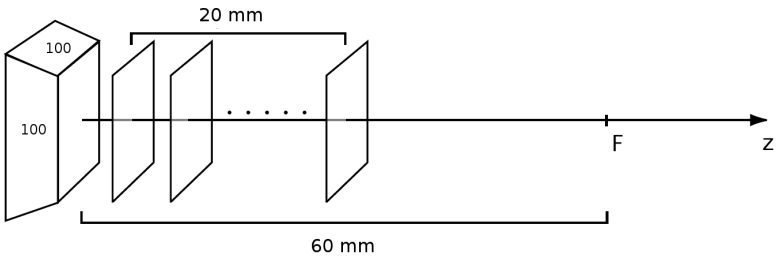


Figure 5.1: Simulation with 100 × 100 transducer elements, 20 mm body wall, 60 mm focal depth, 3.5 MHz center frequency producing 512 samples. Reproduced from [31], edited.

$$Z = a + bi \tag{5.1}$$

$$A = |Z| = \sqrt{a^2 + b^2} \tag{5.2}$$

$$\theta = \angle Z = \tan^{-1} \frac{b}{a} \tag{5.3}$$

The resulting amplitude and phase values may then be applied to the next transmitted signal to counter the aberration.

The resulting correction matrices can be visualized using MATLAB. Figures 5.2 and 5.3 show the amplitude weights and the time delay that should be applied to the next transmitted signal respectively. The scale in the amplitude image is logarithmic, and the scale in the phase image is linear.

5.2.1 Iteration

The weights are iteratively computed, and applied to the previous correction filter. Figures 5.4 and 5.5 show the iteration for both the amplitude and phase values of the correction filter. 21 iterations are needed in the test case for the filter to converge to a precision of 0.01. For a threshold of 0.05 and 0.1, the number of iterations needed are 8 and 3 respectively.

5.2.2 Speedup

The wall clock time used by the GPU and CPU was measured using a timer function utilizing the SSE functionality of the CPU. To allow for variations caused by programs running in the background or other elements, the

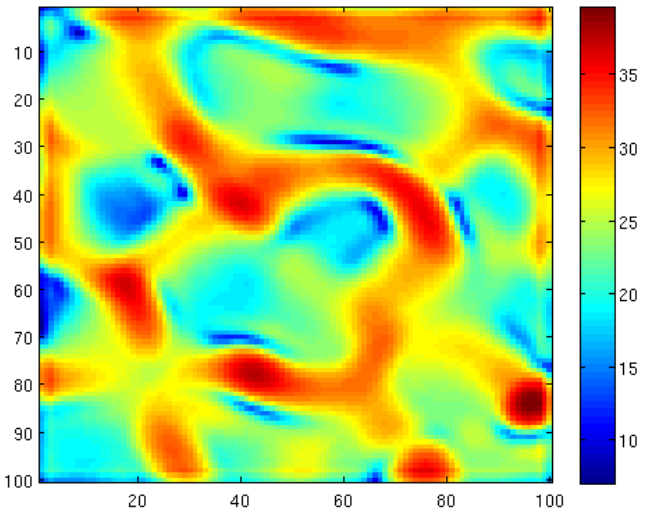


Figure 5.2: The amplitude of the correction filter to be applied to next transmitted signal, computed using GPU.

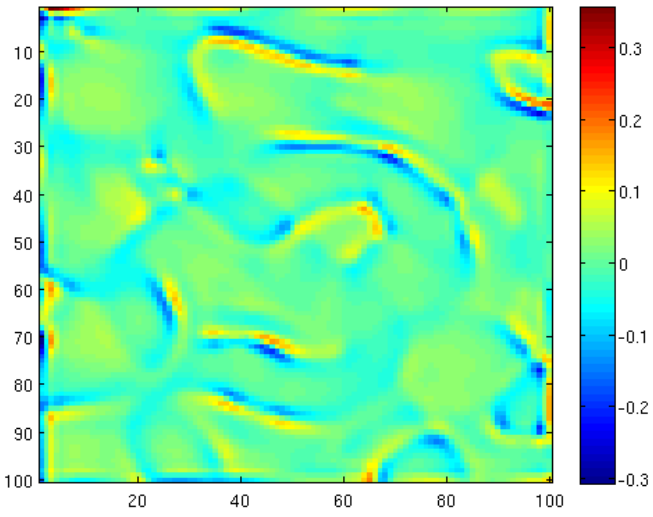


Figure 5.3: The phase of the correction filter to be applied to next transmitted signal, computed using GPU.

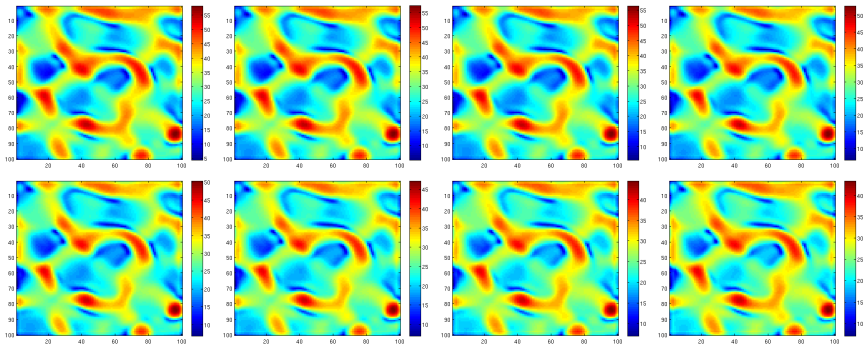


Figure 5.4: The amplitude of the correction filter after 1, 2, 4, 7, 10, 14, 18, and 21 iterations.

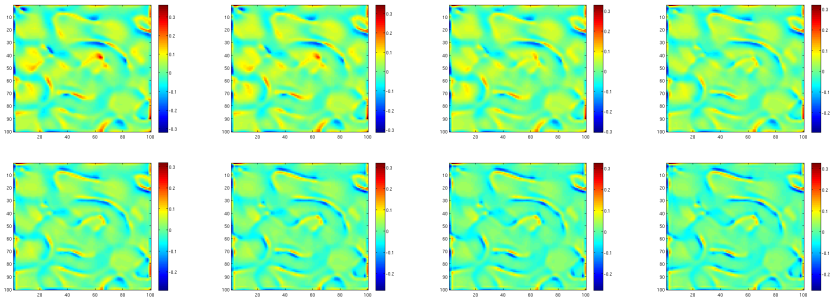


Figure 5.5: The phase of the correction filter after 1, 2, 4, 7, 10, 14, 18, and 21 iterations.

Table 5.2: The timing results for 21 iterations of the algorithm on the GPU.

Copy data to device	13.541 ms
Cast using CUDA	1.66 ms
FFT computation	3.468 ms
Correlation computation	133.6 ms
Copy result to host	0.002 ms
Total, without allocations	152.299 ms

Table 5.3: The timing results for 21 iterations of the algorithm on the CPU.

Copy input data to FFTW structured array	23.188 ms
FFT computation	67.826 ms
Correlation computation	716.192 ms
Total without allocations	807.075 ms

computations were repeated 101 times. Also to remove any inconsistencies caused by memory allocations, caching, CPU auto clocking function, or similar events, the first measurement was discarded.

Figures 5.2 and 5.2 shows the output of the program when printing all timings. Each section of the algorithm is timed. There is no timing of the subparts of the correlation computation, as the kernel does not return any feedback while executing.

By comparing the time taken to complete the same task on the CPU versus the time taken to complete the same task on the GPU, one can calculate the speedup. Figure 5.6 shows average execution time using a NVIDIA Tesla C1060 GPU and an Intel Core 2 Quad CPU running at 2.83 GHz. The average time taken for the GPU is 152.49 milliseconds while the CPU does the task in 807.23 milliseconds, giving a total speedup of 807.23 / 152.49 = 5.29. Figure 5.6 is a visual representation of the time difference.

The time to copy data from host to device is much larger than that of copying from device to host. This is because the whole $100 \times 100 \times 512$ 3D matrix of doubles is copied to the GPU, cast to float, and Fourier transformed, while only a 100×100 matrix representing the correction filter is copied back. Preliminary steps are difficult to compare, as they are not

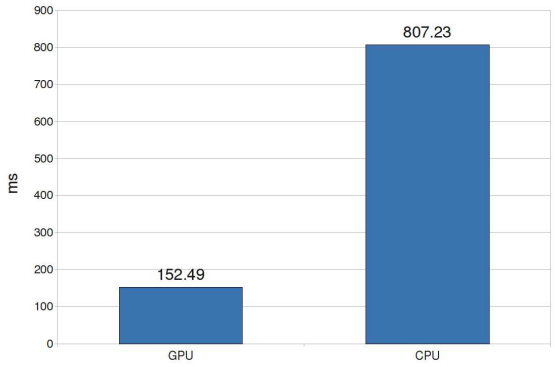


Figure 5.6: Average time for GPU and CPU to compute one iteration of the algorithm, less is better.

equal between the two architectures.

The Fourier transform takes much less time on the GPU. This can be accredited to the batch option used to compute several FFTs simultaneously, in comparison the CPU version computes all FFTs sequentially.

Finally, the most time is used on the computation of the correlation, taking 133 milliseconds on the GPU and 716 milliseconds on the CPU. This measurement includes all the steps of the algorithm after the FFT, correlation, coherence, weight, correction filter, and iteration.

5.3 Discussion

This section considers the results and looks their significance. It focuses on how successful the parallelization of the algorithm has been as well as the correctness of the result produced, execution time and comparing the use of GPU against CPU. It looks at the possibility of computing this algorithm in real time, that is, producing a "live" image with the algorithm. Finally, how the results achieved for this field relates to other fields and if the GPU is a good hardware to build problem solving software on.

5.3.1 Parallelization

The algorithm used in the program is well suited for parallelization. Since computations can be done on individual elements of the matrix, one utilizes many computational units to perform the computations simultaneously. For

the Fourier transform, using the batch functionality of CUFFT enables the completion of the FFT less time than using the sequential version of the CPU.

The correlation is also parallelizable, it can be compared to performing a convolution over the dataset using a limited kernel. This is especially efficient on the GPU as it is a graphical task [41]. It is parallelizable since only a small set of values around the element being considered is needed, and the calculations for each point can be computed separately. However, it is not embarrassingly parallelizable, as computation relies on a larger set of data when considering the bandwidth around the center frequency. The technique of separable convolution was attempted, but had to be rejected because the computations for each frequency in the bandwidth could not be separated. If one could find a way to separate the computation for each frequency, one would achieve a better parallelization, and probably a more efficient program.

The implementation should not have a scaling problem, as the dataset currently uses about 40 MB and the Tesla C1060 has a global memory of 4 GB. For this one needs 49 thread blocks to correlate, while the maximal number of thread blocks is 65535^2.

There is, however, little room for extending the bandwidth, which affects the number of threads one can have per block. Currently, a 15 frequency band limits the number of threads to 121 per block. The minimum number of threads one can have per block is 25, 5×5, which is the size of the kernel. Using the minimum number of threads, one is able to have a bandwidth of 79, that is, 39 elements on each side of and including the center frequency. This will slow execution down enormously as each thread needs to execute several for loops of length 79 in the correlation kernel, giving diverging execution paths. Limited bandwidth may become a problem with long samples of data, since this would lead to wider FFT results and thus more elements to cover the same frequency range.

5.3.2 Correctness

The result of computing the algorithm is compared to doing the computations using MATLAB. Two methods were used in MATLAB and Figure 5.8 shows the results of the different methods. The main difference is in the algorithm used.

The MATLAB results are based on Kaupang[31], which does not use the

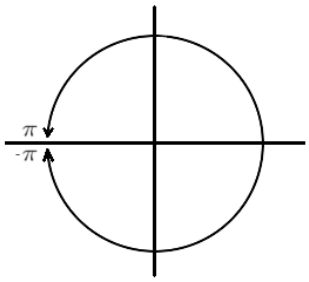


Figure 5.7: Unit circle describing how $-\pi$ and π are opposite, but still very close values. Wrapping effect is the small change leading to great difference in value, but not in phase.

same algorithm to find the amplitude and phase of the correction filter. Kaupang also smooths the input value to avoid noise and extreme values. This produces a larger range of values, and different shapes in the resulting visualizations. This is especially true for the phase, which seems to be more similar to the amplitude results than the phase results produced by the program. This data was, however, used to compare if results from the program seem reasonable at an early stage with regards to variations in values, and extreme values. Also, because a different estimation and correction algorithm are used to compute this data, it was not expected to match the program output.

The second set of images shows the result of a MATLAB script performing the algorithm used in this thesis. It was developed to have an exact value to compare program output with. The script was developed by Ph.D. student Thor Andreas Tangen at The Department of Circulation and Medical Imaging at The Norwegian University of Science and Technology. The script can be found in Appendix E.

The third set of images shows the program output. As can be seen in the figure, the output the script produces is a scaled version. The results of the program has the correct form, but differences may be attributed to numerical instability or the fact that the result is computed using floats while MATLAB uses doubles. Another reason that the results are different may be wrapping effects[46], values that exceed π will "flip" around to the negative side. There is, as can be seen from Figure 5.7, little difference between $-\pi$ and π, but it looks quite different as values in the result image.

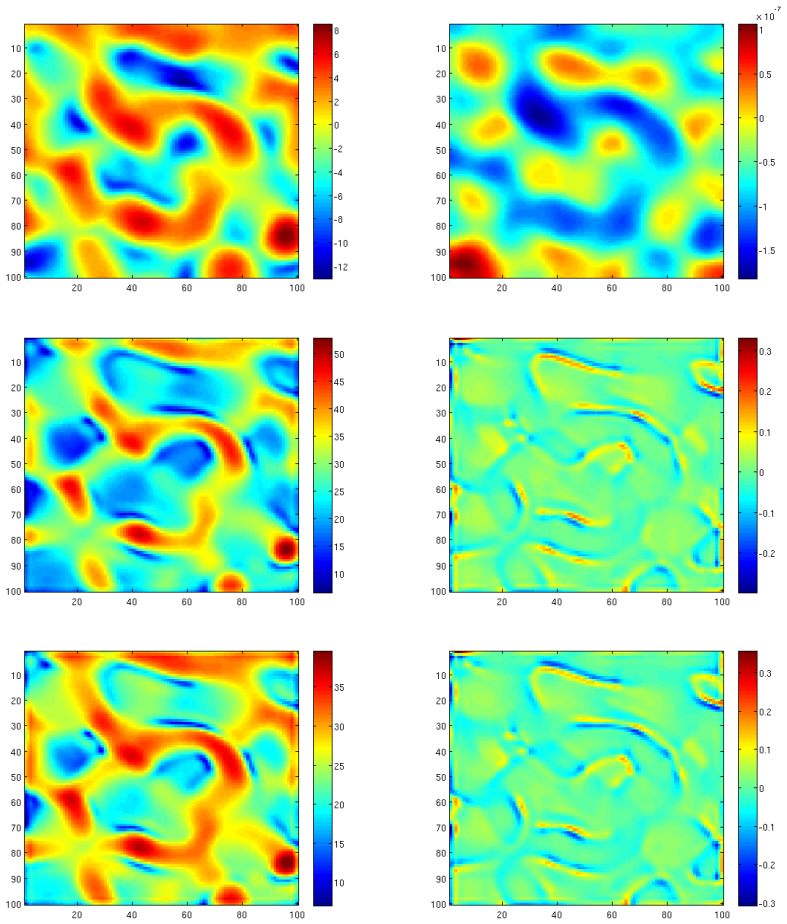


Figure 5.8: Amplitude and Phase of aberration correction filter, top images are results from [31], the middle images are from the MATLAB script in Appendix E, and bottom images are the program output.

5.3.3 Timings

As is clearly evident from Section 5.2.2, using the GPU instead of the CPU is beneficial for this algorithm, with over 5 times greater speed. Performing the FFT on the GPU is by itself 20 times faster than on the CPU.

The computations of the correction filter for each beam uses information from the previous beam the subsequent beams should need fewer than 21 iterations for the calculation to converge. This should result in further significant speedups, as the difference between computing 21 and 1 iterations is $133 - 39 = 94$ milliseconds.

In current ultrasound imaging equipment, there is a standard of 256 transducer elements and often only a subgroup of these are used to transmit a beam. The number of elements in this thesis is 10000, computing the algorithm in an actual environment will probably use significantly less time. In addition, on the smaller dataset a 5×5 kernel would probably be too large, using a 3×3 kernel would give smaller for loops inside the computational kernel, resulting in fewer branches and faster computation.

Bottlenecks

There are some bottlenecks in the program that work against the GPU, and contribute substantially to the execution time. The two most evident bottlenecks are shared memory constraints, and single threaded execution.

The major bottleneck in this program is memory access from the correlation kernel. This kernel needs to hold a large amount of shared memory, because each of its threads accesses an area of 25 neighbors on 15 planes. To accommodate this, almost the whole shared memory available per block of threads is used. The amount of shared memory needed per thread block limits the number of thread blocks that can be executed on a SM at one time, and thereby the number of warps that can be scheduled, limiting the use of the GPU. It may also cause memory collisions where threads try to access the same shared memory bank and have to be serialized, causing further delay.

It is difficult to measure the impact of the larger memory set on performance, since there also more computations with the larger set. The correlation kernel takes 39.73 milliseconds running the application with a bandwidth of 0, meaning only the center frequency. This is about 29.74 % of the computation using a bandwidth of 14 elements, plus the center frequency. The number of computations is fewer, but it illustrated that the extra memory usage has large impact on total performance.

A solution may be to align memory reads, or restructure the shared memory to avoid bank conflicts. Also, a solution may be to perform less of the computations for every thread, letting the threads reuse other thread's computations instead of doing them over again. This would limit the accesses to shared memory. Solutions are further elaborated in Section 6.2.

As explained in Section 3.1.3, execution time suffers in cases where there are not enough threads to hide the memory access latency. This is caused by at least two of the bottlenecks in the program, few or single threaded execution and the cast.

The correlation suffers from the computations for each element being computed by a single thread. So even though the matrix is divided into multiple threads, each compute a long set of sequential instructions. With 49 thread blocks and 30 SMs, the number of warps available for each SM to use for memory latency hiding is few. This leads to SMs stalling while waiting for memory accesses and a low efficiency in computing the result. The performance impact of this is unclear, since one would have to divide the computations into more threads to calculate the difference. With such a low SM occupancy and long sequential instructions in the kernel, the speedup from parallelization in relation to the sequential CPU version is limited. Splitting up this kernel into smaller kernels would help this. However, this would use more global memory accesses as kernels are dependent on each other.

Equally, the cast process suffers from the same problem. All the data in the input matrix $100 \times 100 \times 512 = 5120000$ is both copied to the device as double precision floating point values, and then cast to single precision floating point values. When casting, the only operation carried out by each thread is to read a value from the double matrix and write it to the float matrix as a float. There are no computational instructions to run while waiting for loads to return, the only other tasks that can be scheduled to run are other memory access instructions. This increases the execution time, evident from the program output where it takes about 1.09 % of the total GPU execution time. In a real-time system this may not be relevant as data from the transducer is received as 16-bit integers, but could be cast to floats in hardware.

The correlation kernel contains a large amount of for loops and branches which degrades performance severely, as explained in Section 3.1.3. Removing the loops by using more threads and more memory might speed up the process even though one would have more memory accesses.

5.3.4 Real-time

To achieve a "live" image possibility with all the data being filtered by the algorithm, the following criteria must be met:

24 images per second Time available to compute image is 1000 / 24 = $41.67ms$.

Up to 250 beams per image Available time per beam computation is 41.67 / 250 = $0.167ms$

Algorithm applied to each beam Needs to be applied in less than 0.167ms

As can be seen from the calculations, the limit for applying the algorithm to every beam in a real-time system is less than 0.2ms. Looking at the results of the current computation which is completed in approximately 152 milliseconds, this is far off. It is unrealistic to implement this algorithm in real-time in its current state.

Optimization

There are several possibilities that may be considered to come closer to the "live" image goal. First, optimization of the code may greatly increase the speed at which it is executed. Examples, of codes that can be optimized are, for example, checking when to stop the iteration, using the GPU to do this work instead of copying data to the host, or running the whole iteration on the GPU as a kernel without intermediate steps on the host.

Another optimization is to do fewer iterations. Currently, the loop will stop when the difference between the current iteration and the last is less than 0.01. If this limit is higher, the iteration would finish earlier, at the expense of accuracy. If one selected fewer elements to measure the difference, this would also decrease total time taken. There is a 4 millisecond difference between computing the filter with and without error computation.

Finally, the same correction filter may be used for more than one beam, or frame. Since the signal is transmitted through very similar tissue for each beam, using the same correction filter could achieve almost equal quality of the image.

By utilizing the Occupancy Calculator spreadsheet, one can find possible changes that can optimize the kernel execution. Small changes can have a great effect, for example by decreasing the bandwidth, from 14 to 10; one enables a higher number of threads per block. As can be seen from the first

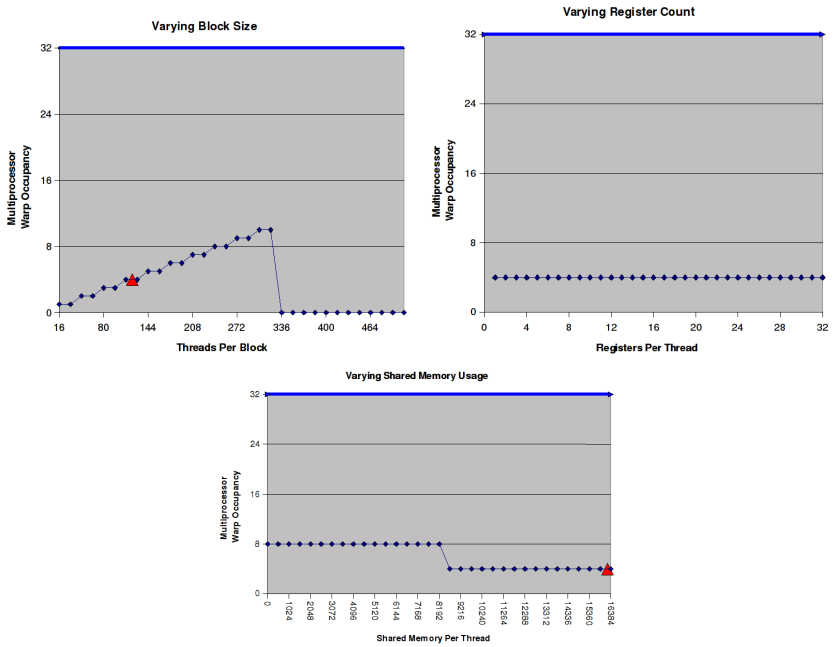


Figure 5.9: The graphs show how many thread warps are available per multiprocessor to schedule to hide memory access latency.

graph in Figure 5.9, changing from 121 threads per block to 169 threads gives a higher occupancy. This speeds up the total time from about 152 milliseconds to 113 milliseconds. Bringing it to around 310 threads per block would be optimal. As can also be seen from the Figure, given the current settings, there is little else one can do to increase the occupancy without large changes.

5.3.5 Related work

Implementing the aberration estimation and correction algorithm on the GPU achieves a speedup. This adds the problem to the group of problems that when implemented on the GPU give a speedup. Ryoo [22] evaluates GPU implementations of applications and the speedup gained. All applications presented by Ryoo [22] have to a smaller or greater degree gained a speedup from GPU implementation.

The characteristics of this problem is similar to the Lattice-Boltzman Method application in [22], which gains a 12x speedup, but has two distinct bottlenecks. These are, synchronization issues at every time step when all

threads have to be globally synchronized, and constraints on the number of threads that can be executed because of shared memory capacity. Both these problems are evident in the aberration estimation and correction problem, in addition to the smaller speedup because of branching. Similarly, Bryson [41] achieves large speedups in convolution computation, which is a process similar to the correlation process of the aberration correction problem.

These problems, including aberration correction, shows a speedup that can be interpreted as a trend of how GPU increases program speeds, even with little optimization. As explained in Section 3.1.2, exposing parallelism in problems and using multicore processing units is an attempt so circumvent *the brick wall*. Looking at the results of among others, Ryoo [22], Bryson [41], and Stone [45], the GPU seems to be a viable options for circumventing *the brick wall* speedwise. The availability of a programming language extension with a similar syntax and behavior to current programming languages equally enhances the prospect of using the GPU's multicore architecture for general purpose problems.

Chapter 6

Conclusion and Future Work

This chapter summarizes the findings of this thesis, and look at how well goal was reached. It also looks at directions the work can be extended.

6.1 Conclusion

Ultrasound is a non-invasive imaging procedure that delivers good images of the human body. However, the human body wall introduces aberration in the ultrasound signal, resulting in distortions and blurring of the image. This is especially problematic in obese patients, where the fat in form of globules in the body wall generally causes the most aberration. Correcting this aberration would improve image quality and help physicians make accurate diagnoses.

In this thesis, Måsøy's algorithm for correcting the aberration by estimating the time-delay and amplitude fluctuations caused by the body wall was considered [1]. The estimate was used to create a correction filter that was applied to the next ultrasound signal, to cancel the effects of the aberration. This thesis parallelized this algorithm, and investigated how this parallel algorithm could be implemented on off-the-shelf GPUs based on NVIDIA's CUDA architecture.

Our results show that the algorithm is highly parallelizable. The transducer is made up of many elements; computation of the correction filter can be computed almost separately for each transducer element making it very suitable to compute in parallel. The limit of the parallelization is that one cannot easily decompose the problem further, and thus, each thread must work sequentially for its transducer element. There are usually 256 or more

transducer elements in a transducer. The GPU is a highly parallel SIMD device, specialized at performing the same computation on several data elements simultaneously. In fact the NVIDIA Tesla C1060 has 240 cores. This architecture maps very well to the computation of the correction filter, and achieves a substantial speedup of over 5x compared to a single CPU. The problem joins a growing set of problems found to benefit from implementation on the GPU.

Altough it was shown that using the GPU was a huge improvement over single CPU computation, but there is still some work needed to reach speeds high enough to implement in ultrasound equipment. However, with some configuration variations and additional hardware to speed up memory transactions, it should be possible to achieve this.

6.2 Future Work

There are certainly areas that would be interesting to develop further. Some ideas to where the result of this thesis could be taken are listed below.

It would also be interesting to use the procedure together with a live stream of ultrasound signals. The test case was only performed on a single signal, so attempting to utilize the program with several new datasets would make it possible to test the streams capability. This could also be combined with variations in how many and which frames and beams of the ultrasound imaging that should be used as datasets for the algorithm. Limiting the number of datasets leaves more time for the computation to complete before a new set arrives.

Implementing more of the iteration on the GPU would also be interesting, utilizing a reduction kernel over each correction filter matrix to find the difference from last iteration, instead of copying the data to the host and using the CPU to compare the iterations. This may improve computation time.

It would be interesting to look at how one could limit the amount of shared memory used, or at least limit the number of accesses to shared memory. This would improve performance by removing bank conflicts. One could possibly restructure the shared data and the kernel code so that reads to the same memory is not read at the same time by many threads. To do this, one would have to look at how the memory is accessed.

The data generated by the transducer is 16-bit integers. If the algorithm

could be implemented for integers instead of floats, one could skip the casting and achieve a speedup. Computing with integers smaller than 24-bits is equally fast as using floats on the GPU. However, there may be a problem representing the values and computing correct results using integers.

Comparing the GPU version to a multicore CPU version of the algorithm to see how large the improvement using the GPU is would also be interesting.

Since one only needs a subset of the frequencies provided by a Fourier transform, it might be enough to use other frequency extraction techniques. One such technique is the spectral estimator described in Angelsen [38]. This might be much faster than computing the whole transform. As the Fourier transform is a major part of the computations this could speed it up significantly.

Utilizing the texture memory to hold the data from the FFT might speed up accesses. Texture memory is data placed in global memory, but cached in the SMs. Each SM has a 16kb cache. Texture memory has a latency of 100+ cycles as opposed to 200-300 cycles of the global memory. Texture memory capitalizes on 2D locality so accessing neighboring memory locations with 2D locality should give a performance increase. Texture memory is read-only, but since the result of FFT is only read it could be placed in texture memory.

Bibliography

[1] *Estimation and correction of aberration in medical ultrasound imaging*, Svein-Erik Måsøy, Doctoral Thesis, Department of Engineering Cybernetics, Norwegian University of Science and Technology, October 2004.

[2] *Correction of ultrasonic wave aberration with a time delay and amplitude filter*, Svein-Erik Måsøy, Tonni F. Johansen, and Bjørn Angelsen, The Journal of the Acoustical Society of America 113 (4) Pt. 1 April 2003.

[3] *Estimation of ultrasound wave aberration with signals from random scatterers*, Svein-Erik Måsøy, Bjørn Angelsen, and Trond Varslot, The Journal of the Acoustical Society of America 115 (6) June 2004.

[4] *Iteration of transmit-beam aberration correction in medical ultrasound imaging*, Svein-Erik Måsøy, Trond Varslot, and Bjørn Angelsen, The Journal of the Acoustical Society of America 117 (1) January 2005.

[5] *GPU Sound Processing*, Åsmund Herikstad, Specialization Project, Department of Computer and Information Science, Norwegian University of Science and Technology. December 2008.

[6] *Wikipedia on GPGPU*, Available online `http://en.wikipedia.org/wiki/GPGPU`, Wikipedia, accessed February, 2009.

[7] *GPU Gems 2*, Matt Pharr, Randima Fernando, Addison-Wesley Professional, March 13, 2005, Available online `http://developer.nvidia.com/object/gpu˙gems˙2˙home.html`.

[8] *CUDA, Supercomputing for the Masses*, Available online `http://www.ddj.com/cpp/207200659`, Rob Farber, Dr. Dobb's Journal, accessed February, 2009.

[9] *Parallel Programming*, Barry Wilkinson, Michael Allen, Prentice Hall; 2 edition, March 14, 2004.

[10] *CUDA Programming Guide*, Available online http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf, NVIDIA Corporation 2008, accessed August, 2008.

[11] *CUDA Reference Manual*, Available online http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/CudaReferenceManual_2.1.pdf, NVIDIA Corporation 2008, accessed August, 2008.

[12] *CUDA CUFFT 2.1 Library Documentation*, Available online http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/CUFFT_Library_2.1.pdf, NVIDIA Corporation 2008, accessed October, 2008.

[13] *Image Convolution with CUDA*, Available online http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/convolutionSeparable/doc/convolutionSeparable.pdf, Victor Podlozhnyuk, NVIDIA Corporation 2007, accessed June, 2007.

[14] *Wikipedia on CUDA*, Available online http://en.wikipedia.org/wiki/CUDA, Wikipedia, accessed April, 2009.

[15] *What is CUDA*, Available online http://www.nvidia.com/object/cuda_what_is.html, NVIDIA corporation, accessed June, 2009.

[16] *Wikipedia on Medical Ultrasonography*, Available online http://en.wikipedia.org/wiki/Medical_ultrasonography, Wikipedia, accessed February, 2009.

[17] *Essentials of Medical Ultrasound: A Practical Introduction to the Principles, Techniques and Biomedical Applications (Medical methods)*, Michael H. Repacholi and Deirdre A. Benwell, Humana Pr, October 1982.

[18] *Parallel Computing in CUDA*, Available online http://www.gpgpu.org/asplos2008/ASPLOS08-3-CUDA-model-and-language.pdf, Michael Garland, NVIDIA Research, Tutorial at ASPLOS 2008, NVIDIA Corporation 2008.

[19] *Effects of Abdominal Wall Morphology on Ultrasonic Pulse Distortion*, Laura M. Hinkelman, T. Douglas Mast, Michael J. Orr, and Robert C. Waag, Ultrasonics Symposium, 1997. Proceedings., 1997 IEEE, October 1997.

[20] *Evaluating Parallel Algorithms: Theoretical and Practical Aspects*, Lasse Natvig, Dr. Ing. thesis, Department of Computer Systems and Telematics, The Norwegian Institude of Technology, 1991.

[21] *CUFFT vs FFTW comparison*, Available online `http://www.science.uwaterloo.ca/~hmerz/CUDA_benchFFT/`, Hugh Merz, University of Waterloo, accessed April, 2009

[22] *Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA*, Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, Wen-mei W. Hwu, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming 2008.

[23] *Wikipedia on SIMD*, Available online `http://en.wikipedia.org/wiki/SIMD`, Wikipedia, accessed April, 2009.

[24] *How GPUs Work*, David Luebke, Greg Humphreys, Computer, vol. 40, no. 2, pp. 96-100, February 2007.

[25] *GPU Computing*, John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips, Proceedings of the IEEE Vol 96. No. 5 May 2008.

[26] *Scientific Visualization Studio*, Available online `http://svs.gsfc.nasa.gov/documents/arch_6.html`, NASA, accessed June, 2009.

[27] *The Landscape of Parallel Computing Research: A View From Berkeley*, Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, Katherine A. Yelick, Technical Report No. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18, 2006.

[28] *Computer Architecture: A Quantitative Approach*, J. Hennessy and D. Patterson, 4th edition, Morgan Kaufman, San Francisco, 2007.

[29] *NVIDIA Tesla Architecture Specifications*, Available online `http://www.nvidia.com/object/tesla_supercomputer_tech_specs.html`, NVIDIA Corporation, accessed June, 2009.

[30] *Utilizing GPUs for Real-Time Visualization of Snow*, Robin Eidissen, Master Thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, February 2009.

[31] *Second-Harmonic Aberration Correction*, H. Kaupang, T. Varslot, and S.-E. Måsøy, Proc. 2007 IEEE International Ultrasonics Symposium, 2007.

[32] *Real-Time Wavelet Filtering on the GPU*, Erik Axel Rønnevig Nielsen, Master Thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, May 2007.

[33] *CUDA GPU Occupancy Calculator*, Available online `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`, NVIDIA Corporation, accessed June, 2009.

[34] *MAT File I/O Library*, Available online `http://sourceforge.net/projects/matio`, Christopher Hulbert, accessed June, 2009.

[35] *Wikipedia on Fast Fourier Transform*, Available online `http://en.wikipedia.org/wiki/Fast_Fourier_transform`, Wikipedia, accessed June, 2009.

[36] *Digital Image Processing*, R.C. Gonzalez and R.E Woods, 2nd Edition, Prentice Hall, 2002.

[37] *Fastest Fourier Transform in the West*, Available online `http://www.fftw.org/`, Matteo Frigo, Massachusetts Institute of Technology, accessed June, 2009.

[38] *SUPREME ultrasound imaging system*, Bjørn A.J. Angelsen, Tonni F. Johansen, Svein–Erik Måsøy, S Peter Nãsholm, Øyvind K.–V.Standal, Trond Varslot, internal document at Department of Circulation and Imaging, Norwegian University of Science and Technology, April 2005.

[39] *Measurements of ultrasonic pulse arrival time and energy level variations produced by propagation through abdominal wall*, Laura M. Hinkelman and Dong-Lai Liu, Leon A. Metlay, Robert C. Waag, J. Acoust. Soc. Am. Volume 95, Issue 1, pp. 530-541, January 1994.

[40] *Adaptive filtering in the frequency domain*, Dentino, M.; McCool, J.; Widrow, B., Proceedings of the IEEE , vol.66, no.12, pp. 1658-1659, Dec. 1978.

[41] *Accelerated 2D Image Processing on GPUs*, Bryson R. Payne, Saeid O.Belkasim, G. Scott Owen, Michael C.Weeks, Ying Zhu, Computational Science – ICCS 2005, Springer Berlin / Heidelberg, 2005.

[42] *Advanced Engineering Mathematics, 8th Edition*, Erwin Kreyszig, John Wiley & Sons inc., 1999.

[43] *Understanding Correlation*, Available online `http://www.hawaii.edu/powerkills/UC.HTM`, R.J. Rummel, Honolulu: Department of Political Science, University of Hawaii, 1976.

[44] *Rise of the Graphics Processor*, David Blythe, Proceedings of the IEEE Vol 96. No. 5 May 2008.

[45] *How GPUs Can Improve the Quality of Magnetic Resonance Imaging*, Sam Stone, Haoran Yi, Wen-mei Hwu, Justin Haldar, Bradley Sutton, Zhi-Pei Liang, Unpublished article, Available online `http://www.gigascale.org/pubs/1175.html`, The First Workshop on General Purpose Processing on Graphics Processing Units (GPGPU) October 2007. Boston MA.

[46] *Correction of phase wrapping in magnetic resonance imaging*, Leon Axel and Daniel Morton, Med. Phys. 16, 284 (1989).

Appendix A

Annotated Bibliography

This appendix presents papers that are especially relevant for this thesis and that may be useful before reading the thesis or as a deeper study.

A.1 Aberration in Medical Ultrasound

The basis for this thesis can be found in the doctoral thesis *Estimation and correction of aberration in medical ultrasound imaging* by Svein-Erik Måsøy [1], it presents the aberration phenomenon and also gives a complete presentation of the process of finding the algorithm used in this thesis. There are also three articles based on this thesis, they are: *Correction of ultrasonic wave aberration with a time delay and amplitude filter* [2], *Estimation of ultrasound wave aberration with signals from random scatterers* [3], and *Iteration of transmit-beam aberration correction in medical ultrasound imaging* [4] and correspond respectively to chapters 2, 3, and 4 of the doctoral thesis. These articles are an excellent introduction to the doctoral thesis as they clearly present the work and results, but do not go to the same depth as the thesis.

A.2 C for CUDA

To get a general introduction to General-Purpose Computation on GPUs the Wikipedia article on GPGPU [6] and the book GPU Gems 2 [7] presents basic concepts.

To understand more of writing programs using C for CUDA (Compute

Unified Device Architecture) the web page *CUDA, Supercomputing for the Masses* by Rob Farber [8] is a very good tutorial for understanding and writing CUDA code. Also, the *CUDA Programming Guide* [10] and *CUDA Reference Manual* [11] are invaluable when writing C for CUDA code.

Since this thesis focuses on parallelizing the aberration correcting algorithm and programs for the GPU is most effective when they are highly parallelized, studying the book *Parallel Programming 2nd edition* [9] or similar material should give an understanding of the process taken.

Appendix B

NOTUR poster

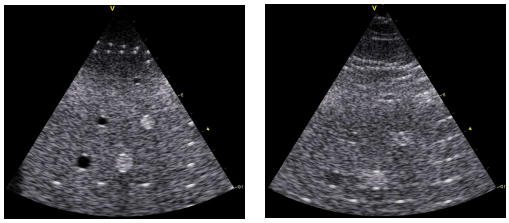
The poster on the following page was presented at the NOTUR conference in Trondheim on May 19. 2009.

Parallel Techniques for Estimation and Correction of Aberration on Medical Ultrasound Images

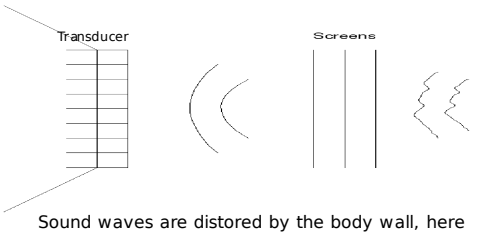
Åsmund Herikstad Master Student and Dr. Anne C. Elster, Advisor
Department of Computer and Information Science, NTNU

In collaboration with: Dr. Svein-Erik Måsøy and Thor Andreas Tangen
Department of Circulation and Medical Imaging, NTNU

Medical ultrasound is a great tool because of its noninvasiveness. However, the sound waves have to travel through different tissues, the resulting image can become aberrated.



Left: Unaberrated image, right: Aberrated Image



Sound waves are distored by the body wall, here represented by screens which can be estimated.

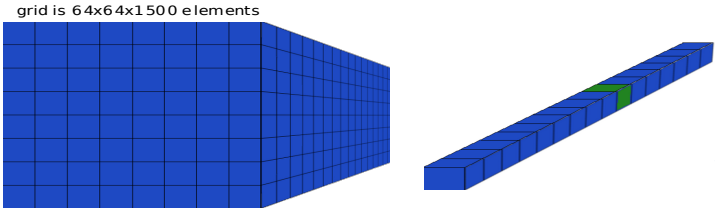
$$b_A(t) = \sum_{k=0}^{n} A_k * y(t-\tau) = A * y(t-\tau)$$

The aberrated is s sum of several screens, if we can find A, we can aquire the unaberrated signal.

The process of estimating and correcting ultrasound aberration:

1. Perform FFT to get relevant frequency
2. Compute correlation matrix
3. Compute coherence matrix
4. Calculate weight matrix
5. Iteratively apply weight matrix to previous correction filter to get new correction filter

FFT can be done using CUFFT, CUDA Fast Fourier transform, transformes are computed in parallel. This is up to 10x faster than serial implementation.

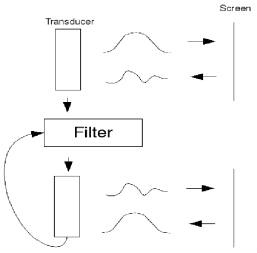


A signal is received per transducer element and Fourier transformed. The center freqency of each transform is selected.

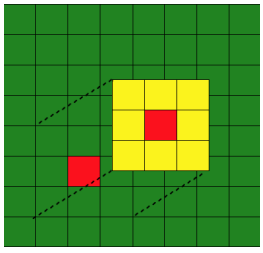
Step 2-5 can be computed iteratively as a convolution using the GPU to apply the filter to all elements of the matrix.

$$S_{i,j} = \sum_k \sum_l \frac{|Y_{i,j} Y^*_{i-k,j-l}|^2}{\sqrt{|Y_{i,j}|^2}\sqrt{|Y_{i-k,j-l}|^2}} \; Y_{i,j} Y^*_{i-k,j-l} \; \frac{1}{S_{i-k,j-l}}$$

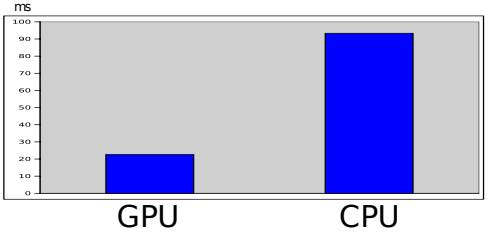
Each element is compared to its neighbours and then weighted based on how much it deviates.



Convolution process is repeated until adequate convergence is obtained.



Above algorithm is applied to all elements of the matrix in parallel.



Preliminary Results
4-5x speedup

Acknowledgement: We would like to thank NVIDIA for providing several of the graphics cards used in this project through Dr. Elster's membership in their Professor Affiliates Program.
Contact: asmundhe@stud.ntnu.no

HPC Research Group

NTNU
Norwegian University of Science and Technology

Appendix C

CUDA Function Description

This chapter describes the CUDA functions and variables used in the program created in this thesis. For a complete list of all CUDA functions and variables see [10], [11], and [12].

cudaMalloc() allocates linear memory on the device.

cudaMalloHost() allocates page-locked memory on the host. This enables a much higher bandwidth for for example cudaMemcpy than pageable memory since it is directly accessible by the device.

cudaMallocPitch() allocates linear memory on the device, but in addition may pad the allocation to ensure that the memory is aligned so as to enable coalesced accesses.

cudaFree() frees allocated memory on the device.

cudaFreeHost() frees page-locked memory on the host allocated with cudaMallocHost().

cufftPlan1d() creates a 1D FFT plan configuration of given size, may add *batch* to specify how many 1D transforms to configure.

cufftExecC2C() executes a CUFFT complex-to-complex transform plan.

cufftComplex is a $float2$ struct containing 2 floats x and y. Used to hold complex numbers for use with CUFFT.

__global__ declares a function that is executed on the device, but callable only from the host.

__device__ declares a variable that resides on the device or declares a function that is executed on the device and only callable from the device.

__shared__ declares a variable that resides in shared memory of a thread block. Is only accessible from the threads within the block.

cudaThreadSynchronize() blocks to makes the host wait for all threads to finish computation on the device.

cudaMemcpy() copies data between device and host.

cudaMemcpy2D() copies 2-dimensional matrices between device and host.

cudaMemcpy2DAsync() does the same as cudaMemcpy2D, but is asynchronous, that is the call returns before the copying is complete. Only works on page-locked host memory.

cudaMemcpyDeviceToDevice is an enum specifying direction of the copy to cudaMemcpy$*$-functions, copy from one memory are to another on the device.

cudaMemcpyHostToDevice is an enum specifying direction of the copy to cudaMemcpy$*$-functions, copy from host to device.

cudaMemcpyDeviceToHost is an enum specifying direction of the copy to cudaMemcpy$*$-functions, copy from device to host.

dim3 is a *uint*3 struct containing 3 unsigned integers x, y, and z. Used to hold number of blocks or number of threads per block for execution configuration of kernels.

blockIdx is a *uint*3 struct used inside a kernel to hold the block index in a grid.

blockDim is a *uint*3 struct used inside a kernel to hold dimensions of the block.

gridDim is a *uint*3 struct used inside a kernel to hold the dimensions of the grid.

threadIdx is a *uint*3 struct used inside a kernel to hold the thread index within the block.

#pragma unroll is a directive that tells the compiler how many times to unroll a loop. Without it compiler only unrolls small loops.

Appendix D

Program Installation and Execution

This chapter will list the steps needed to be taken to "install" the program as well as how to use its functionality. Instructions are intended for Linux.

D.1 Installation

The following steps should be done to "install" the aberration correction program:

- Set up the CUDA environment: Install NVIDIA drivers, the CUDA toolkit, and the CUDA SDK from `http://www.nvidia.com/object/cuda_get.html`.
- Download MATIO, MAT File I/O Library, from `http://sourceforge.net/projects/matio`.
  - Unpack MATIO.
  - Run $./configure$ with the option $--enable-shared = yes$ within the matio folder.
  - Add the option $-fPIC$ to line 108 of the Makefile in the matio folder.
  - run *make*.
  - run *sudo make install*.

- Download and install FFTW3 from `http://www.fftw.org/download.html`.
- Install libcblas.
- Install libatlas.
- Install libm.
- Install libz.
- Install libg2c.
  - make a symlink *sudo ln -l libg2c.so.0 libg2c.so* in */usr/lib*.
- Download the stand alone C version of Abersim 2.0 from `http://www.ntnu.no/abersim/download`.
- Unpack the Abersim 2.0 files.
- Unpack the source files of the correction program into the abersim2 folder.
- Edit the *Makefile.in* file to reflect the position of the NVIDIA_CUDA_SDK folder.
- In the abersim2 folder run *make all*.

D.2 Execution

There are two ways to utilize this program. One is the command line option when calling Abersim 2.0. The other, is calling the functions from within the Abersim 2.0 application.

Command Line

Here is the directions of use printed by the Abersim 2.0 program with the program added:

```
———————————— Abersim 2.0 Usage ————————————

For aberration simulation include filename containing MATLAB
data.
```

```
  ./abersim2 mfile

For aberration correction include filename containing MATLAB
data, as well as the center frequency index in the FFT, the
bandwidth around the FC in the FFT, the X-, Y- and Z-dimensions.
Optionally include number of streams to use, the number of
iterations for the estimation loop and the device to use.
  ./abersim2 mfile 25 14 100 100 512 3 1 1
```

Basically, one would have to supply the information about the dataset as well as the MATLAB file containing the data.

Function call

To use the function from another function within the Abersim2.0 application one would need to include the $abersim_correction.h$ file and call the functions with the information needed in the same way as the command line version. Before calling the *abcorrection*() function, one needs to select the device on which to do the computations. If this is not done, the default, device 0, is used. the *indata* and *result* matrices must already be allocated using the *allocate*() and the *loadData*() function is used to load the input data from file if this is desirable. Finally, when the computation is complete the data may be saved to file using *saveData*() function and the matrices deallocated using *deAllocate*().

D.3 Troubleshooting

If one get the error: "./abersim2:

error while loading shared libraries: libabersim2.so: cannot open shared object file: No such file or directory", the fix seems to be to add the local directory to the library path. That is, run $export\ LD\_LIBRARY\_PATH = .:\$LD\_LIBRARY\_PATH$.

Appendix E

MATLAB Script for Computing Correction Filter

This chapter contains the MATLAB script to compute the aberration correction filter. It assumes the file "testDataDouble2" contains the input set u_z. The script will generate an avi file with each of the steps computed. By default the script computes 30 iterations, that is it does not check for convergence.

```
%%
clear all;
load testDataDouble2;
%%
close all;
Nfft = size(u_z,3);
u_z = u_z/max(u_z(:));
U_z = fft(u_z,[],3);
% indx for frequencies
fc_indx = 27;
f_win = 7;
f_ix = fc_indx + (-f_win:f_win);

% kernel size
ks = 5;
% half kernel size
hks = (ks-1)/2;

% Pad with zeros
U_z2 = zeros(size(U_z,1)+2*hks,size(U_z,2)+2*hks,size(U_z,3));
U_z2((1+hks):(end-hks),(1+hks):(end-hks),:) = U_z;
U_z = U_z2;
```

```
23
24 N = size(U_z,1);
25 ix = (1+hks):(N-hks);
26
27 % Make movie
28 make_movie = false;
29 if make_movie
30     aviObj = avifile('aberation.avi','fps',10,'Quality',100,'
           Colormap',jet(256),'compression','None');
31 end
32
33 % Computing correlation matrices
34 R = zeros(size(U_z,1),size(U_z,2));
35 for ff = -f_win:f_win
36     R = R + U_z(:,:,fc_indx+ff).*conj(U_z(:,:,fc_indx+ff));
37 end
38 R = R/length(f_ix);
39
40 M_klmn = zeros(length(ix),length(ix),ks*ks);
41 cnt = 1;
42 for n=-hks:hks
43     for m=-hks:hks
44         R_klmn = 0;
45         for ff = -f_win:f_win
46             R_klmn = R_klmn + U_z(ix,ix,fc_indx+ff).*conj(U_z(ix
                   +n,ix+m,fc_indx+ff));
47         end
48
49         R_klmn = R_klmn/length(f_ix);
50         R_kl = R(ix,ix);
51         R_mn = R(ix+n,ix+m);
52         w = R_klmn./(sqrt(R_kl)*sqrt(R_mn) + 1e-5);
53
54         M_klmn(:,:,cnt) = abs(w).*R_klmn;
55         cnt = cnt + 1;
56     end
57 end
58
59 % Doing the iterations
60 S = ones(size(R));
61 S_new = zeros(size(S));
62 mju = 0.3;
63
64 for iter = 1:30
65     S_new(:) = 0;
66     cnt = 1;
67     for n=-hks:hks
68         for m=-hks:hks
69             S_new(ix,ix) = S_new(ix,ix) + M_klmn(:,:,cnt)./(S(ix
```

```
                +n,ix+m) + 1e-5);
70              cnt = cnt + 1;
71          end
72      end
73
74      S = S + mju*(S_new - S);
75
76      amp = abs(S(ix,ix));
77      phase = angle(S(ix,ix));
78
79      figure(1);
80      subplot(121)
81      imagesc(20*log10(amp)),colorbar;
82      title(sprintf('Amplitude, iteration %d',iter))
83      subplot(122)
84      imagesc(phase),colorbar;
85      title(sprintf('Phase, iteration %d',iter))
86
87      if make_movie
88          F = getframe(gcf);
89          aviObj = addframe(aviObj,F);
90      else
91          pause(0.1);
92      end
93  end
94
95  if make_movie
96      aviObj = close(aviObj);
97  end
```

Appendix F

Program Code

This chapter lists the code of the program. The .h files are not included and neither are files from Abersim2.0.

F.1 Main Program Code

The main functions on the host. These functions are called through Abersim2.0 and will subsequently call GPU code.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <time.h>
5  #include <math.h>
6
7  extern "C" {
8  #include "../general/abersim_general.h"
9  #include "../general/propcontrol.h"
10 #include "../math/abersim_math.h"
11 #include "../io/abersim_io.h"
12 #include "abersim_correction.h"
13 }
14
15 #include "convolutionS.h"
16 #include "gpuOperations.h"
17 #include "cpuCorrelation.h"
18 #include "cpuFFT.h"
19 #include "gpuFFT.h"
20 #include "gpuCast.h"
21
```

```
22
23 ////////////////////////////////////////////////////////
24 // CONFIGURATION
25 ////////////////////////////////////////////////////////
26 // Number of times to loop convolution kernels
27 const int N = 50;
28 // Convergence factor
29 const float MU = 0.1f;
30 // Error limit, epsilon
31 const float EPS = 0.01f;
32 // Number of loops to run to collect test data
33 const float T = 1; //101
34
35 // Code that gives very accurate timer.
36 static unsigned long long rdtsctime()
37 {
38     unsigned int eax, edx;
39     unsigned long long val;
40     __asm__ __volatile__("rdtsc":"=a"(eax), "=d"(edx));
41     val = edx;
42     val = val << 32;
43     val += eax;
44     return val;
45 }
46
47 extern "C" {
48 void abcorrection( fftw_complex *src,
49           cufftComplex *rM_h,
50           const int fc,
51           const int bandwidth,
52           const int xsize,
53           const int ysize,
54           const int zsize,
55           const int streams,
56           const int n) {
57
58   ////////////////////////////////////////////////////////
59   // VARIABLES
60   ////////////////////////////////////////////////////////
61   int i, j, k, t=0;
62   //Error holder
63   cudaError err;
64   //Timer variables
65   unsigned long long  timestart, timefinish, totstart,
66                       totfinish, loopstart, loopfinish;
67   //Number of blocks to execute for kernel
68   //and number of threads in each block.
69   dim3 nBlocks, blockSize;
70   //Stores number of bytes used for memcpy
```

```
71  size_t bytes;
72  int totsize;
73  totsize = xsize*ysize*zsize;
74
75
76
77  ////////////////////////////////////////////////////////
78  //ALLOCATIONS (These are only done once per program run)
79  ////////////////////////////////////////////////////////
80  timestart = rdtsctime();
81
82  fftwf_complex *fsrc; //float version of fftw complex numbers.
83  fftw_complex *cudaSrc;
84  cudaMallocHost( (void **) &fsrc,
85                  streams*totsize * sizeof(fftwf_complex));
86  cudaMalloc(     (void **) &cudaSrc,
87                  streams*totsize * sizeof(fftw_complex));
88
89  //allocation of matrices used for FFTW.
90  //elementMatrix and resultMatrix
91  fftwf_complex ***cpueM, ***cpurM;
92  cpueM = (fftwf_complex***) malloc(
93          sizeof(fftwf_complex**)*xsize);
94  for (i = 0; i < xsize; i++) {
95    cpueM[i] =  (fftwf_complex**) malloc(
96                sizeof(fftwf_complex*)*ysize);
97    for (j = 0; j < ysize; j++) {
98      cpueM[i][j] = (fftwf_complex*) malloc(
99                    sizeof(fftwf_complex)*zsize);
100   }
101 }
102 cpurM = (fftwf_complex***) malloc(
103                 sizeof(fftwf_complex**)*xsize);
104 for (i = 0; i < xsize; i++) {
105   cpurM[i] =  (fftwf_complex**) malloc(
106                 sizeof(fftwf_complex*)*ysize);
107   for (j = 0; j < ysize; j++) {
108     cpurM[i][j] = (fftwf_complex*) malloc(
109                   sizeof(fftwf_complex)*zsize);
110   }
111 }
112 fillRandom(cpueM, xsize,ysize,zsize);
113 fillRandom(cpurM, xsize,ysize,zsize);
114 setupFFT( zsize, xsize, ysize, zsize,
115           cpueM[0][0], cpurM[0][0]);
116
117 //Allocation of matrices used for CPU kernel.
118 cufftComplex **elementMatrix_h, **correlationMatrix_h;
119 createMatrix(&elementMatrix_h,xsize,ysize);
```

```
120   createMatrix(&correlationMatrix_h,xsize,ysize);
121
122   //GPU matrices
123   cufftComplex  *elementMatrix,
124                 *cufftResultMatrix, *tempMatrix_h;
125   //Data to CUFFT
126   cudaMalloc( (void**) &elementMatrix,
127               sizeof(cufftComplex)*streams*totsize);
128   //Result from CUFFT
129   cudaMalloc( (void**) &cufftResultMatrix,
130               sizeof(cufftComplex)*streams*totsize);
131   cuda_setupFFT(zsize, xsize*ysize);
132
133   cufftComplex *s_h, *sOLD_h;
134   cudaMallocHost((void**) &s_h, sizeof(cufftComplex)
135               *streams*xsize*ysize);
136   cudaMallocHost((void**) &sOLD_h, sizeof(cufftComplex)
137               *streams*xsize*ysize);
138
139   cufftComplex *eM_d, *sM_d, *sOM_d, *rM_d;
140
141   //sPitch gives size of padded array in bytes.
142   //sPitch/8 gives number of elements in
143   //padded array (of cufftComplex of doubles).
144   size_t ePitch;
145   size_t sPitch;
146   size_t sOPitch;
147   size_t rPitch;
148
149   //cudaMallocHost allocates page-locked memory on
150   //host (enables cudaMemcpy2DAsync and other functionality).
151   //Allocate strided memory that is aligned, speeds
152   //up memory transfers and access.
153   //Data for correlation
154   cudaMallocPitch(  (void **) &eM_d, &ePitch, xsize
155               * sizeof(cufftComplex), streams*ysize);
156   //Correctionfactors that change ever iteration of correlation
157   cudaMallocPitch(  (void **) &sM_d, &sPitch, xsize
158                * sizeof(cufftComplex), streams*ysize);
159   //Original correctionfactors
160   cudaMallocPitch(  (void **) &sOM_d, &sOPitch, xsize
161               * sizeof(cufftComplex), streams*ysize);
162   //Result from correlation
163   cudaMallocPitch(  (void **) &rM_d, &rPitch, xsize
164               * sizeof(cufftComplex), streams*ysize);
165
166   //Create streams to interleave processing
167   cudaStream_t stream[streams];
168   for (i = 0; i < streams; i++) {
```

```
169     cudaStreamCreate(&stream[i]);
170   }
171
172   timefinish = rdtsctime();
173   printf( "\nALLOCATE: Time to allocate (in seconds) (on hpc6):
174           %f\n", ((double)(timefinish-timestart)/2830000000));
175
176
177   ////////////////////////////////////////////////////////
178   // LOOP FOR EACH STREAM
179   ////////////////////////////////////////////////////////
180   loopstart = rdtsctime();
181   for (t = 0; t < T; t++) {
182
183   //resetting variables
184   i = j = k = 0;
185   int blocks, threads;
186   fillRandom(elementMatrix_h, xsize, ysize);
187   nBlocks = dim3(streams*ysize,xsize);
188   cuda_setStartValues<<<nBlocks,1>>>(sM_d, sPitch);
189   //Stream to run current loop on.
190   int s = t % streams;
191
192   totstart = rdtsctime();
193   ////////////////////////////////////////////////////////
194   // COPY DATA TO DEVICE
195   ////////////////////////////////////////////////////////
196   printf("\n");
197   timestart = rdtsctime();
198   //Assuming non-page-locked memory from connected
199   //code so we cannot use Async mode
200   bytes = sizeof(fftw_complex)*totsize;
201   cudaMemcpy( (char*)cudaSrc +
202               s*totsize*sizeof(fftw_complex),
203               (char*)src,
204               bytes,
205               cudaMemcpyHostToDevice);
206   //Need to synchronize to get correct time count.
207   cudaThreadSynchronize();
208   timefinish = rdtsctime();
209   printf( "COPY: Time to copy data to device (in seconds)
210         (on hpc6): %f\n", ((double)
211         (timefinish-timestart)/2830000000));
212   err = cudaGetLastError();
213   if (err != cudaSuccess) printf("ERROR: %s
214         :ERROR\n", cudaGetErrorString( err ) );
215
216
217
```

```
218 ////////////////////////////////////////////////////////
219 // CASTING USING CUDA
220 ////////////////////////////////////////////////////////
221 //Casting from a complex double to a complex float.
222 timestart = rdtsctime();
223 printf("Casting using CUDA...\n");
224 blocks = iDivUp(totsize*2, 512*30) * 30;
225 if (blocks > 512) blocks = 512;
226 threads = 512;
227 if (totsize*2 < 512) threads = totsize*2;
228
229 nBlocks = dim3(blocks);
230 blockSize = dim3(threads);
231
232 double* dcudaSrc = (double*) cudaSrc;
233 float* feM = (float*) elementMatrix;
234 cuda_castFromDouble<<<nBlocks, blockSize, 0, stream[s]>>>(
235         (double*)((char*)dcudaSrc + s*sizeof(double)*totsize),
236         (float*)((char*)feM + s*sizeof(float)*totsize),
237         totsize*2);
238 //Need to synchronize to get correct time count.
239 cudaThreadSynchronize();
240 timefinish = rdtsctime();
241 printf( "CAST: Time to cast using CUDA (in seconds)
242         (on hpc6): %f\n", ((double)
243         (timefinish-timestart)/2830000000));
244 err = cudaGetLastError();
245 if (err != cudaSuccess) printf("ERROR: %s
246         :ERROR\n", cudaGetErrorString( err ) );
247
248
249
250 ////////////////////////////////////////////////////////
251 // Compute FFT using CUFFT
252 ////////////////////////////////////////////////////////
253 printf("\n");
254 timestart = rdtsctime();
255 printf("Computing FFT using CUFFT...\n");
256 cuda_doFFT( (cufftComplex*)((char*)elementMatrix
257             + s*sizeof(cufftComplex)*totsize),
258             (cufftComplex*)((char*)cufftResultMatrix
259             + s*sizeof(cufftComplex)*totsize));
260 //Need to synchronize to get correct time count.
261 cudaThreadSynchronize();
262 timefinish = rdtsctime();
263 printf("CUFFT: Time for CUFFT to compute (in seconds)
264             (on hpc6): %f\n", ((double)
265             (timefinish-timestart)/2830000000));
266 err = cudaGetLastError();
```

```
267   if (err != cudaSuccess) printf("ERROR: %s
268               :ERROR\n", cudaGetErrorString( err ) );
269
270
271
272   ////////////////////////////////////////////////////////
273   //GPU CONVOLUTION
274   ////////////////////////////////////////////////////////
275   printf("\n");
276   timestart = rdtsctime();
277
278   float minError;
279   minError = 100000000000.0f;
280
281   cudaThreadSynchronize();
282   cudaMemcpy2DAsync(  s_h, xsize*sizeof(cufftComplex),
283              sM_d, sPitch,
284              xsize*sizeof(cufftComplex), ysize,
285              cudaMemcpyDeviceToHost, stream[s]);
286   cudaThreadSynchronize();
287
288   threads = floor(sqrt(16384 / ((bandwidth+1) * 8 + 8)));
289   if (threads-KERNEL_W < 1) {
290     printf("FATAL ERROR: Bandwidth to wide.\n");
291     exit(1);
292   }
293   else if (threads*threads*50 > 16199) {
294     //Uses maximum registers available per SM.
295     threads = 17;
296   }
297   if (  (threads * threads * (bandwidth + 1)
298         + threads * threads) * 8 > 16383) {
299     //Uses maximum shared memory available per block.
300     threads = 15;
301   }
302
303   //7 is maximum of threads 11 - KERNEL_RADIUS*2
304   nBlocks = dim3( iDivUp(xsize+KERNEL_W,threads-KERNEL_W),
305                   iDivUp(ysize+KERNEL_W,threads-KERNEL_W));
306   blockSize = dim3(threads,threads); // (11,11)
307
308   printf("Computing new correction matrix...\n");
309   i = 0;
310   do {
311     convolutionS<<<nBlocks, blockSize, (threads*threads*
312              (bandwidth+1)+threads*threads)*8, stream[s]>>>(
313              (cufftComplex*)((char*)cufftResultMatrix
314              + s*sizeof(cufftComplex)*totsize),
315              (cufftComplex*)((char*)rM_d + s*rPitch*ysize),
```

```
316             (cufftComplex*)((char*)sM_d + s*sPitch*ysize),
317             MU,
318             xsize,
319             ysize,
320             zsize,
321             sPitch,
322             bandwidth,
323             fc);
324
325     //Copy current S to old S for error calculation
326     for (j = 0; j < ysize; j++) {
327       for (k = 0; k < xsize; k++) {
328         sOLD_h[k+j*xsize].x = s_h[k+j*xsize].x;
329         sOLD_h[k+j*xsize].y = s_h[k+j*xsize].y;
330       }
331     }
332
333     cudaThreadSynchronize();
334     //Copy new S to current S for error calculation
335     cudaMemcpy2DAsync(  s_h, xsize*sizeof(cufftComplex),
336               rM_d, rPitch,
337               xsize*sizeof(cufftComplex), ysize,
338               cudaMemcpyDeviceToHost, stream[s]);
339     cudaThreadSynchronize();
340
341     //Calculate Error using CPU
342     float temp = 0;
343     for (j = 0; j < ysize; j++) {
344       for (k = 0; k < xsize; k++) {
345         temp +=   sqrt(sOLD_h[k+j*xsize].x*sOLD_h[k+j*xsize].x
346                   + sOLD_h[k+j*xsize].y*sOLD_h[k+j*xsize].y)
347                   / sqrt(s_h[k+j*xsize].x*s_h[k+j*xsize].x
348                   + s_h[k+j*xsize].y*s_h[k+j*xsize].y);
349       }
350     }
351     temp /= ysize*xsize;
352     temp = abs(temp - 1.0f);
353     if (temp < minError) {
354       minError = temp;
355     }
356
357     cudaThreadSynchronize();
358     //Copy new S to current S for next iteration.
359     cudaMemcpy2DAsync(
360             (cufftComplex*)((char*)sM_d
361             + s*sPitch*ysize), sPitch,
362             (cufftComplex*)((char*)rM_d
363             + s*rPitch*ysize), rPitch,
364             xsize*sizeof(cufftComplex), ysize,
```

```
365             cudaMemcpyDeviceToDevice, stream[s]);
366     cudaThreadSynchronize();
367
368     i++;
369   } while (i < N && minError > EPS);
370   printf("Iterations used: %d\n",i+1);
371
372
373   timefinish = rdtsctime();
374   printf( "GPU: Time for GPU to compute convolution (in seconds)
375             (on hpc6): %f\n", ((double)
376             (timefinish-timestart)/2830000000));
377   err = cudaGetLastError();
378   if (err != cudaSuccess) printf("ERROR: %s
379             :ERROR\n", cudaGetErrorString( err ) );
380
381
382
383   printf("\n");
384   ////////////////////////////////////////////////////////
385   // COPY RESULT BACK TO HOST
386   ////////////////////////////////////////////////////////
387   timestart = rdtsctime();
388   cudaMemcpy2DAsync( rM_h, xsize*sizeof(cufftComplex),
389             sM_d, sPitch,
390             xsize*sizeof(cufftComplex), ysize,
391             cudaMemcpyDeviceToHost, stream[s]);
392   timefinish = rdtsctime();
393   printf( "COPY: Time to copy result to host (in seconds)
394           (on hpc6): %f\n", ((double)
395           (timefinish-timestart)/2830000000));
396   err = cudaGetLastError();
397   if (err != cudaSuccess) printf("ERROR: %s
398             :ERROR\n", cudaGetErrorString( err ) );
399
400
401
402   totfinish = rdtsctime();
403   printf( "\nTOTAL: total time for GPU operations without
404           allocations (on hpc6): %fms\n",
405           1000*((double) (totfinish-totstart)/2830000000));
406
407
408
409
410
411
412
413
```

```
414   totstart = rdtsctime();
415   ////////////////////////////////////////////////////////
416   //Compute FFT using FFTW
417   ////////////////////////////////////////////////////////
418   printf("\n");
419   timestart = rdtsctime();
420   printf( "Copying in-data to fftw matrix strutured
421           array (for FFTW)...\n");
422   for (i = 0; i < xsize; i++) {
423     for (j = 0; j < ysize; j++) {
424       for (k = 0; k < zsize; k++) {
425         cpueM[i][j][k][0] = src[k+j*zsize+i*zsize*ysize][0];
426         cpueM[i][j][k][1] = src[k+j*zsize+i*zsize*ysize][1];
427       }
428     }
429   }
430   timefinish = rdtsctime();
431   printf( "FFTW: Time to copy in-data to fftw matrix
432           strutured array (in seconds) (on hpc6): %f\n",
433           ((double)(timefinish-timestart)/2830000000));
434   timestart = rdtsctime();
435   printf("Performing FFT using FFTW...\n");
436   for (i = 0; i < xsize; i++) {
437     for (j = 0; j < ysize; j++) {
438       doFFT(cpueM[i][j], cpurM[i][j]);
439     }
440   }
441   timefinish = rdtsctime();
442   printf("FFTW: Time for FFTW to compute (in seconds)
443           (on hpc6): %f\n", ((double)
444           (timefinish-timestart)/2830000000));
445
446
447
448   ////////////////////////////////////////////////////////
449   //CPU "CONVOLUTION"
450   ////////////////////////////////////////////////////////
451   timestart = rdtsctime();
452   printf("Computing correlation using CPU...\n");
453
454   minError = 100000000000.0f;
455   cufftComplex **errorMatrix_h;
456   createMatrix(&errorMatrix_h,xsize,ysize);
457   for (j = 0; j < xsize; j++) {
458     for (k = 0; k < ysize; k++) {
459       errorMatrix_h[j][k].x = elementMatrix_h[j][k].x;
460       errorMatrix_h[j][k].y = elementMatrix_h[j][k].y;
461     }
462   }
```

```
463
464   i = 0;
465   do {
466     correlateReduce(  cpurM,
467               elementMatrix_h,
468               correlationMatrix_h,
469               xsize, ysize,
470               KERNEL_RADIUS, KERNEL_WF, MU, bandwidth, fc);
471
472     //Copy current S to old S for error calculation.
473     for (j = 0; j < xsize; j++) {
474       for (k = 0; k < ysize; k++) {
475         errorMatrix_h[j][k].x = elementMatrix_h[j][k].x;
476         errorMatrix_h[j][k].y = elementMatrix_h[j][k].y;
477       }
478     }
479
480     //Copy new S to current S for next iteration.
481     for (j = 0; j < xsize; j++) {
482       for (k = 0; k < ysize; k++) {
483         elementMatrix_h[j][k].x = correlationMatrix_h[j][k].x;
484         elementMatrix_h[j][k].y = correlationMatrix_h[j][k].y;
485       }
486     }
487
488     //Error calculation
489     float temp = 0;
490     for (j = 0; j < xsize; j++) {
491       for (k = 0; k < ysize; k++) {
492         temp += sqrt(errorMatrix_h[j][k].x
493                 *errorMatrix_h[j][k].x
494                 + errorMatrix_h[j][k].y
495                 *errorMatrix_h[j][k].y)
496                 /
497                 sqrt(elementMatrix_h[j][k].x
498                 *elementMatrix_h[j][k].x
499                 + elementMatrix_h[j][k].y
500                 *elementMatrix_h[j][k].y);
501       }
502     }
503     temp /= ysize*xsize;
504     temp = abs(temp - 1.0f);
505     if (temp < minError) {
506       minError = temp;
507     }
508
509     i++;
510   } while (i < N && minError > EPS);
511   printf("Iterations used: %d\n",i+1);
```

```
512
513   timefinish = rdtsctime();
514   printf("CPU1: Time for CPU to compute (in seconds)
515           (on hpc6): %f\n", ((double)
516           (timefinish-timestart)/2830000000));
517
518   totfinish = rdtsctime();
519   printf("\nTOTAL: total time for CPU computations
520           (on hpc6): %fms\n", 1000*((double)
521           (totfinish-totstart)/2830000000));
522
523   } //end for testing for-loop
524   loopfinish = rdtsctime();
525   printf("\nTOTAL: total time for loop including cast
526           to float (on hpc6): %fms\n", 1000*((double)
527           (loopfinish-loopstart)/2830000000));
528   printf("\n");
529
530
531
532   ////////////////////////////////////////////////////////
533   //DEALLOCATION (Only done once per program run)
534   ////////////////////////////////////////////////////////
535
536   for(i = 0; i < xsize; i++) {
537     for(j = 0; j < ysize; j++) {
538       free(cpueM[i][j]);
539     }
540     free(cpueM[i]);
541   }
542   free(cpueM);
543   for(i = 0; i < xsize; i++) {
544     for(j = 0; j < ysize; j++) {
545       free(cpurM[i][j]);
546     }
547     free(cpurM[i]);
548   }
549   free(cpurM);
550   cudaFreeHost(s_h);
551   cudaFreeHost(sOLD_h);
552   destroyMatrix(&elementMatrix_h, ysize);
553   destroyMatrix(&correlationMatrix_h, ysize);
554   cudaFree(elementMatrix);
555   cudaFree(cufftResultMatrix);
556   cudaFree(eM_d);
557   cudaFree(sM_d);
558   cudaFree(sOM_d);
559   cudaFree(rM_d);
560   cuda_cleanupFFT();
```

```
561   cleanupFFT();
562   for (i = 0; i < streams; i++) {
563     cudaStreamDestroy(stream[i]);
564   }
565 }
566 } //extern "C"
567
568
569
570 ////////////////////////////////////////////////////////
571 // READ DATA FROM FILE
572 ////////////////////////////////////////////////////////
573 void loadData(  fftw_complex *src,
574         char *fnstr,
575         char *logstr,
576         const int xsize,
577         const int ysize,
578         const int zsize,
579         char *matrixName) {
580
581   double *u;
582   long long int ret;
583   int n1,n2,n3,x,y,z;
584   n1 = xsize;
585   n2 = ysize;
586   n3 = zsize;
587
588   u = (double*)malloc( n1*n2*n3 * sizeof(double) );
589   for (y = 0; y < n1*n2*n3; y++) {
590     u[y] = 0.0;
591   }
592
593   //Read indata matrix from file.
594   ret = read_genfile(n1, n2, n3, fnstr, matrixName, u);
595   if ( ret ) {
596     write_log(logstr, "Matrix may not be loaded fully");
597   }
598
599   //Change from Row major of MATLAB to column major.
600   for (x = 0; x < n1; x++) {
601     for (y = 0; y < n2; y++) {
602       for (z = 0; z < n3; z++) {
603         src[z+y*n3+x*n2*n3][0] = u[z*n1*n2+y+x*n1];
604       }
605     }
606   }
607   free(u);
608   write_log(logstr, "Signal loaded");
609 }
```

```
610
611 ////////////////////////////////////////////////////////
612 // WRITE RESULT TO FILE
613 ////////////////////////////////////////////////////////
614 void saveData(  cufftComplex *result,
615         char *fnstr,
616         char *logstr,
617         const int xsize,
618         const int ysize,
619         const int zsize,
620         int harmonic,
621         char *realName,
622         char *imagName) {
623
624   int nh, n1, n2, x, y;
625   nh = harmonic;
626   n1 = xsize;
627   n2 = ysize;
628
629   long long int ret;
630   double *real, *imag;
631   real = (double*) malloc(n1*n2*sizeof(double));
632   imag = (double*) malloc(n1*n2*sizeof(double));
633
634   //Extract data from cufftComplex matrix
635   //into two seperate matrices
636   for (y = 0; y < n2; y++) {
637     for (x = 0; x < n1; x++) {
638       real[x*n1+y] = result[x+y*n1].x;
639       imag[x*n1+y] = result[x+y*n1].y;
640     }
641   }
642
643   //Write real data to file
644   ret = write_genfile(nh, n1, n2, fnstr, realName, real);
645   if ( ret ) {
646     write_log(logstr, "Real matrix saving failed.");
647   }
648   //Write imaginary data to file
649   ret = write_genfile(nh, n1, n2, fnstr, imagName, imag);
650   if ( ret ) {
651     write_log(logstr, "Imaginary matrix saving failed.");
652   }
653
654   free(real);
655   free(imag);
656
657   write_log(logstr, "Matrices stored");
658 }
```

```
659
660
661 ////////////////////////////////////////////////////////
662 // CUDA DEVICE INFORMATION
663 ////////////////////////////////////////////////////////
664 void initializeCuda(int device) {
665   //Select CUDA device to use.
666   cudaDeviceProp deviceProp;
667   cudaGetDeviceProperties(&deviceProp, device);
668   printf("Using Device %d: %s\n", device, deviceProp.name);
669   cudaSetDevice(device);
670 }
671
672 ////////////////////////////////////////////////////////
673 // ALLOCATE MATRICES
674 ////////////////////////////////////////////////////////
675 void allocate(  fftw_complex **src,
676                 cufftComplex **rM_h,
677                 const int xsize,
678                 const int ysize,
679                 const int zsize,
680                 const int streams) {
681
682   int totsize = xsize * ysize * zsize;
683   //Allocate page-locked memory for faster access.
684   cudaMallocHost( (void **) src, totsize *
685                   streams * sizeof(fftw_complex));
686   fillRandom(src[0], totsize*streams);
687
688   //Result from correlation copied to host //XSIZE*YSIZE
689   cudaMallocHost( (void **) rM_h, xsize * ysize
690                   * streams * sizeof(cufftComplex));
691 }
692
693 ////////////////////////////////////////////////////////
694 // DEALLOCATE MATRICES
695 ////////////////////////////////////////////////////////
696 void deAllocate(  fftw_complex *src,
697                   cufftComplex *rM_h) {
698
699   cudaFreeHost(src);
700   cudaFreeHost(rM_h);
701 }
```

F.2 GPU Correlation Kernel Code

The following is the comparison GPU kernel used in the iterations.

```
1  #define a3d(a,x,y,z,xs,ys,zs) a[x*ys*zs + y*zs + z]
2
3  __global__ void convolutionS( cufftComplex* d_Data,
4                  cufftComplex* d_Result,
5                  cufftComplex* d_S,
6                  const float mu,
7                  const int XSIZE,
8                  const int YSIZE,
9                  const int ZSIZE,
10                 const size_t pitch) {
11
12   const int bandwidthStart = FC - BANDWIDTH / 2;
13
14   const int x = threadIdx.x;
15   const int y = threadIdx.y;
16
17   const int xs = blockDim.x;
18   const int ys = blockDim.y;
19
20   const int blockStartx = blockIdx.x;
21   const int blockStarty = blockIdx.y;
22
23   const int globalx = x - KERNEL_RADIUS
24                       + blockStartx*xs - blockStartx*2*
                            KERNEL_RADIUS;
25   const int globaly = y - KERNEL_RADIUS
26                       + blockStarty*ys - blockStarty*2*
                            KERNEL_RADIUS;
27
28   const int smemPos = x + y*xs;
29   const int B = xs*ys;
30
31   //Set shared memory including aprons:
32   __shared__ cufftComplex data[11*11*(BANDWIDTH+1)];
33   __shared__ cufftComplex s[11*11];
34
35   //Optimization, uses less registers
36   bool active = false; //Inside matrix
37   bool write = false; //Inside computed result
38   if( globalx >=  0 && globaly >= 0
39       && globalx <= XSIZE-1 && globaly <= YSIZE-1) {
40     active = true;
41     if ((x >= KERNEL_RADIUS && x <= ((xs-1)-KERNEL_RADIUS)
42         && y >= KERNEL_RADIUS && y <= ((ys-1)-KERNEL_RADIUS))) {
```

```
43        write = true;
44      }
45    }
46
47    //If thread inside matrix
48    if (active) {
49      //Load S matrix:
50      //load data
51      cufftComplex *element =   (cufftComplex*) ((char*)d_S
52                               + globaly*pitch) + globalx;
53      s[smemPos].x = element->x;
54      s[smemPos].y = element->y;
55
56      //Load 3D data:
57      //for loop all planes
58      #pragma unroll 15
59      for (int i = 0; i < (BANDWIDTH+1); i++) {
60        //load data
61        data[smemPos + i*B].x = a3d(d_Data, globalx, globaly,
62                                    bandwidthStart+i, XSIZE,
63                                    YSIZE, ZSIZE).x;
64        data[smemPos + i*B].y = a3d(d_Data, globalx, globaly,
65                                    bandwidthStart+i, XSIZE,
66                                    YSIZE, ZSIZE).y;
67      }
68    }
69    //else
70    else {
71      //set to zero
72      s[smemPos].x = 0.0f;
73      s[smemPos].y = 0.0f;
74      //for loop all planes
75      #pragma unroll 15
76      for (int i = 0; i < (BANDWIDTH+1); i++) {
77        //set to zero
78        data[smemPos + i*B].x = 0.0f;
79        data[smemPos + i*B].y = 0.0f;
80      }
81    }
82
83    //synchronize all threads.
84    __syncthreads();
85
86
87    //Compute:
88    cufftComplex sum;
89    sum.x = 0.0f;
90    sum.y = 0.0f;
91    //if thread inside matrix == if pos-kernel_w > outside lower
```

```
      block
92   //and pos+kernel_w < outside upper block
93   if (write) {
94     cufftComplex R_klmn, W_klmn, M_klmn;
95     float R_kl, R_mn;
96
97     //R(k,l)
98     R_kl = 0.0f;
99     #pragma unroll 15
100    for (int i = 0; i < BANDWIDTH+1; i++) {
101      R_kl += data[smemPos+i*B].x
102                  * data[smemPos+i*B].x
103                  + data[smemPos+i*B].y
104                  * data[smemPos+i*B].y;
105    }
106    R_kl /= (BANDWIDTH+1);
107    R_kl = sqrtf(R_kl);
108
109    //for loop vertical k
110    #pragma unroll 5
111    for (int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++) {
112
113      //for loop horizontal l
114      #pragma unroll 5
115      for (int l = -KERNEL_RADIUS; l <= KERNEL_RADIUS; l++) {
116        //compute new S
117
118        R_mn = 0.0f;
119        R_klmn.x = 0.0f;
120        R_klmn.y = 0.0f;
121        W_klmn.x = 0.0f;
122        W_klmn.y = 0.0f;
123        M_klmn.x = 0.0f;
124        M_klmn.y = 0.0f;
125
126        //R(k,l,m,n)
127        #pragma unroll 15
128        for (int i = 0; i < BANDWIDTH+1; i++) {
129          R_klmn.x += data[smemPos+i*B].x
130                      * data[smemPos + k*xs + l + i*B].x
131                      + data[smemPos+i*B].y
132                      * data[smemPos + k*xs + l + i*B].y;
133          R_klmn.y += data[smemPos+i*B].y
134                      * data[smemPos + k*xs + l + i*B].x
135                      - data[smemPos+i*B].x
136                      * data[smemPos + k*xs + l + i*B].y;
137        }
138        R_klmn.x /= (BANDWIDTH+1);
139        R_klmn.y /= (BANDWIDTH+1);
```

```
140
141         //R(k-m, l-n)
142         #pragma unroll 15
143         for (int i = 0; i < BANDWIDTH+1; i++) {
144           R_mn +=  data[smemPos + k*xs + l + i*B].x
145                      * data[smemPos + k*xs + l + i*B].x
146                      + data[smemPos + k*xs + l + i*B].y
147                      * data[smemPos + k*xs + l + i*B].y;
148         }
149         R_mn /= (BANDWIDTH+1);
150         R_mn = sqrtf(R_mn);
151
152         //W(k, l, m, n)
153         W_klmn.x = R_klmn.x / (R_kl * R_mn + 1e-5);
154         W_klmn.y = R_klmn.y / (R_kl * R_mn + 1e-5);
155
156         //M(k, l, m, n)
157         M_klmn.x =  sqrtf(W_klmn.x*W_klmn.x
158                    + W_klmn.y*W_klmn.y) * R_klmn.x;
159         M_klmn.y =  sqrtf(W_klmn.x*W_klmn.x
160                    + W_klmn.y*W_klmn.y) * R_klmn.y;
161
162         //new_s
163         sum.x += (M_klmn.x * s[smemPos + k*xs + l].x
164                  + M_klmn.y * s[smemPos + k * xs + l].y)
165                  / (s[smemPos + k*xs + l].x
166                  * s[smemPos + k*xs + l].x
167                  + s[smemPos + k*xs + l].y
168                  * s[smemPos + k*xs + l].y + 1e-5);
169         sum.y += (M_klmn.y * s[smemPos + k*xs + l].x
170                  - M_klmn.x * s[smemPos + k * xs + l].y)
171                  / (s[smemPos + k*xs + l].x
172                  * s[smemPos + k*xs + l].x
173                  + s[smemPos + k*xs + l].y
174                  * s[smemPos + k*xs + l].y + 1e-5);
175       }
176     }
177     //calculate new S element
178     sum.x = (1.0f - mu) * s[smemPos].x + mu * sum.x;
179     sum.y = (1.0f - mu) * s[smemPos].y + mu * sum.y;
180
181     //write s to memory
182     cufftComplex *element = (cufftComplex*) ((char*)d_Result
183                             + globaly*pitch) + globalx;
184     element->x = sum.x/(KERNEL_W*KERNEL_W);
185     element->y = sum.y/(KERNEL_W*KERNEL_W);
186   }
187 }
```

F.3 CPU Correlation Code

The CPU version of the code used to compute the correction filter. Very similar to the GPU version.

```
1  void correlateReduce( fftwf_complex ***fftM,
2                        cufftComplex **sM,
3                        cufftComplex **corrM,
4                        int xsize,
5                        int ysize,
6                        int kernel_radius,
7                        int kernel_wf,
8                        float mu,
9                        int bandwidth,
10                       int fc) {
11
12   //Compute:
13   cufftComplex  sum, R_klmn, W_klmn,
14                 M_klmn, data, dataNeighbor, s;
15   int bwS = fc - bandwidth / 2; //bandwidthStart
16   float R_kl, R_mn, bandwidthf;
17   bandwidthf = bandwidth * 1.0f;
18   //for loop x-direction
19   for (int x = 0; x < xsize; x++) {
20     //for loop y-direction
21     for (int y = 0; y < ysize; y++) {
22       sum.x = 0.0f;
23       sum.y = 0.0f;
24
25       //R(k,l)
26       R_kl = 0.0f;
27       for (int i = 0; i < bandwidth+1; i++) {
28         data.x = fftM[x][y][bwS+i][0];
29         data.y = fftM[x][y][bwS+i][1];
30         R_kl +=   data.x  * data.x
31             +   data.y  * data.y;
32       }
33       R_kl /= (bandwidthf+1.0f);
34       R_kl = sqrtf(R_kl);
35
36       //for loop vertical k
37       for (int k = -kernel_radius; k <= kernel_radius; k++) {
38         //for loop horizontal l
39         for (int l = -kernel_radius; l <= kernel_radius; l++) {
40           //compute new S
41           R_mn = 0.0f;
42           R_klmn.x = 0.0f;
```

```
43            R_klmn.y = 0.0f;
44            W_klmn.x = 0.0f;
45            W_klmn.y = 0.0f;
46            M_klmn.x = 0.0f;
47            M_klmn.y = 0.0f;
48            //R(k,l,m,n)
49            for (int i = 0; i < bandwidth+1; i++) {
50              data.x = fftM[x][y][bwS+i][0];
51              data.y = fftM[x][y][bwS+i][1];
52              if (  x+l >= 0 && x+l < xsize &&
53                    y+k >= 0 && y+k < ysize) {
54                dataNeighbor.x = fftM[x+l][y+k][bwS+i][0];
55                dataNeighbor.y = fftM[x+l][y+k][bwS+i][1];
56              }
57              else {
58                dataNeighbor.x = 0.0f;
59                dataNeighbor.y = 0.0f;
60              }
61              R_klmn.x +=     data.x  * dataNeighbor.x
62                      +   data.y  * dataNeighbor.y;
63              R_klmn.y +=     data.y  * dataNeighbor.x
64                      -   data.x  * dataNeighbor.y;
65            }
66            R_klmn.x /= (bandwidthf+1.0f);
67            R_klmn.y /= (bandwidthf+1.0f);
68
69            //R(k-m,l-n)
70            for (int i = 0; i < bandwidth+1; i++) {
71              if (  x+l >= 0 && x+l < xsize &&
72                    y+k >= 0 && y+k < ysize) {
73                dataNeighbor.x = fftM[x+l][y+k][bwS+i][0];
74                dataNeighbor.y = fftM[x+l][y+k][bwS+i][1];
75              }
76              else {
77                dataNeighbor.x = 0.0f;
78                dataNeighbor.y = 0.0f;
79              }
80              R_mn +=   dataNeighbor.x  * dataNeighbor.x
81                  +   dataNeighbor.y  * dataNeighbor.y;
82            }
83            R_mn /= (bandwidthf+1.0f);
84            R_mn = sqrtf(R_mn);
85
86            //W(k,l,m,n)
87            W_klmn.x = R_klmn.x / (R_kl * R_mn + 1e-5);
88            W_klmn.y = R_klmn.y / (R_kl * R_mn + 1e-5);
89
90            //M(k,l,m,n)
91            M_klmn.x =  sqrtf(W_klmn.x*W_klmn.x
```

```
92                          + W_klmn.y*W_klmn.y) * R_klmn.x;
93              M_klmn.y =  sqrtf(W_klmn.x*W_klmn.x
94                          + W_klmn.y*W_klmn.y) * R_klmn.y;
95
96              //new_s
97              if (  x+l >= 0 && x+l < xsize &&
98                    y+k >= 0 && y+k < ysize) {
99                s.x = sM[x+l][y+k].x;
100               s.y = sM[x+l][y+k].y;
101             }
102             else {
103               s.x = 0.0f;
104               s.y = 0.0f;
105             }
106             sum.x +=  (M_klmn.x * s.x + M_klmn.y * s.y)
107                       / (s.x*s.x + s.y*s.y + 1e-5);
108             sum.y +=  (M_klmn.y * s.x - M_klmn.x * s.y)
109                       / (s.x*s.x + s.y*s.y + 1e-5);
110           }
111         }
112
113         sum.x = (1.0f - mu) * sM[x][y].x + mu * sum.x;
114         sum.y = (1.0f - mu) * sM[x][y].y + mu * sum.y;
115
116         //Write new correction matrix to memory:
117         corrM[x][y].x = sum.x;
118         corrM[x][y].y = sum.y;
119       }
120     }
121 }
```

F.4 Other Kernels

Other computational kernels used in the program.

```
1  /*
2   * Sets default startvalues for the correctionfactor matrix
3   */
4  __global__ void cuda_setStartValues(
5          cufftComplex *matrix, size_t pitch) {
6    cufftComplex *element = (cufftComplex*)
7          ((char*)matrix + blockIdx.y*pitch) + blockIdx.x;
8    element->x = 1.0f;
9    element->y = 0.0f;
10 }
```

```
1  extern __shared__ float sharedCast[];
2  /*
3   * Casts from double to float on the CUDA device.
4   */
5  __global__ void cuda_castFromDouble(
6                           double* src, float* eM, int size) {
7
8    int idx = threadIdx.x + blockIdx.x*blockDim.
9              x + blockIdx.y*blockDim.x*gridDim.x;
10   for (int i = idx; i < size; i+=blockDim.x*gridDim.x*gridDim.y)
11   {
12     eM[i] = (float) src[i];
13   }
14 }
```

F.5 Other CPU functions

Other functions used by the program.

```
1  cufftHandle plan;
2  /*
3   * Sets up the CUFFT: creating a plan and assigning
4   * the matrices that are to be computed.
5   */
6  void cuda_setupFFT(const int N, const int BATCH) {
7    cufftPlan1d(&plan, N, CUFFT_C2C, BATCH);
8  }
9
10 /*
11  * Run the FFT calculation.
12  */
13 cufftResult cuda_doFFT(cufftComplex *eM, cufftComplex *rM) {
14   return cufftExecC2C(plan, eM, rM, CUFFT_FORWARD);
15 }
16
17 /*
18  * Cleanup: Delete the plan.
19  */
20 void cuda_cleanupFFT() {
21   cufftDestroy(plan);
22 }
```

```
1  fftwf_plan p;
2
```

```
3  void setupFFT(  const int N, int xsize, int ysize, int zsize,
4              fftwf_complex *matrix, fftwf_complex *result){
5    p = fftwf_plan_dft_1d(N, matrix, result,
6        FFTW_FORWARD, FFTW_ESTIMATE);
7  }
8
9  void doFFT(fftwf_complex *matrix, fftwf_complex *result) {
10   fftwf_execute_dft(p,matrix,result);
11 }
12
13 void cleanupFFT() {
14   fftwf_destroy_plan(p);
15 }
```

```
1  /*
2   * "Constructor" for matrices.
3   */
4  void createMatrix(  cufftComplex **matrix, int xsize, int ysize,
5                      size_t &pitch) {
6    if (xsize == ysize) {
7      //cudaMallocPitch inserts correct padding for CUDA
8      cudaMallocPitch(  (void **) matrix, &pitch,
9                        xsize * sizeof(cufftComplex), ysize);
10   }
11   else {
12     //Allocate a 1D array
13     cudaMalloc( (void **) matrix,
14                 sizeof(cufftComplex) * xsize * ysize);
15   }
16 }
17
18 /*
19  * "Destructor" for matrices.
20  */
21 void destroyMatrix(cufftComplex **matrix) {
22   cudaFree(matrix);
23 }
```

```
1  //Round a / b to nearest higher integer value
2  #define IDIVUP(a, b) ( (a % b != 0) ? (a / b + 1) : (a / b) )
3  //Round a / b to nearest lower integer value
4  #define IDIVDOWN(a, b) ( a / b )
5  //Align a to nearest higher multiple of b
6  #define IALIGNUP(a, b) ( (a % b != 0) ?  (a - a % b + b) : a )
7  //Align a to nearest lower multiple of b
8  #define IALIGNDOWN(a, b) ( a - a % b )
```

```
1  /*
```

```
 * Fills a host fftw matrix with zeroes
 */
void fillRandom(fftw_complex **matrix, int size) {
  int i;
  for (i = 0; i < size; i++) {
    matrix[0][i][0] = 0.0;
    matrix[0][i][1] = 0.0;
  }
}
```

```
/*
 * Fills a host fftwf matrix with random numbers
 */
void fillRandom(  fftwf_complex ***matrix, int xsize,
                  int ysize, int zsize) {
  int i, j, k;
  for (i = 0; i < ysize; i++) {
    for (j = 0; j < xsize; j++) {
      for (k = 0; k < zsize; k++) {
        matrix[j][i][k][0] = 0.0f;
        matrix[j][i][k][1] = 0.0f;
      }
    }
  }
}
```

F.6 Makefile

The following is the Makefile used.

```
#
# Makefile for Abersim 2.0
#

CC = gcc
CFLAGS = -c -O2
NVCC = nvcc

INSTPATH = .
LIB = dynlib

dynlib: LIBFLAGS = -fPIC
dynlib: LD = gcc
dynlib: LDFLAGS = -shared
dynlib: LIB = dynlib
```

```
16
17 statlib: LIBFLAGS = -static
18 statlib: LD = ar
19 statlib: LDFLAGS = rcs
20 statlib: LIB = statlib
21
22 include ./Makefile.in
23
24 main: LIBFLAGS =
25 main: LD = gcc
26 main: MLDINCLS = $(LDINCLS) -L$(INSTPATH)
27 main: MINCLS = $(INCLS) -I$(INSTPATH)
28 main: MLIBINC = -labersim2 $(LIBINC)
29
30 HEADERS =    general/abersim_general.h \
31      io/abersim_io.h \
32      math/abersim_math.h \
33      material/abersim_material.h \
34      propagation/abersim_propagation.h \
35      simscripts/abersim_simscripts.h \
36      aberration/abersim_aberration.h \
37      general/propcontrol.h \
38      material/material.h \
39      aberration/phantom.h \
40      debug/debug.h \
41      locblas/locblas.h \
42      abersim2.h \
43      correction/abersim_correction.h \
44      correction/gpuOperations.h \
45      correction/cpuCorrelation.h \
46      correction/cpuFFT.h \
47      correction/gpuFFT.h \
48      correction/gpuCast.h \
49      correction/convolutionS.h \
50
51 GENOBJS =    general/init_log.o \
52      general/start_log.o \
53      general/update_log.o \
54      general/close_log.o \
55      general/write_log.o \
56      general/get_linspace.o \
57      general/perext.o \
58      general/get_window.o \
59      general/get_raisedcos.o \
60      general/get_xchannels.o \
61      general/apply_window.o \
62
63 IOOBJS =  io/export_beamprofile.o \
64      io/propio.o \
```

```
65      io/pulseio.o \
66      io/profileio.o \
67      io/abfileio.o \
68      io/matlabio.o \
69      io/genio.o \
70      io/get_strpos.o \
71
72  MATHOBJS =  math/minmax.o \
73      math/stats.o \
74      math/fftw2d.o \
75      math/interp1d.o \
76      math/bandpass.o \
77      math/hilbert.o \
78
79  MATOBJS = material/check_material.o \
80      material/isregular.o \
81      material/set_material.o \
82      material/get_matparam.o \
83      material/get_wavespeed.o \
84      material/get_massdensity.o \
85      material/get_betan.o \
86      material/get_attconst.o \
87      material/get_attexp.o \
88      material/get_attenuation.o \
89      material/get_compressibility.o \
90      material/get_epsn.o \
91      material/get_epsa.o \
92      material/get_epsb.o \
93      material/water.o \
94      material/muscle.o \
95      material/fat.o \
96      material/liver.o \
97      material/brain.o \
98      material/bone.o \
99      material/breast.o \
100     material/blood.o \
101     material/phantom.o \
102     material/seawater.o \
103
104 PRPOBJS = propagation/cpropagate.o \
105     propagation/diffract.o \
106     propagation/diffract_fourier.o \
107     propagation/diffract_fdtd.o \
108     propagation/burattsplit.o \
109     propagation/bursolve.o \
110     propagation/attsolve.o \
111     propagation/get_permutation.o \
112     propagation/get_wavenumbers.o \
113     propagation/assemble_wavenumbers.o \
```

```
114     propagation/get_kvecs.o \
115     propagation/get_attcoeffs.o \
116     propagation/adjust_attcoeffs.o \
117     propagation/get_diffmatrix.o \
118     propagation/get_diffstencil.o \
119     propagation/get_shocklength.o \
120     propagation/eqresample.o \
121
122 SIMOBJS =   simscripts/plan_simulation.o \
123     simscripts/beamsim.o \
124     simscripts/bodywall.o \
125
126 ABOBJS =  aberration/prepare_aberration.o \
127     aberration/prepare_delayscreen.o \
128     aberration/apply_aberration.o \
129     aberration/shift_array.o \
130
131 DBOBJS =  debug/chkpt.o \
132     debug/print_array.o \
133     debug/print_subarray.o \
134
135 BLASOBJS =  locblas/locccblas1.o \
136     locblas/loccdblas1.o\
137     locblas/loccsblas1.o\
138     locblas/locczblas1.o\
139     locblas/locccblas2.o\
140     locblas/loccdblas2.o\
141     locblas/loccsblas2.o\
142     locblas/locczblas2.o\
143     locblas/locccblas3.o\
144     locblas/loccdblas3.o\
145     locblas/loccsblas3.o\
146     locblas/locczblas3.o\
147
148 CORROBJS =  correction/correlation.o \
149     correction/gpuOperations.o \
150     correction/cpuCorrelation.o \
151     correction/cpuFFT.o \
152     correction/gpuFFT.o \
153     correction/gpuCast.o \
154     correction/convolutionS.o \
155
156
157 PROG =      abersim2
158 MAINOBJ =   $(PROG).o \
159
160 OBJS =  $(GENOBJS) $(IOOBJS) $(MATHOBJS) $(MATOBJS) \
161   $(PRPOBJS) $(SIMOBJS) $(ABOBJS) $(DBOBJS) $(BLASOBJS) $(
        CORROBJS)
```

```
162
163 DYNLIB = libabersim2.so
164 STATLIB = libabersim2.a
165
166 SUBDIRS =    general \
167      io \
168
169 dynlib: $(OBJS)
170   $(LD) $(LDFLAGS) $(OBJS) -o $(DYNLIB) $(LDINCLS) $(
        CUDA_LDINCLS) $(LIBINC) $(CUDA_LIBINC)
171
172 statlib: $(OBJS)
173   $(LD) $(LDFLAGS) $(STATLIB) $(OBJS)
174   make oclean
175
176 main: $(MAINOBJ)
177   $(LD) $(LDFLAGS) $(MAINOBJ) -o $(PROG) $(MLDINCLS) $(
        CUDA_LDINCLS) $(MLIBINC) $(CUDA_LIBINC)
178
179 all:
180   make statlib
181   make dynlib
182   make oclean
183   make main
184
185 $(MAINOBJ): $(PROG).c $(HEADERS)
186   $(CC) $(LIBFLAGS) $(CFLAGS) $(DBFLAGS) $(PLATFLAGS) $(MINCLS)
        $(CUDA_INCLS) $<
187
188 %.o: %.c $(HEADERS)
189   $(CC) $(LIBFLAGS) $(CFLAGS) $(DBFLAGS) $(PLATFLAGS) $(INCLS) $
        (CUDA_INCLS) $< -o $@
190
191 #--compiler-options == -Xcompiler
192 %.o: %.cu $(HEADERS)
193   $(NVCC) -Xcompiler $(LIBFLAGS) $(NVCCFLAGS) $(DBFLAGS) $(
        PLATFLAGS) $(INCLS) $(CUDA_INCLS) $< -o $@
194
195
196 clean:
197   make oclean
198   make libclean
199   make progclean
200
201 oclean:
202   $(RM) $(RMFLAGS) $(OBJS) $(MAINOBJ)
203
204 libclean:
205   $(RM) $(RMFLAGS) $(DYNLIB) $(STATLIB)
```

```
206
207 progclean:
208     $(RM) $(RMFLAGS) $(PROG)
209
210 install:
211     cp $(HEADERS) $(INSTPATH)include/.
212     cp $(DYNLIB)  $(INSTPATH)lib/.
213     cp $(STATLIB) $(INSTPATH)lib/.
214     make main
215     cp $(PROG) $(INSTPATH)bin/.
```

The following is the Makefile.in.

```
1 #
2 # Makefile.hpc06
3 #
4
5 DB0 = -DSILENT
6 DB1 = $(DB0) -DQUIET
7 DB2 = $(DB1) -DNOISY
8 DB3 = $(DB2) -DLOUD
9
10 DBFLAGS = $(DB1)
11
12 #nvcc debugflag
13 #NVCCFLAGS = -g
14
15 NVCCFLAGS = -c -O3 -use_fast_math --ptxas-options=-v --gpu-
       architecture sm_13 -g
16
17 CUDA_INCLS = -I/usr/local/cuda/include -I../../NVIDIA_CUDA_SDK/
       common/inc
18
19 CUDA_LDINCLS = -L/usr/local/cuda/lib -L../../NVIDIA_CUDA_SDK/
       common/lib/linux -L../../NVIDIA_CUDA_SDK/lib
20
21 CUDA_LIBINC = -lcuda -lcudart -lcutil -lcufft
22
23
24 PLATFLAGS =
25
26 INCLS = -I. -I/usr/local/include -I/usr/include
27
28 LDINCLS = -L/usr/lib -L/usr/local/lib -L. -L/usr/lib/atlas
29
30 LIBINC = -llapack_atlas -lcblas -latlas -lmatio -lm -lz -lg2c -
       lfftw3f -lfftw3
31
```

```
32 INSTPATH = ./simscripts
```

Appendix G

Results

The timing results from the 101 runs of the algorithm. Values are in milliseconds.

0	210.68	814.47
1	152.56	806.79
2	152.46	807.61
3	152.48	807.02
4	152.59	808.26
5	152.46	808.41
6	152.41	807.31
7	152.54	808.61
8	152.21	812.85
9	152.63	807.81
10	152.53	807.16

11	152.27	807.72
12	152.44	807.41
13	152.34	808.07
14	152.54	807.54
15	152.59	807.16
16	152.36	807.1
17	152.5	807.24
18	152.62	807.64
19	152.4	807.83
20	152.46	808.22

21	152.38	807.69
22	152.2	807.35
23	152.36	807.23
24	152.48	807.61
25	152.31	806.96
26	152.22	807.01
27	152.51	807.71
28	152.35	807.02
29	152.5	807.7
30	152.65	807.51

31	152.46	806.94
32	152.54	807.15
33	152.44	807.39
34	152.26	807.18
35	152.42	809.42
36	152.3	807.21
37	152.47	807.44
38	152.36	807.45
39	152.57	807.37
40	152.69	808.12

41	152.42	807.39
42	152.53	807.32
43	152.44	807.87
44	152.57	807.6
45	152.41	807.73
46	152.35	808.1
47	152.37	810.37
48	152.35	806.8
49	152.53	806.46
50	152.3	806.75

51	152.56	806.32
52	152.63	806.73
53	152.52	806.95
54	152.53	806.48
55	152.44	806.6
56	152.36	806.52
57	152.52	806.23
58	152.31	807.39
59	152.53	806.59
60	152.61	806.89

61	152.49	806.88
62	152.52	806.66
63	152.47	806.95
64	152.72	806.55
65	152.43	806.69
66	152.43	806.94
67	152.57	806.75
68	152.42	806.57
69	152.75	806.62
70	152.57	806.17

71	152.48	806.27
72	152.49	806.16
73	152.73	806.61
74	152.55	806.18
75	152.51	806.43
76	152.46	811.3
77	152.48	806.77
78	152.58	806.43
79	152.44	806.81
80	152.72	807.18

81	152.46	807.46
82	152.64	806.91
83	152.62	806.88
84	152.52	806.67
85	152.53	806.63
86	152.57	806.73
87	152.56	806.68
88	152.5	806.47
89	152.55	807.57
90	152.45	806.48

91	152.61	806.4
92	152.55	806.44
93	152.59	806.12
94	152.56	806.56
95	152.65	806.36
96	152.41	806.56
97	152.38	807.2
98	152.52	807.57
99	152.71	806.82
100	152.47	807.17