NTNU

Norwegian University of
Science and Technology

Implementing Ultrasound Beamforming on the GPU using CUDA

Lars Grønvold

Master of Science in Engineering Cybernetics
Submission date: May 2008
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

Todays ultrasound equipment consists mainly of a PC that is attached to several large cards that process the received signals in hardware. These cards take up a lot of space and are costly to develop. As processing power in PC's increase it is possible move some of this signal processing from specialized hardware to standard PC hardware. Such a transition from hardware over to software could open up new possibilities for new and more advanced signal processing that could increase the image quality.

The task is mainly about focusing ultrasound beams(beamforming) and can be divided in a theoretical and a practical part. First, study current and "future" ultrasound beamforming technology. Secondly, design algorithms and implement them on the NVIDIA Tesla GPU card. This GPU card provides a PC with high performance parallel number crunching.

Assignment given: 07. January 2008
Supervisor: Amund Skavhaug, ITK

Master thesis

Implementing ultrasound beamforming on the GPU using CUDA

Lars Grønvold

Advisor:
Thor Andreas Tangen

Supervisor:
Amund Skavhaug

Trondheim, May 30th, 2008

NTNU
Norwegian University of
Science and Technology

Faculty of Information Technology, Mathematics and Electrical Engineering
DEPARTMENT OF ENGINEERING CYBERNETICS

Preface

This thesis was submitted in fulfilment of the degree Master of Science at the Norwegian University of Science and Technology (NTNU), in the Department of Engineering Cybernetics.

I would like to thank the following people:

Amund Skavhaug for the cake. And for being my helpful supervisor.

Thor Andreas Tangen for being my helpful advisor.

Li Wei for everything else.

Contents

List of Figures

List of Tables

Abstract

This thesis discusses the implementation of ultrasound beamforming on the GPU using CUDA. Fractional delay filters and the need for it when implementing beamforming is discussed. An introduction to CUDA programming is given as well as a study of the workings of the NVIDIA Tesla GPU(or G80). A number of suggestions for implementing beamforming on a GPU is presented as well as an actual implementation and an evaluation of it's performance.

1 Introduction

1.1 Ultrasound Imaging

Ultrasound imaging is widely used in medicine. It allows a doctor to see inside the body of a patient for diagnosing(e.g. detecting cancer or heart diseases) or to guide surgery. Most people probably know about it from its use in obstetrical ultrasound, where it is used during pregnancy to check on the development of the fetus.

1.1.1 Ultrasound compared to other medical imaging techniques

Compared to other medical imaging techniques(like MRI, CT and X-ray) the advantages of ultrasound are(1):

- It images muscle and soft tissue very well and is particularly useful for delineating the interfaces between solid and fluid-filled spaces.
- It renders "“live"” images, where the operator can dynamically select the most useful section for diagnosing and documenting changes, often enabling rapid diagnoses.
- It has no known long-term side effects and rarely causes any discomfort to the patient.
- Equipment is widely available and comparatively flexible. Small, easily carried scanners are available; examinations can be performed at the bedside.
- Relatively inexpensive compared to CT and MRI.

On the downside:

- Limited penetration(3-25cm depending on frequency).
- Difficult to penetrate bone. For example, ultrasound imaging of the adult brain is very limited.
- Performs very poorly when there is a gas between the transducer and the organ of interest, due to the extreme differences in acoustic impedance.
- The method is operator-dependent. A high level of skill and experience is needed to acquire good-quality images and make accurate diagnoses.

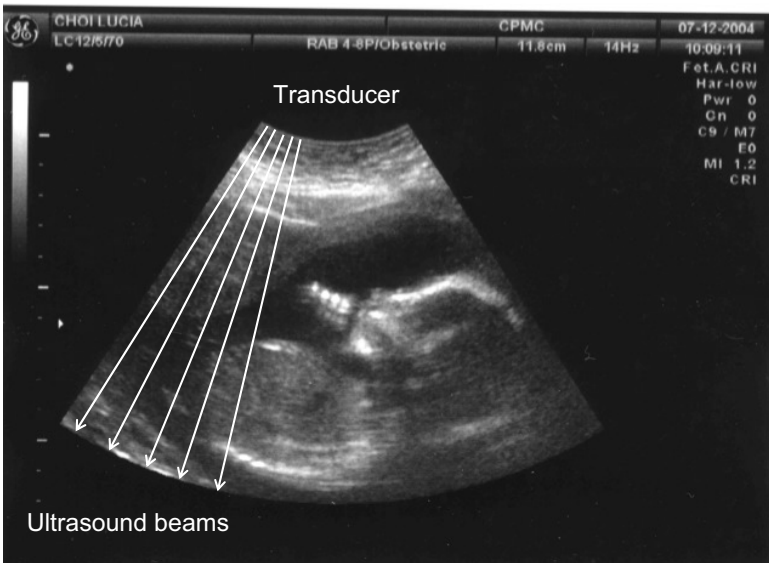

Figure 1.1: Ultrasound image of a fetus. With arrows indicating the direction of the scanlines/beams.

1.1.2 How it works

1.1 shows an ultrasound image of an unborn baby. The image is made out of several[1] scan lines(The orientation of the scanlines are illustrated by the arrows).

Each scan line is produced by one ultrasound pulse. The pulse is focused to form a narrow beam in the directions illustrated by the white arrows in the image. The pulse is partially reflected back when it hits boundaries between tissue of different density(see 1.2). The reflected pulses are recorded and the amplitude and time of the returned pulses gives an image of the tissue structure along the path of the beam(bright parts in the image are where pulses are reflected). Each pulse is transmitted in slightly different directions to get all the scanlines needed to make the complete image.

1.2 Beamforming

The ultrasound pulse is transmitted from a transceiver that has an array of small piezo-electric elements. Each element can transmit and receive ultrasound pulses independant of the other elements. When multiple elements transmits simultaneously, the combined pressurewaves produces an interference pattern. Figure 1.3 shows an interference pattern generated by eight wave sources. In the left plot, all the sources generates waves with the same phase, but in the

[1] about 100 scanlines would be my guess.

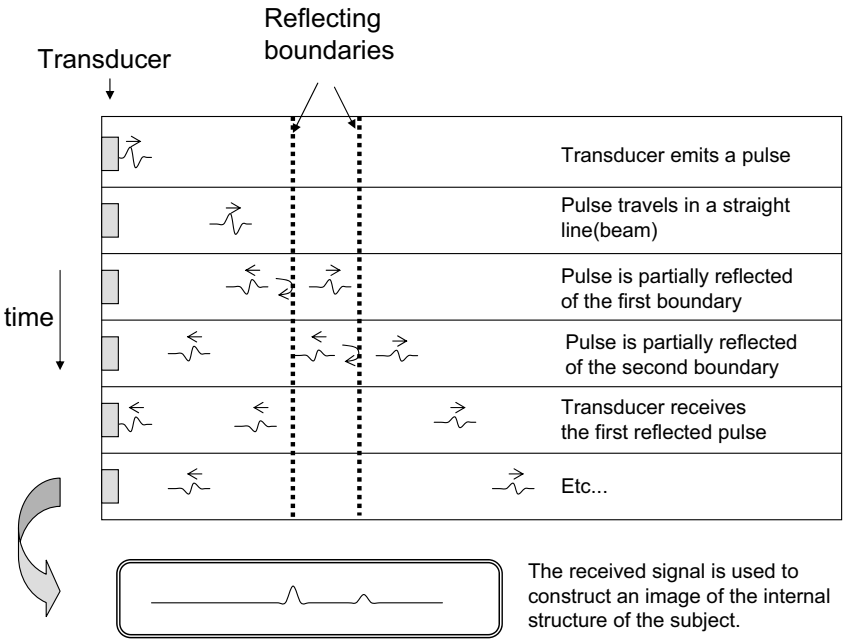

Figure 1.2: Illustration of how the pulse is reflected and image is reconstructed.

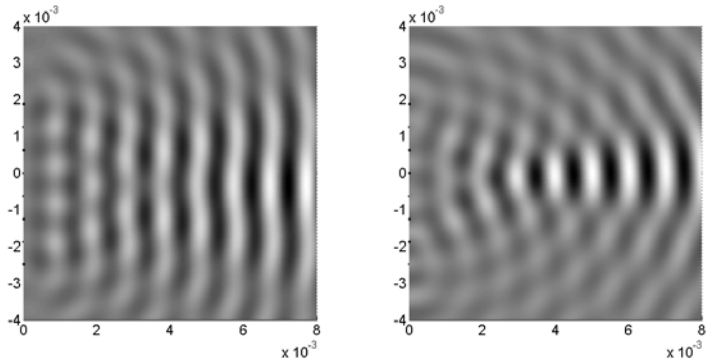

Figure 1.3: Interference pattern with 8 wave sources. a) no phase delay(unfocused) b) with phase delay(focused)

right plot, some of the waves are delayed in order to create a focused beam in the interference pattern.

Transmit beamforming is essentially the process of controlling the delay of the emitted pulse for each transducer element in order to form a focused beam.

Beamforming is also performed when receiving. The pulses that are reflected back to the transceiver will hit the individual transceiver elements and be recorded at slightly different times depending on their origin. The receive beamformer delays the signals from each channel(element) so that signals generated by pulses originating from along the beam will be synchronized. The sig-

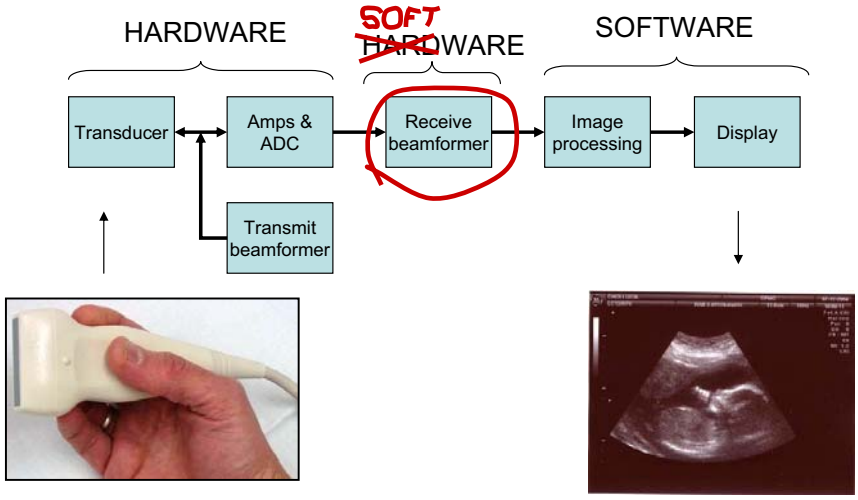

Figure 1.4: Block diagram of an ultrasound system. The time has come for beamforming to turn soft.

nals from all the channels are then summed, making the synchronized pulses stronger and the unsynchronized pulses weaker. In this way the beamformer focuses the signals that originates from the beam.

Receive beamforming is in a way the reverse of transmit beamforming. One important difference though, is that when transmitting the pulse, it is only possible to focus on one point, while when receiving, the delay can be adjusted dynamically to focus on all the points along the beam.

1.3 Software beamforming on GPU

The purpose of this thesis is to investigate the possibility of performing beamforming in software using GPUs.

1.3.1 Moving From Hardware to Software

As microprocessors get faster we see that more and more tasks that were previously implemented in hardware now is being implemented in software. One reason for this is that specially designed hardware and ASICs(Application Specific ICs) are produced in smaller volumes and therefore more expensive than general purpose processors. Another reason is that software easily can be upgraded as new versions are developed.

Receive beamforming requires a lot of computing power and is therefore usually performed on hardware using ASICs and FPGAs. GPU processing power has lately been evolving rapidly and might now make it possible to perform beamforming in software. Figure 1.5 shows how the processing power of the

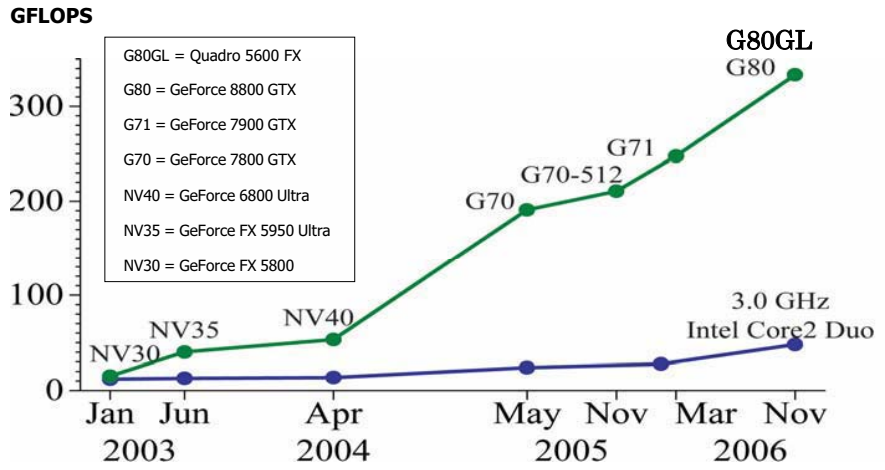

Figure 1.5: The evolution of FLOPS on GPU vs GPU for the last couple of years

GPU has been increasing and surpassed CPUs.

1.3.2 CUDA

CUDA ("Compute Unified Device Architecture"), is a technology that allows a programmer to use the C programming language to code general purpose algorithms for execution on the newest NVIDIA GPUs.

CUDA will be used in this project because it is a new and interresting technology and it allows for better control of the GPU than writing shaders.

1.3.3 Tesla C870 Specifications

Tesla C870 is a GPU card from NVIDIA that is specifically made for general purpose high perfomance computing. It has a GPU that is essentially the same as those used on video cards, but this card does not have any connection for a monitor.

Specifications for the Tesla C870(from nvidia.com[1]):

- Processor: based on the G80[2]

  Contains 128 processor cores running at 1.35GHz

- Memory: 1.5GB

- Memory bandwidth: 76.8GB/s peak

[1] http://www.nvidia.com/object/tesla_c870.html

[2]No documentation available on this particular chip. Assuming that it is the same as the G80

- Floating Point Precision: IEEE 754 single-precision
- Floating Point Performance: 430 GFLOPs achievable with a CUDA program. (512 peak[1])
- Power Consumption: 170W peak, 120W typical
- System Interface: PCI Express x16 (Generation 1)
- Compute capabilty: 1.0

1.3.4 Getting the data into device memory

Beamforming requires a lot of bandwidth. A typical transceiver could have 128 channels and use a 40 MHz sampling frequency. Each channel is typically 12-bit, but since the GPU does not support 12-bit data types it has to be converted to 16-bit. This would generate:

$$128 \times 40 \times 10^6 \times 2 = 9.54GB/s$$

All this data has to be transferred to the device memory(the memory on the GPU card) before it can be used. The Tesla card has PCI Express x16 (Generation 1) interface and can therefore not manage this datarate.

In this paper we will therefore assume that the data is already stored in device memory (as 16-bit signed integers).

[1] 128 cores $\times$ 1.35GHz $\times$ 3 operations(Dual issue multiply-add and multiply)

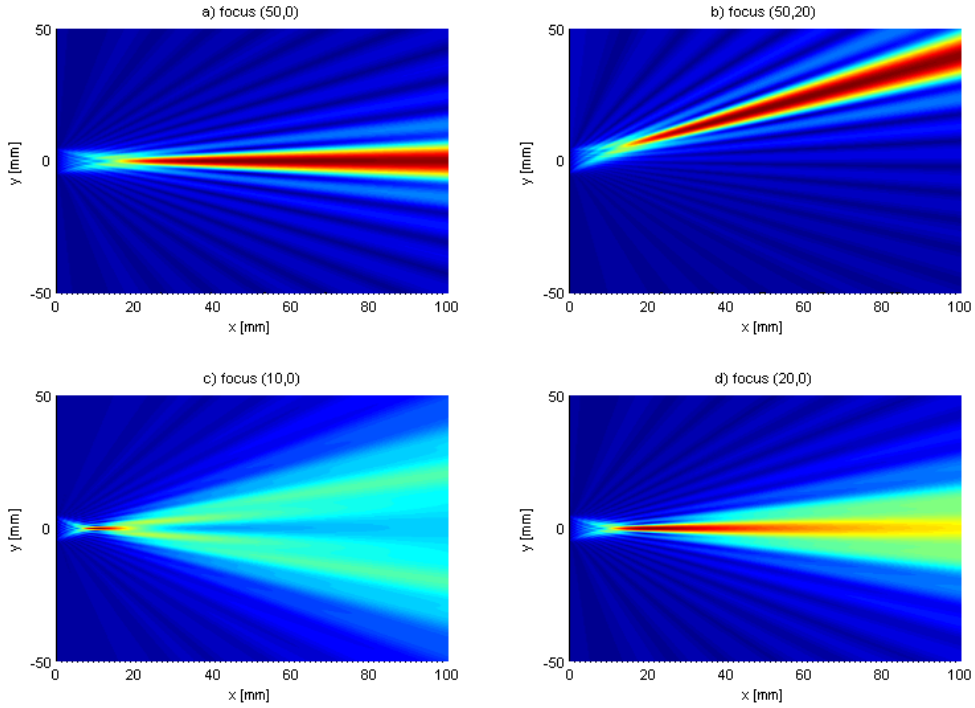

Figure 2.1: Shape of transmit beam with different focus points.

2 Beamforming

2.1 Transmit Beam Shape

Figure 2.1 shows an amplitude plot of a transmit beam given four different focus point[1]. Notice that the beam is thinnest at the focus point.

The shape of the beam is determined by delaying the pulses transmitted from each individual transducer element. The delay is calculated so that the pulses all reach the focus point at the same time and is therefore only dependent on the geometry of the transducer, the focus point and the speed the pulses travel[2].

2.2 Receive Beam Shape

As mentioned in the introduction, a receive beamformer delays the received signals and sums. The delays are calculated according to a focus point that moves along the beam. In section 4.3 the calculation of the delay is discussed in detail.

[1]The transducer in this case has a 1 cm aperture(it is 1 cm wide).
[2]the speed of sound through flesh is about 1560m/s. About the same as in water

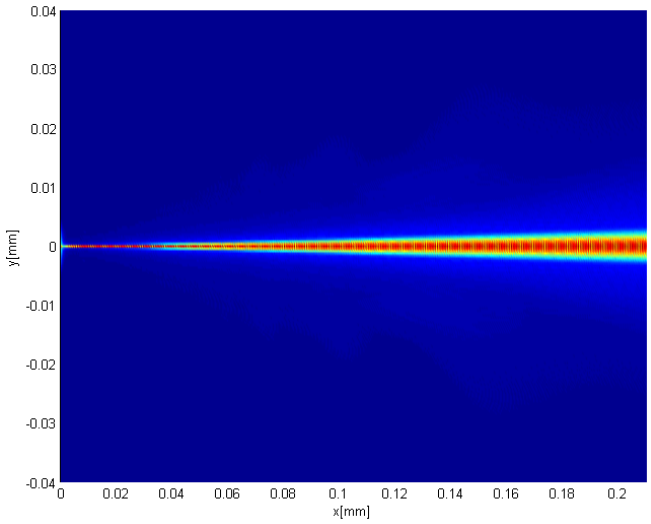

Figure 2.2: Shape of receive beam

Figure 2.2 shows an amplitude plot of the receive beam. The width of the receive beam is the same as the transmit beam at the focus point, but since the receive beam is focused along the whole beam while the transmit beam only has one focus point, the receive beam is generally thinner.

2.3 Apodization

If you look at the plots of the transmit beam shape you will se there are many weaker beams at each side of the beam, these are called sidelobes. The sidelobes can cause artifacts in the ultrasound image and is therefore not wanted. Apodization is a method to reduce the sidelobes and works by reducing the signal strength on the channels at the edge of the transducer. Figure 2.3 shows how the shape of the beam is affected by the apodization. The sidelobes almost completely disappears at the cost of a little wider beam.

Apodization is also used with receive beamforming and is implemented by multiplying the signals from the channels with a windowing function.

2.4 MLA

The speed of sound imposes a physical limitation to the pulse repetition rate. Example: If you want to scan to a depth of 10 cm, it will take $2 \times 0.1/1560 = 1.3ms$(two times the depth divided by the speed of sound) until the last pulse echo is received and the next pulse can be transmitted. If each frame is made out of 100 scanlines you can only construct about 8 frames per second which is far to slow to image moving parts like the heart. The solution is multiple line

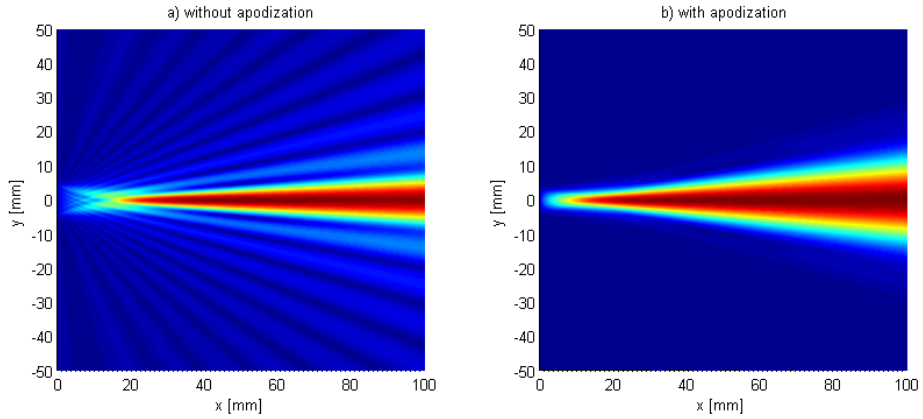

Figure 2.3: Shape of beam with and without apodization.

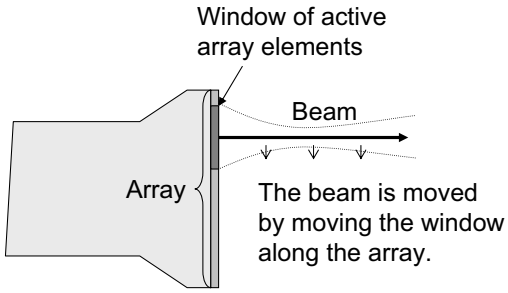

Figure 2.4: Linear array transducer

acquisition(MLA). MLA uses the received data from one pulse and performs the receive-beamforming several times to acquire multiple scanlines.

2.5 Transducer types

Transducers come in many shapes and sizes but the same principles of beamforming applies to any shape. The most common are the linear array transducer and the phased array transducer.

2.5.1 Linear array transducers

Linear arrays only use a window of the total elements when shooting a beam(see 2.4). That window is moved along the array to create several parallel beams that make up the whole picture.

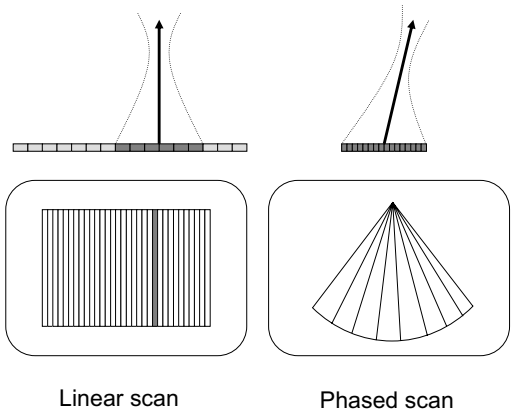

Figure 2.5: Linear and phased scan image shapes

There are also curved linear arrays which work in the same way, but the curve makes the beams fan out to create a wider image.

2.5.2 Phased arrays

Phased arrays use all of the array elements. The beam is steered in different directions to make up the image. The transducer is usually smaller to allow the beams to enter through narrow openings like between the ribs.

2.6 Fractional Delay Filters/Interpolation

As stated earlier, beamforming is just delaying signals and adding them. Adding signals is trivial but delaying is more complicated and will be discussed in this section.

2.6.1 Delaying Sampled Signals

A sampled signal can easily be delayed by a discrete number of samples by using a sample buffer. But as shown in figure 2.6, this produces a degraded result when performing delay and sum.

To produce a better result it is necessary to be able to delay the signal by a fractional number of samples. This can be achieved by using fractional delay filters.

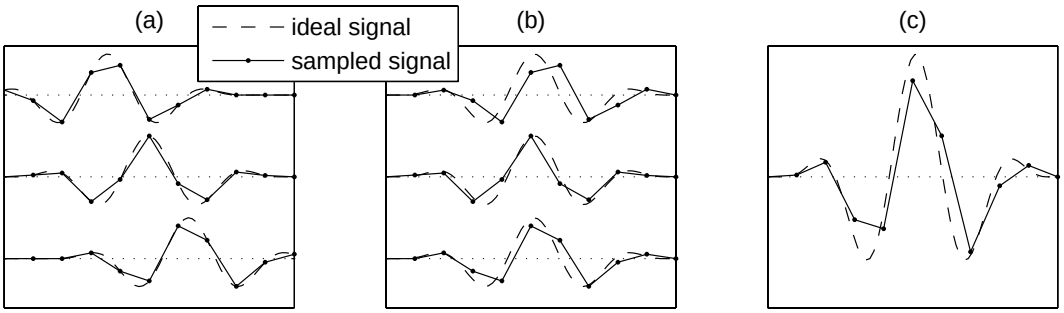

Figure 2.6: Delay and sum without interpolation.(a) Incoming signals. (b) Delayed signals. (c) Sum of delayed signals.

2.6.2 Ideal Fractional Delay Filter

A continuous signal can be delayed by an arbitrary amount using convolution with a dirac impulse:

$$x(t-\tau) = x(t) * \delta(t-\tau) \tag{1}$$

where x is the input signal, τ is the delay and $\delta(t-\tau)$ is the dirac function.

If the dirac function could be represented as a discrete signal it could be used as a FIR filter. The frequency response of the dirac can be found by fourier transformation:

$$\delta(t-\tau) \overset{\Im}{\rightarrow} e^{-2\pi j f \tau} \tag{2}$$

Which shows that the dirac function contains all frequencies i.e. $|e^{-2\pi j f \tau}| = 1$ for all f. A sampled signal can only represent frequencies up to half of the sampling frequency f_s. The frequency response outside that region is therefore removed by multipling with the *rect* function. The discrete filter can then be found by reverse fourier transform:

$$e^{-2\pi j f \tau}\text{rect}(f_s f) \overset{\Im^{-1}}{\rightarrow} \frac{\sin(\pi f_s(t-\tau))}{\pi f_s(t-\tau)} = \text{sinc}(f_s(t-\tau)) \tag{3}$$

Which shows that the ideal fractional delay filter is found by using the *sinc* function. If we define D as the delay given in number of samples and n as sample number we get the FIR filter coefficients:

$$h[n] = \text{sinc}(n-D) \tag{4}$$

Figure 2.7 shows the impulse response of the ideal fractional delay filter.

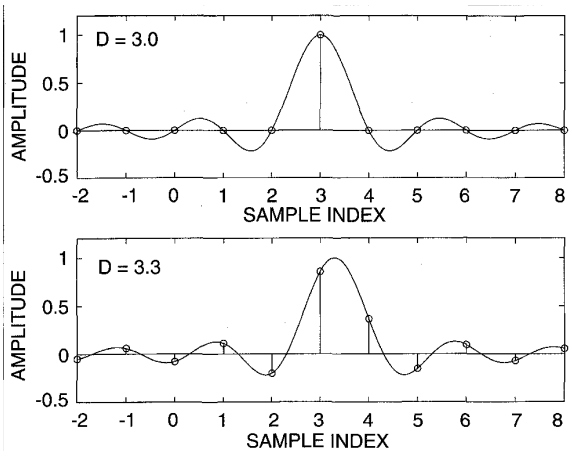

Figure 2.7: Impulse response of the ideal delay filter with the delay a) D = 3.0 and b) D = 3.3

2.6.3 Delay Filters

The ideal delay filter, i.e the sinc filter, is infinitely long, it is therefore purely theoretical. In practice it is necessery to make an approximation which has a limited number of filter coefficients. There are many ways to design such a delay filter. In this thesis only a few will be considered. Linear interpolation is implemented in the GPU hardware and will therefore be considered. The lagrange interpolation filter and the truncated sinc will also be considered because they are very easy to calculate and may be calculated on-the-fly on the GPU. More advanced filters needs to be pre-calculated. One such filter will be considered.

2.6.4 Choosing the Right Filter

The characteristics of a delay filter is usually measured by looking at the amplitude response and and phase delay response with respect to the normalized frequency. This response is also dependent on the amount of the delay and the length of the filter. All these factors combined with the amount of filter designs that are available, makes it hard to choose. In an attempt to make it easier to compare the filters a filter-test program has been developed that measures the performance of the filters. The filter-test tries to imitate what is actually happening when beamforming is performed. It works like this:

1. Generate a reference signal by combining several pulses with random arrival times and amplitudes. This signal should have the same charactestics and bandwidth as a typical signal received by an ultrasound transciever but with no noise.

2. Downsample the reference pulse to get a new reference pulse with the desired normalized frequency.

3. Delay and downsample the original reference pulse to get a number of delayed pulses with the desired normalized frequency. This simulates the time difference of the arrived signal on the different transducers. The delays are negative and evenly distributed between 0 and -1.

4. Add white noise to each of the delayed signals. This represents the uncorrelated noise that will occur in an actual system. The reason for including this is to see if the filters have any noise reducing effect.

5. Use the filters that are being tested to undo the negative delay created in the previous step. This simulates the delay filter in the beamformer and should reconstruct the reference signal generated in step 2.

6. Sum the reconstructed signals and divide by the number of different delays to get the mean reconstructed signal.

7. Subtract the mean reconstructed signal from the reference to get the noise. The noise is a combination of the added white noise and the error from the filters.

8. Calculate a signal-to-noise ratio between reconstructed signal and the noise using the formula in equation 5.

9. Repeat steps 2-8 for each frequency.

$$SNR = \sqrt{\frac{\sum signal[n]^2}{\sum noise[n]^2}} \tag{5}$$

2.6.5 Linear and Lagrange Interpolation

Linear interpolation can be implemented as a FIR filter with two coefficients $h[0] = 1 - D$ and $h[1] = D$. This is the same as a 1.order lagrange filter. The higher the order of the filter the closer it gets to the ideal sinc filter. The formula for generating the coefficients for a Nth order lagrange filter is

$$h[n] = \prod_{k=0, k \neq n}^{N} \frac{D-k}{n-k} \text{for} n = 0, 1, 2, ...N \tag{6}$$

Figure 2.8 shows the results from the filter-test program for lagrange filters of order 1,2,3,4 and 5. One thing to point out is that the filters of even order, i.e. 2nd and 4th order, seems to perform better than the odd ordered filters. Figure 2.9 shows that magnitude and phase response of lagrange filters with delay=0.5 samples. The even-ordered filters, i.e those with length L=3 and 5, have a poor phase delay response but a better magnitude response. The reason why the poor phase response does not affect the filter-test result is connected

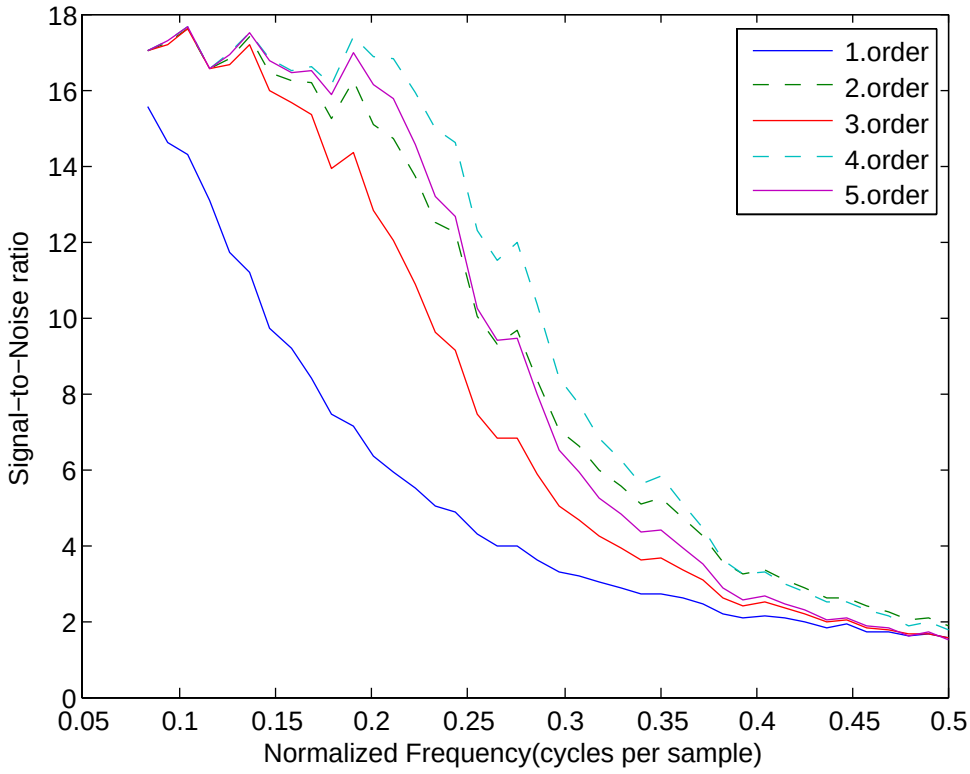

Figure 2.8: Signal-to-Noise ratio using lagrange filters

to the fact that the delays are evenly distrubuted between 0 and 1, causing an equal amount of phase shift in both directions. Indeed, when delays are limited to the range between 0 and 0.5 there is no difference in the test-results for even and odd length lagrange filters.

How the filter's phase response affects the image quality of an actual ultrasound image is still unclear. The filter-test can therefore not be completely trusted.

2.6.6 Truncated sinc filter

Another way to make a delay filter is to truncate the sinc filter to only a number of samples around the center. Figure 2.12 compares the truncated sinc filter with lagrange. The lagrange filter seems to perform better at lower frequencies but above 0.26 cycles/samples the sinc filter performs better.

2.6.7 LS Optimized Delay Filter

A more computationally demanding approach is to generate the filter by creating an optimal filter for certain frequencies. The idea is that it does not matter if

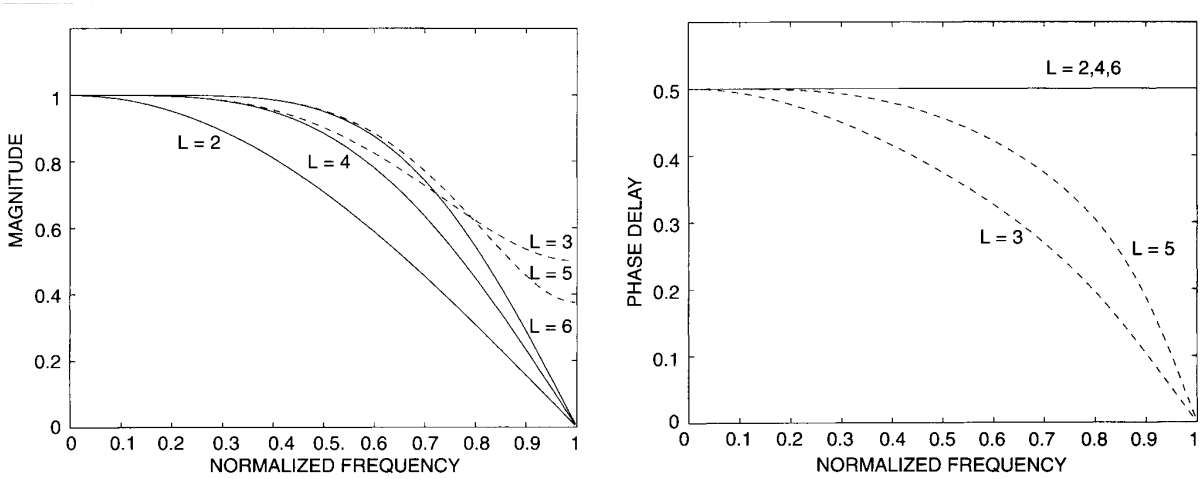

Figure 2.9: Magnitude and phase delay response of lagrange delay filters with 0.5 sample delay. L is filter length. (from (5). Note that the frequency is given in cycles per half-sample)

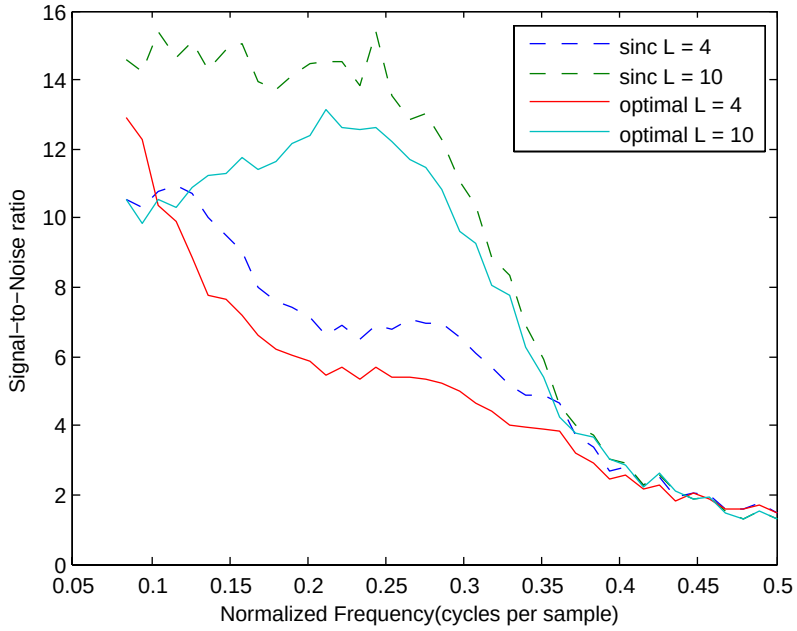

Figure 2.10: Signal-to-Noise ratio using truncated sinc filter and optimal filter.

the delay filter has a poor response for frequencies that does not exist in the signal. The frequency spectrum of the signal is estimated to be gauss distributed around the center frequency. The filter is genereted by using least square error estimation, resulting in a filter that gets best possible phase and magnitude response at frequencies near the center frequency(5).

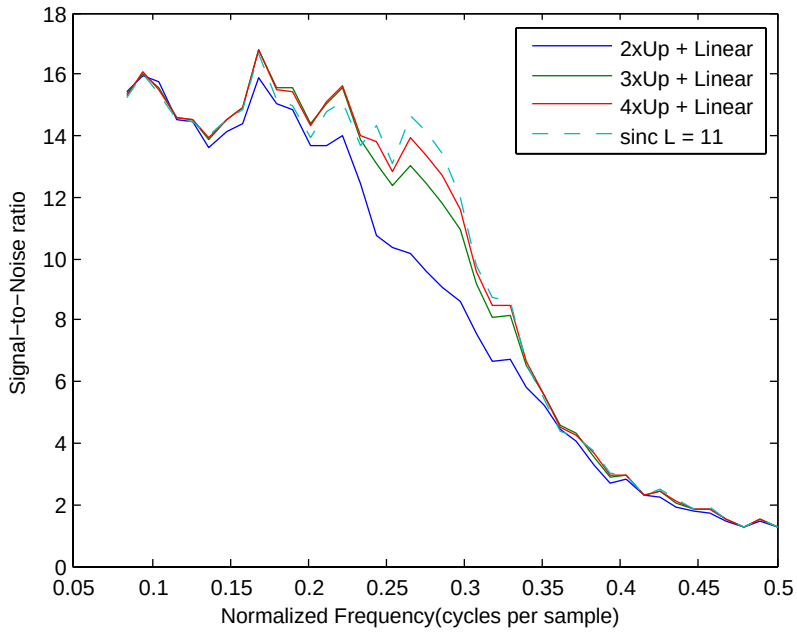

Figure 2.11: Signal-to-Noise ratio using truncated sinc filter to upsample followed by linear interpolation.

Figure 2.10 compares the optimal filter with the truncated sinc filter. The truncated sinc filter seems to have an overall better performance.

2.6.8 Upsample + Delay

When performing MLA beamforming, delay filters are applied to the same received signals several times to produce multiple scanlines. In this case it could be advantageous to first use a long filter to upsample the received data once, and then use a shorter filter (e.g. linear interpolation) on the upsampled data to calculate the delay for each scanline.

Figure 2.11 shows that when the signal has been upsampled to 3 or 4 times higher sampling rate, then using linear interpolation on the upsampled signal will produce almost as good results as using the filter used for upsampling.

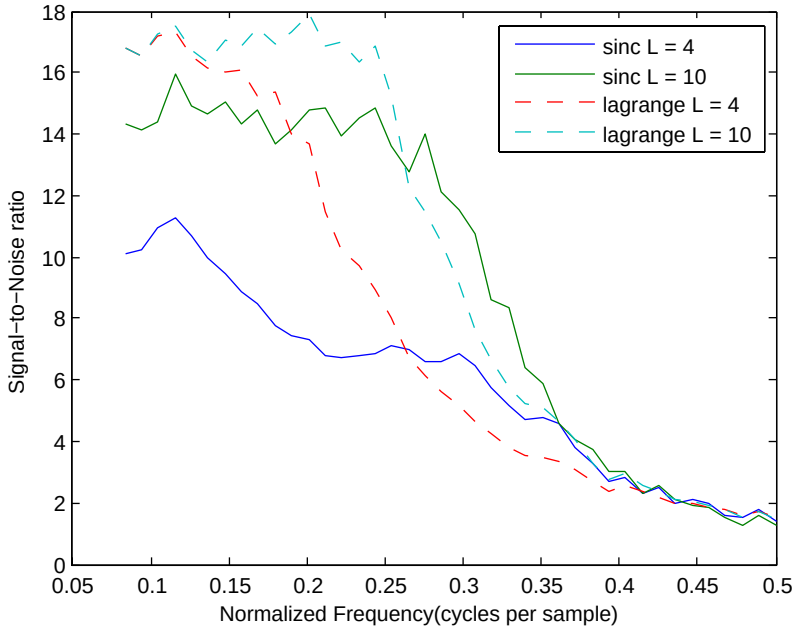

Figure 2.12: Signal-to-Noise ratio using lagrange and truncated sinc filters.

3 CUDA

This section gives a brief introduction to CUDA programming. For more details refer to the CUDA Programming Guide (3)(hereby referred to as The Guide).

In an attempt to make things easier to understand I refer to specific values (number of multiprocessors, cache size, performance tests, etc.) instead of generalizing. These values are valid for the Tesla C870 that was used in this project but may vary on other devices. Since there is little documentation on the C870 it is assumed that information on the G80 also applies to the C870.

3.1 The CUDA processor

3.1.1 Core architecture

The GPU can execute 128 instructions in parallel. It contains 16 multiprocessors(Also refered to as SM = Streaming Multiprocesor) that each has 8 processors(Also refered to as SP = Streaming Procesor), as illustarted in figure 3.1. The SM's have a Single Instruction, Multiple Data (SIMD) architecture: At any given clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data.

Each SM has on-chip memory of the four following types:

- 8192 32-bit registers divided among the processors.

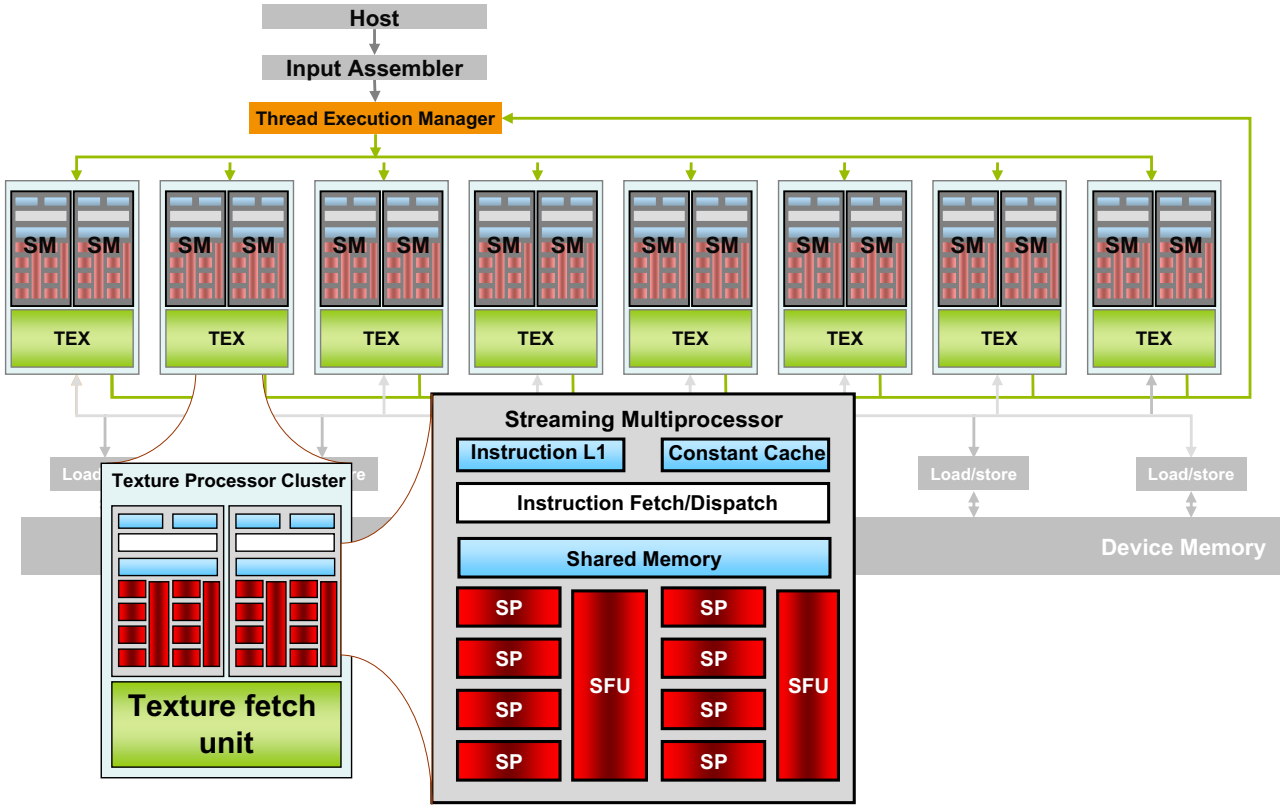

Figure 3.1: The G80 architecture

- 16Kb of shared memory.
- 8Kb read-only constant cache that is shared by all the processors and speeds up reads from the constant memory space, which is implemented as a read-only region of device memory. Only
- 8Kb read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory.[1]

The on-chip memory has virtually no latancy and is not limited by the bandwidth of the memory bus. When accessing the device(off-chip) memory there is a latency of about 400 to 600 clock cycles.

Each multiprocessor accesses the texture cache via a texture unit that implements the various addressing modes and (bi)linear interpolation which will be further discussed in 3.2.7.

[1]Actually there is 16Kb cache per texture unit, but each texture unit is shared by two SMs.

3.1.2 Execution model

The GPU is designed as a stream processor, which means that it takes a large set of data (a stream) and applies a series of operations (a kernel function) to each element in the stream. The processing of each element must be independent of each other so that they can be executed in parallel.

Another way of seeing it, is as if we have a large set of threads, each executing the same kernel function in parallel on its respective element of the dataset.

However, in the CUDA architecture, not all threads need to be independent. The threads are grouped into equally sized *blocks* of threads(max 512 threads per block). Whereas each *block* has to be completely independant of each other, the threads within the *blocks* can share data and synchronize their execution if neccessary. This is possible because each block is executed on only one multiprocessor.

The *thread execution manager* schedules the blocks for execution by passing it to one of the multiprocessors (see 3.2). Each multiprocessor can run up to 8 blocks at a time (max 768 threads[1]). The blocks that are running are called *active blocks*. When all the threads of an active block has finished, a new block is scheduled. This goes on until there are no more blocks in the *grid*[2]. A new kernel can only be launched when all the blocks of the previous launch have finished.

The SM divides the blocks into *warps* of 32 consecutive threads. *Active warps* (warps in the active blocks) are scheduled in a way that maximizes utilization and tries to hide the delays associated with memory loads. This means that every 4th clock cycle, the warp scheduler(instruction dispatch unit) selects one of the active warps that are ready to go(all operands are available), and loads the next instruction for that warp into the SPs. That instruction is then executed on all 8 processors for 4 clock cycles(once for each of the 32 threads of the warp).

3.1.3 The Special Functions Unit(SFU)

Each SM consists of 8 SPs and 2 Special Functions Units(SFU's). Each SP has one 32-bit, single-precision floating-point, multiply-add arithmetic unit that also can perform 32-bit integer arithmetic (but only 24 integer multiplication). The SFU on the other hand, is responsibel for calculating more complex floating-point operations like sine, cosine, reciprocal, reciprocal square root, log_2 and 2^x (sin, cos, rsqrt, rcp, lg2 and exp2)(4)(2). The SFU uses only one

[1] It is also limited by the amount of free registers and shared memory as will be discussed in 3.3.1

[2] The grid refers to the set of blocks in a kernel launch

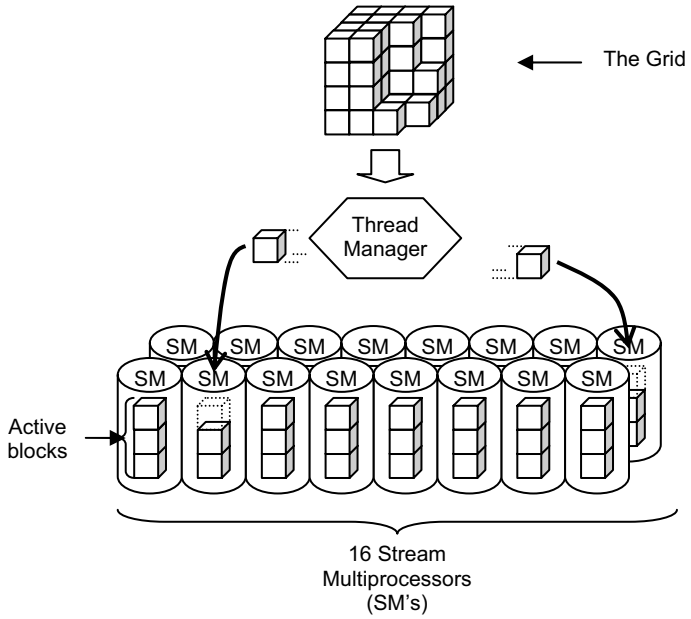

Figure 3.2: Illustration of how the blocks are scheduled for execution by the multiprocessors

clock cycle to execute one instruction, but sin and cos takes two instructions[1]. Since there is only 1 SFU per 4 SP's it needs to run the same instruction 4 times(on different data) which makes the SFU instructions effectively run 4 times slower than the normal instructions.

3.1.4 Speculations about the Instruction Issue rate

Every 4th cycle the instruction fetch/dispatch unit issues a new instruction to the SPs. Some instructions, i.e SFU instructions and texture fetch instructions needs to be issued to their corresponding units. The question is: Do these instructions need to through the SP's first?

While analyzing the performance of the implemented kernels it was found that sometimes the kernels actually executed faster than the expected optimal performance(see section 4.5.1). This implies that more than one instruction is issued per cycle.

It is possible that the instruction fetch/dispatch unit can load more than one instruction every 4 cycles. If two following instructions are independant it should be possible to dispatch one of them to the SPs and one to the texture unit or SFUs within 4 cycles.

[1]This is discussed in section 3.3.3

3.1.5 The Texture Fetch Unit

There is one texture fetch unit for every two SMs.

- (bi)linear interpolation.
- Type conversion.
- Address calculations.
- Load up to four 32 bit registers at one time.
- Runs in parallel with the SM's, thus making it possible to hide the time it uses. Issueing an instruction to the texture fetch unit takes only one clock cycle.
- Can load a total of 32(4 per tex unit) pixels per clock with bilinear filtering.

3.2 CUDA programming

3.2.1 Example of a CUDA program

Say we wanted to make a procedure that calculates the square of every element in an array. In C we would write something like this:

```
void square(float *in, float *out, int length){
    int i;
    for( i = 0; i < length; i++)
        out[i] = in[i] * in[i];
}
```

In CUDA it would be more natural to create a kernel that calculates the square of one element and launch one thread for each element of the array. The kernel would look something like this:

```
__global__ squareKernel( float *in, float *out)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;

    float x = in[i]
    out[i] = x*x;
}
```

Selecting the number of threads that will be launched is done by specifying the block and grid dimensions when the kernel function is called. In this example the block size is 128, and the grid size is calculated according to the length of the array so that threads = block size $\times$ grid size = arraylength.

```
dim3  dimBlock( 128 );
dim3  dimGrid( length/128 );

squareKernel<<< dimGrid, dimBlock >>>( in, out );
```

Listing 1 shows a more complete version of this example. Let's first take a closer look at the kernel code:

- Line 1: The `__global__` qualifier tells the compiler that the function is a kernel and is compiled to run on the GPU. Just like any normal C function, kernels can take anything as parameters but there is a limit to size. This kernel takes two pointers as parameters. Note that for kernels, pointers refer to global(device) memory. Also note that kernels don't have a return value.

- Line 3: `threadIdx`, `blockDim` and `blockIdx` are special built-in variables of the type `dim3`[1]. As the the function is executed once for each thread, `threadIdx` and `blockIdx` tells which thread and block is currently being executed. `blockDim` tells how many threads there are per block. The whole line calculates a unique index that can be used to access data. This is a very normal pattern.

- Line 5: Reads one element of the `in` array into the variable `x`. Variables like `x` will use a register on the GPU. The number of available registers is limited and using to many will cause poor performance.

- Line 6: Some calculation is done on the data and it is stored in the array `out`. Note that lines 5 and 6 accesses global memory and care should be taken to ensure coalescing, in this example there is no problem. Coalescing will be discussed later in section 3.3.2.

Because the kernel only can read from device memory it is neccessary to copy the data from host to device before the kernel is launched. Most of the code that runs on the host deals with allocating device memory and moving data to and from the device.

Launching the kernel and choosing block and grid size will be discussed in the following sections.

3.2.2 Launching a Kernel

Launching a kernel is done just like calling any other function except for the extra <<< >>> between the name of the kernel function and the parameters. It contains configuration parameters for the kernel launch.

[1] dim3 is one of the built-in vector types in CUDA

Listing 1: Simple CUDA program

```
1  __global__ squareKernel( float *in, float *out)
2  {
3      int i = threadIdx.x + blockDim.x*blockIdx.x;
4
5      float x = in[i]
6      out[i] = x*x;
7  }
8  void square(float *h_idata, float *h_odata, int length)
9  {
10     // allocate device memory
11     float *d_idata, *d_odata;
12     cudaMalloc( (void**) &d_idata, sizeof(float)*length);
13     cudaMalloc( (void**) &d_odata, sizeof(float)*length);
14
15     // copy data from host to device
16     cudaMemcpy( d_idata, h_idata, mem_size,
17                                 cudaMemcpyHostToDevice);
18
19     // choose block and grid size
20     dim3  dimBlock( 128, 1, 1);
21     dim3  dimGrid( length/dimBlock.x, 1, 1);
22
23     // launch the kernel
24     myKernel<<< dimGrid, dimBlock >>>( d_idata, d_odata );
25
26     // copy result from device to host
27     cudaMemcpy( h_odata, d_odata, sizeof(float) * length,
28                                 cudaMemcpyDeviceToHost);
29
30     // free memory
31     cudaFree(d_idata);
32     cudaFree(d_odata);
33 }
```

`<<< dim3 grid, dim3 block, size_t sharedMem, cudaStream_t stream >>>`

- grid: specifies how many blocks of this kernel that are to be launched. The maximum size of each dimension of the grid is 65535.
- block: specifies how many threads there are per block. The maximum size of the x-, y- and z-dimension of a block are 512, 512 and 64, respectively.
- sharedMem: specifies the amount of shared memory (bytes) that are allocated per block. This parameter is optional and defaults to 0.
- stream: defines which stream the kernel belongs to(will be discussed later in 3.2.6). This parameter is optional and defaults to 0.

3.2.3 Grid and Block Dimensions

When specifiyng how many threads and blocks that are to be launched, this is done with the vector type `dim3`. You can say that the threads are divided into a three dimensional grid of three dimensional blocks of threads. Each thread knows its position in the grid and block by the `dim3 threadIdx` and `dim3 blockIdx` variables. Organizing the threads in this way is convenient when the data you are working is organized in a similar way.

Example: If the data is a 1024×1024 texture and each thread is responsible for processing one element of the texture. The texture is to big to handle in one block so it needs to be divided into smaller subtextures, let's say, of the size 16×16. Then we can set the block dimensions to (16, 16, 1) and the grid dimensions to (64, 64, 1). The following kernel code calculates the x and y texture coordinates for the current thread:

```
x = threadIdx.x + blockIdx.x*blockDim.x;
y = threadIdx.y + blockIdx.y*blockDim.y;
```

3.2.4 Using Shared Memory

To access the shared memory you declare a variable in your kernel with the `__shared__` qualifier. Example:

```
__shared__ float mat[16][16];
__shared__ int n;
extern __shared__ float whatever[];
```

The size of the `extern __shared__` array is chosen at launch time[1].

Shared memory is limited to 16kB per multiprocessor which is diveded among the active blocks. The kernel will fail to launch if it requires more than 16kB of shared memory. Kernel function parameters are also stored in shared memory. Check the `.cubin` file to see how much shared memory your kernel is using.

[1] Shared memory size is the third kernel launch configuration parameter. Refer to 3.2.2

3.2.5 Using Constant Memory

Using constant memory can in some cases greatly increase the performance. Constant memory is cached and on a cache hit it is as fast as reading from a register as long as all the threads in the half-warp[1] reads from the same address. Constants are declared as global variables with the `__constant__` qualifier:

```
__constant__ float pi = 3.14;
__constant__ int sequence[3] = {1,2,3};
__constant__ char notSoConstantString[100];
```

It is possible to write to the constants from the host using `cudaMemcpyToSymbol()`:

```
char str[] = "Hello World!";
    cudaMemcpyToSymbol(notSoConstantString, str, strlen(str));
```

Constant memory cache on each multiprocessor is 8Kb. The total amount of constant memory is 64Kb. Declaring more than 64Kb of constants will give a compiler error.

3.2.6 Asynchronous execution and streams

Kernel launches are asynchronous, meaning that the host will continue execution in parallel with the GPU. If several kernels are launched, they will be launched in sequence after the previous has finished.

Memory operations can also be run asynchronously by using the `cudaMemcpyAsync()` and similar functions with the `Asynch` suffix.

Compute capabilty 1.1[2] allows concurrent execution of memory copies and kernel execution. To control the sequence of execution all operations are assigned to a *stream*. Operations in the same stream will be executed in sequence while operations in different streams may execute in parallel.

For now it is not possible to have several kernels running concurrently, but this might be implemented in future versions of CUDA.

3.2.7 Using textures

The main reasons for using textures is that:

- Texture reads are cached.
- Data can be accessed in a random fashion. No need to worry about coalescing.

[1] A half-warp is either the first 16 or last 16 threads of a warp.
[2] The Tesla C870 has compute capability 1.0

There are two different ways to use textures in CUDA. The difference comes from whether the texture reference is bound to linear memory (using `cudaMalloc()` and `cudaBindTexture()`) or a CUDA array (using `cudaMallocArray()` and `cudaBindTextureToA` Each way has its advantages and disadvantages:

- Texturing from CUDA arrays

  Supports 2D textures with maximum width=2^{16} and maximum height=2^{15}.

  Supports 1D textures with maximum width=2^{13}.

  Supports (bi)linear interpolation

  Supports normalized texture coordinates(values in the range [0.0, 1.0]).

  Supports alternative addressing modes for out-of-bound texture coordinates(only with normalized coordinates): Clamp, wrap or mirror.

  It is not possible to write to a CUDA array from a kernel.

- Texturing from device memory(linear addressing)

  No 2D textures.

  Supports 1D textures with maximum width=2^{27}.

  No interpolation.

  Integer texture coordinates.

  Out-of-bound texture coordinates return 0.

  Writing to texture is done just like writing to global memory.

The following list shows the usual stages that is implemented when using textures and which functions to use for both linear and array textures:

- Declare a texture reference.

  Both: `texture<Type, Dim, ReadMode> texRef;`

- Create a channel descriptor.

  Both: `cudaCreateChannelDesc()`

- Allocate or free device memory.

  linear: `cudaMalloc()` and `cudaFree()`

  array: `cudaMallocArray()` and `cudaFreeArray()`

- Transfer data between host and device.

  linear: `cudaMemcpy()`

  array: `cudaMemcpyToArray()`

- Bind texture reference to data.

  linear: `cudaBindTexture()`

  array: `cudaBindTextureToArray()`

- Fetch from texture(in kernel).

  linear: `tex1Dfetch(texRef, int x)`

  array: `tex1D(texRef, float x)` or `tex2D(texRef, float x, float y)`

The channel descriptor defines the format of the texture. The `Type` parameter of the texture reference defines the datatype that is returned when fetching from that texture and should match the format defined in the channel descriptor. An exception to this, and a nice feature, is that 8-bit and 16-bit integer input data can be converted to 32-bit normalized floating-point values i.e values in the range [0.0, 1.0] or [-1.0, 1.0]. This is done by setting `ReadMode` to `cudaReadModeNormalizedFloat`. The default is `cudaReadModeElementType` where no conversion is performed.

3.3 Optimization

Writing CUDA code is very much like writing c code, but if you want to take full advantage of the speed of the GPU you have to carefully design your algorithms to match the architecture of the GPU. This section discusses the most important points.

3.3.1 Occupancy

The multiprocessor can have up to 24 active warps and schedules them to maximize the computational power and hide the delays associated with device memory access. However there are several factors that may limit the number of warps that can be active. These are:

- Block size: Since only whole blocks can be loaded, the size of the blocks(in number of warps per block), must be a factor of 24 warps to achieve full warp occupancy. Maximum block size is 16 warps(512 threads). Maximum number of active blocks[1] is 8. This leaves only 5 optimal choices when choosing the block size: 96[2],128, 192, 256 or 368 threads (3, 4, 6, 8 or 12 warps)

[1] active blocks = blocks running concurrently on the multiprocessor

[2] The Guide states that the block size should be a multiple of 64 to minimize register bank conflicts.

- Shared memory: The available shared memory(16kB) will be divided among the active blocks. Therefore, the number of active blocks is limited by how much shared memory each block uses. A good rule of thumb is to use no more than 5 floats per thread (16kB/768 $\approx$ 21 bytes $\approx$ 5 floats). Be aware that also kernel parameters are stored in shared memory.
- registers: Each multiprocessor has 8192 registers that are divided among the active threads. To be able to have the maximum of 768 active threads each thread must not use more than 8192/768 $\approx$ 10 registers. To find out how many registers the kernel is using you can check the .cubin file that the CUDA compiler can generate for you.

Note that compute intensive kernels might run just as fast without full occupancy.

3.3.2 Coalescing

When accessing global(device) memory each thread in a half-warp should cooperate to read/write one contiguous chunk of memory whose address is aligned to the size of the chunk. Failing to do this, the memory access will be split into several parts and performance could be lost.

More precisely, as The Guide(3) states: In each half-warp, thread number N within the half-warp should access address

$$HalfWarpBaseAddress + N$$

where HalfWarpBaseAddress is of type `type*` and `sizeof(type)` is 4,8 or 16 (32-bit, 64-bit or 128-bit). Moreover, HalfWarpBaseAddress should be aligned to `16*sizeof(type)` bytes (i.e. be a multiple of `16*sizeof(type)`).

Note that if a half-warp fulfills all the requirements above, the per-thread memory accesses are coalesced even if some threads of the half-warp do not actually access memory.

3.3.3 Instruction throughput

Most intructions take only one clock cycle to execute, including floating-point multiply-add instructions. There are however some instructions that take more than one cycle. Here is a list of some of those that where mentioned in section 5.1.1.1 of The Guide(3)[1]:

- 4 cycles: `1/x` (reciprocal)
- 4 cycles: `rsqrt(x)` (reciprocal square root)

[1]The Guide states the number of cycles one multiprocessor(8 processors) takes to execute one instruction for one warp(32 threads) and therefore has 4 times higher values.

- 4 cycles: `__log(x)` (fast version of log)
- 4 cycles: 32-bit integer multiplication. Note that you can use `__mul24` or `__umul24`, which gives you 24-bit integer multiplication in one cycle.
- 5 cycles: `fdividef(x,y)` (faster floating-point division)
- 8 cycles: `sqrt(x)` (is implemented as `1/rsqrt(x)`)
- 8 cycles: `__sin(x)` (fast version of `sin(x)`)
- 8 cycles: `__cos(x)` (fast version of `cos(x)`)
- 9 cycles: `x/y`

All the math functions that are available in C (like `sin(x)`) seems to be available in CUDA. Many of these math functions has a faster version that has the same name but with two undescores prefixed (like `__sin(x)`). If you specify the compiler flag `--use-fast-math` then only the fast functions are used. Be aware that the fast functions are not as accurate. For a full listing of math functions and their respective accuracy refer to appendix B in The Guide(3).

As discussed in section 3.1.3, the arithmetic functions `sin`, `cos`, `rsqrt`, `rcp` and `lg2` are implemented on the SFU. There is one SFU per 4 SP's, which can explain why instructions like `1/x` and `rsqrt(x)` are said to take 4 cycles. This indicates that the SFU actually executes one instruction per cycle but has to execute 4 instructions to keep up with the SP's.

The instructions that are said to take more than 4 cycles are actually compiled to several instructions. Decuda shows that the sine function is divided into two instructions called `pre.sin` and `sin`. It also shows that `x/y` compiles to several instructions that multiplies x and y with 0.25 if $|y| > 8.5 \cdot 10^3 7$ before the actual division, which is implemented with `rcp` and `mul`.

The SFU can run in parallel with the SP. This implies that the latancy of the SFU can be hidden by scheduling other threads while the SFU is busy. The result of this is that SFU instruction can effectively execute in one cycle if less than 1/4 of the instructions are SFU instructions.

3.3.4 Bandwidth

The Tesla Card has a peak bandwidth of 76.8GB/s but it in practice the bandwidth is a bit lower. Table 1 shows the result of of a test where each thread performs a read and/or write. The kernels use either float(32-bit), float2(64-bit) or float4(128-bit). As you can see, the actual bandwidth varies with how the memory is accessed.

Figure 3.3 and 3.4 shows how gridsize and blocksize affects the bandwidth. In both cases it is clear that it is necessary to launch many blocks to get good

	float	float2	float4
read and write	61.62 GB/s	61.13 GB/s	47.21 GB/s
read only	38.61 GB/s	53.64 GB/s	27.69 GB/s
write only	51.10 GB/s	51.96 GB/s	52.44 GB/s
texture read only	43.72 GB/s	51.00 GB/s	55.44 GB/s
texture read and write	58.52 GB/s	59.92 GB/s	60.49 GB/s

Table 1: Bandwidth test.

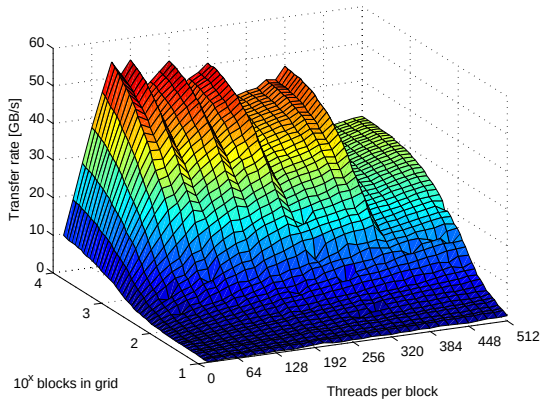

Figure 3.3: Bandwidth test plot. Read only. Each thread reads one 32-bit float.

bandwidth. The case in figure 3.3 only read is performed and there is a clear pattern of peaks at the blocksizes that give maximum occupancy (as discussed earlier in 3.3.1). It also seems that the smaller blocksizes are faster. The memory latancy is probably limiting the performance here. In figure 3.4 where both read and write is performed it seems that occupancy isn't that important but there is still a slightly better performance at the optimal blocksizes and a significant drop as the occupancy drops to 50% when the blocksize passes 384.

3.3.5 The Compiler

When compiling CUDA code `nvcc` separates the GPU code from the CPU code. The CPU code is compiled by a normal C compiler. The GPU code is first translated to PTX code, which is an assembly-like device-independent language. The PTX code is then compiled to a device-dependant binary.

By using the `-keep` option when invoking `nvcc`, all intermediate files are kept. Among them are the PTX code (`.ptx`) and `.cubin` file which can be very usefull. The `.cubin` lets you know how much shared memory and how many registers each kernel uses.

There exists a disassembler called `decuda` made by a third party. It takes the

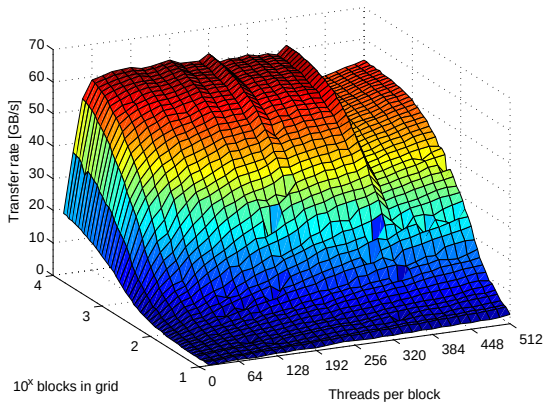

Figure 3.4: Bandwidth test plot. Read and write. Each thread reads and writes one 32-bit float.

binary code from the `.cubin` file and translates it back to a PTX-like language. Unfortunately it does not understand the whole instruction-set, but it will give you a better idea of what the actual code is doing than by just looking at the PTX code.

4 Implementation

This section discusses different ways to implement beamforming on the GPU using CUDA. Several methods have been implemented and tested. A detailed analysis of the code and how it performs is presented. The analysis helps give a better understanding of the G80 architecture and how to write optimized applications for it.

4.1 How to Implementat the Fundamental Parts of a Beamformer

Figure 4.1 shows the basic building blocks that make up a beamformer[1]. Each block needs to do calculations for every sample from every channel and therefore has the same order of impact on the overall performance.

4.1.1 Calculate Delay

Each sample is delayed by a different amount. This delay is represented as an floating point index into the receive buffer which is sent to the interpolator. The calculation of the delay varies with implementation but there are two main different approaches:

- Calculate delays on-the-fly

  (computationally expensive)

- Load pre-calculated delays

  (requires bandwidth)

[1]The arrows indicate dataflow. The gray ones are optional depending on the implementation.

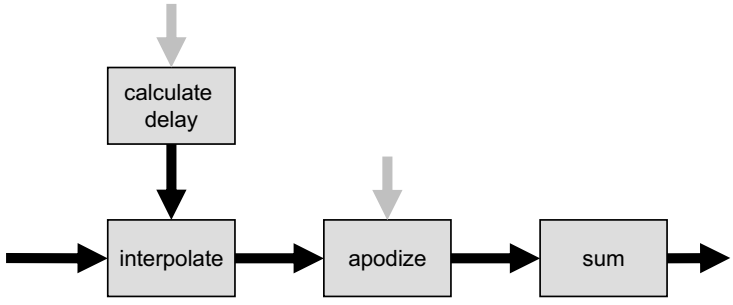

Figure 4.1: Fundamental building blocks of a beamformer

Which approach to choose relies on rest of the beamformer implemetation. To take full advantage of the GPU there needs to be a balance between bandwidth use and arithmetic operations, e.g. If there is no bandwidth to spare it makes sense to calculate delays on-the-fly.

4.1.2 Interpolate

Using the texture unit to perform linear interpolation frees the SM from the job and will therefore always be the fastest way. But as discussed in section 2.6 linear interpolation is not always good enough.

One way to make use of the texture unit interpolator and still get good results is to upsample received signals before interpolation. One problem with this approach is that the amount of data increases and more bandwidth is required. However, when doing MLA beamforming the same data is read several times which makes it possible utilize the cache and reduce the bandwidth requirement.

The other approach is to implement the interpolation as a fractional delay filter. One problem with this is that the coefficients of the delay filter needs to be calculated for each sample which is very computationally expensive. It is therefore only possible with simple delay filters (in section 2.6 lagrange and truncated sinc was suggested as possible candidates). It is also possible to load precalculated coefficients, but this will require considerable bandwidth if they are not cached.

One other problem when using fraction delay filters is coalescing memory access. Since there is an unlinear reletionship between input address and output address it is not easy to get both reads and writes coalesced. One solution is to let the writes be uncoalesced since there are much less writes than reads. Another is to use textures when reading but without using texture interpolation.

4.1.3 Apodize

Apodization is a window function which is multiplied with the data from each channel to reduce sidelobes. The width of the window function is proportional to the depth to keep the width of the beam constant.

It is the same situation as the delay calculation: either calculate on-the-fly or load pre-calculated values.

One interresting aspect with the apodization is that many samples can be ignored because they are outside the apodization window as illustrated by figure 4.2. If delay calculation and interpolation is skipped for these samples it is possible to save up to 50% of computing time (if half of the samples are outside the window).

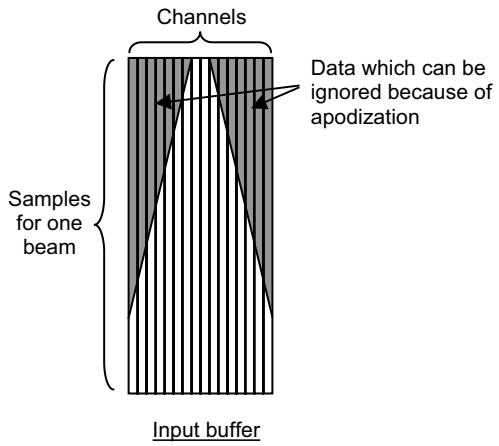

Figure 4.2: Area of input buffer which can be ignored because of apodization.

4.1.4 Sum

The way the sum is implemented depends on how the algorithm is structured with respect to threads and blocks:

- Sum is accumulated sequentially in a loop. E.g. One thread is responsible for one output sample for all channels.
- Sum is acquired by a block of cooperting threads. E.g. Each thread responsible for one input sample, while each block is responsible for one output sample.
- Sum is accumulated sequentially in global memory by threads from different blocks. (This requires some kind of synchronisation mechanism that does not exist in CUDA 1.0.)

4.2 The Implemented Kernels

Section 4.1 showed that there are many ways to implement a beamformer. Only some of the mentioned methods have been used:

- Delay is calculated on-the-fly. Described in section 4.3
- Interpolation is performed by texture filtering(linear interpolation)
- 4x upsamplimg is implemented.
- Apodization is implemented as pre-calculated coefficients.

- Sum is accumulated per thread.

The delay calculation, interpolation and sum is implemented in a single kernel called TEXF[1]. Upsampling is implemented in another kernel called 4xUP.

The TEXF kernel was first implemented in the most basic form then later altered to implement more functions, resulting in an array of different versions:

TEXF : The original.

1/4 TEXF : Altered to accomodate the output from 4xUP.

TEXF MLA : MLA support.

1/4 TEXF MLA : MLA support on the upsampled version.

TEXF APO : Added apodization.

1/4 TEXF MLA APO : Added apodization on the upsampled vesrion.

4.2.1 TEXF

The TEXF kernel calculates one sample of the output data. Texture filtering is used when loading the input data, thus giving linear interpolation for free. The sum is accumulated in a loop, which probably is the easiest and fastest way. The delay calculation is discussed in section 4.3. Apodization is not implemented in this version.

The kernel source code can be found in C.1.1. Here is a pseudo code:

```
TEXF_Kernel
{
  sum = 0;
  For each channel{
    x = CalculateDelay(Idx);
    y = channel;
    sum += textureFetch(x,y);
  }
  outData[Idx] = sum;
}
```

The thread's index is used to calculate the delay so that each consecutive thread calculates the consecutive sample. This ensures that writes are coalesced and that the texture cache is utilized efficiently[2].

Figure 4.3 illustrates the area of the input texture that one block fetches from. At the top and bottom there will be an area of the texture that is cached

[1] It is called TEXF since it employs TEXture Filtering

[2] The texture cache is optimized for 2D locallity i.e. coordinates should not be to spread.

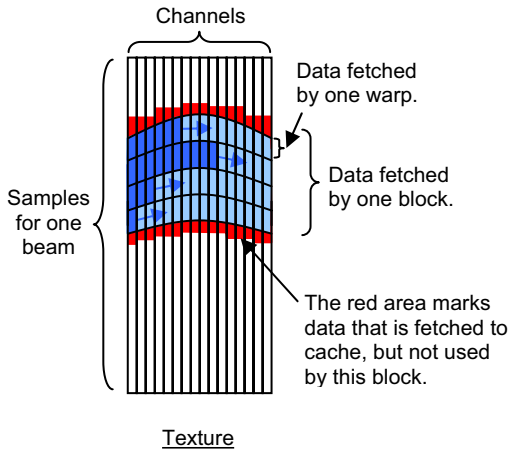

Figure 4.3: Data access pattern for TEXF. The darker blue areas illustrates that the warps may progress at different rates.

but not used by this block. The cache should therefore be best utilized when there are many threads per block[1].

However, the number of threads per beam is the same as the number of samples and must be a factor of the block size. It is therefore not always optimal to use large blocks since the number of samples may vary.

4.2.2 4xUP

Three pre-calculated fractional delay filters are applied to the signal data. The filters delay the signal by 0.25, 0.5 and 0.75 samples. The three delayed signals and the original signal is then interleaved to form a new signal with four times higher sample rate. Figure 4.4 illustrates the principle.

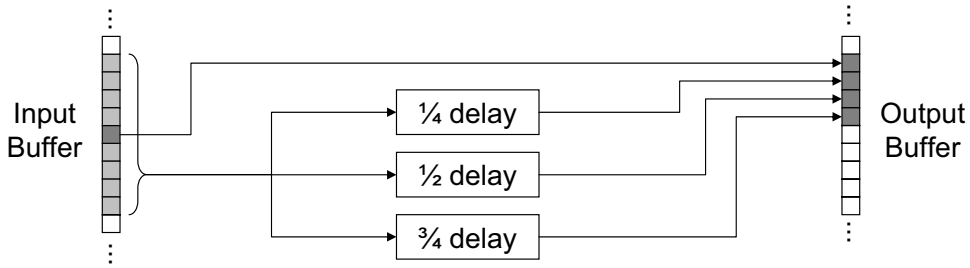

Figure 4.4: 4x upsampling principle diagram.

Kernel source code can be found in C.2.1. Here is a pseudo-code version:

[1]The only way to guarantee that consecutive threads use the same cache(i.e. runs on the same SM) is to put them in the same block.

```
4xUP_Kernel
{
  // cooperative load:
  __shared__ data[Idx] = InBuffer[Idx];

  // 0 delay:
  out[0] = data[Idx];
  // 1/4 delay:
  out[1] = filter(CoEffs[0], &data[Idx-TAPS/2]);
  // 1/2 delay:
  out[2] = filter(CoEffs[1], &data[Idx-TAPS/2]);
  // 3/4 delay:
  out[4] = filter(CoEffs[2], &data[Idx-TAPS/2]);

  // store
  outData[Idx] = out;
}
```

The filtering is performed by convolution of the signal and the filter coefficients:

$$y[k] = \sum_{n=0}^{TAPS} c[n]x[k-n]$$

For every output sample it is necessary to read $TAPS$(length of filter) input samples. To limit the number of times the input data needs to be fetched, the input data is stored in shared memory. For each block it is then only necessary to read one sample per thread in the block + as many samples as there are coefficients in the filter. The coefficients are stored in constant space so that they are cached.

The input data is stored in a texture(linear memory bound). The reason for this is that the input data format is 16-bit integers. The texture unit converts the data to normalized 32-bit floating-point. The output data is stored as 32-bit floating point values.

4.2.3 1/4 TEXF

The TEXF kernel is altered to accommodate the data output from 4xUP. Two alterations were made:

Firstly, the input data now has a 4 times higher sampling rate while output data should have the same sample rate as before. To downsample the data the index is simply multiplied by 4. The sampling frequency used to calculate the delay also needs to be 4 times higher.

Secondly, the input data is now 32-bit floats instead of 16-bit integers. This is fixed by changing the channel descriptor of the texture reference.

The amount of data fetched is now 8 times higher, making the bandwidth a limiting factor.

4.2.4 TEXF MLA

With MLA, beamforming is performed several times on the same input data, but with a different angle each time. The TEXF MLA kernel implements this as a loop around the TEXF kernel code. The loop iterates once for each MLA using a different angle.

A block accesses approximately the same area of the input texture for each MLA. When the first MLA is finished, most of the data for the next MLA would be cached, thus redusing the required bandwidth. Or so was the idea. In perfect hindsight it is easy to see that the 8Kb cache is to small to be of any use. E.g. With 128 channels, 16-bit data and a block size of 256, the amount of data accessed by the first MLA would be $128 \times 256 \times 2 = 64Kb$, which means that the cache would be overwritten about 8 times before the next iteration.

A possible solution to this would be to let the MLAs be executed in parallel. To implement this one could use the second dimension of the thread-block to index the MLA. E.g. A block with dimensions (32, 8, 1) would beamform 32 samples for 8 MLAs. Note that the x dimension should be a multiple of 16 to maintain coalescing of the writes.

4.2.5 1/4 TEXF MLA

The MLA version of 1/4 TEXF is simply a combination of 1/4 TEXF and TEXF MLA. One interresting point is that since 1/4 TEXF requires more bandwidth, it is more important to be able to utilize the cache.

4.2.6 TEXF APO and 1/4 TEXF MLA APO

Apodization is implemented as a pre-calculated coefficient that is multiplied with the delayed data before it is summed. The Apodization data is fetched from a 16-bit texture.

4.3 Delay Calculation

The most important part of the implementation is perhaps the calculation of the delay. All beamforming is based on the same basic idea: delay the signal for each channel according to a focus point then sum all the channels.

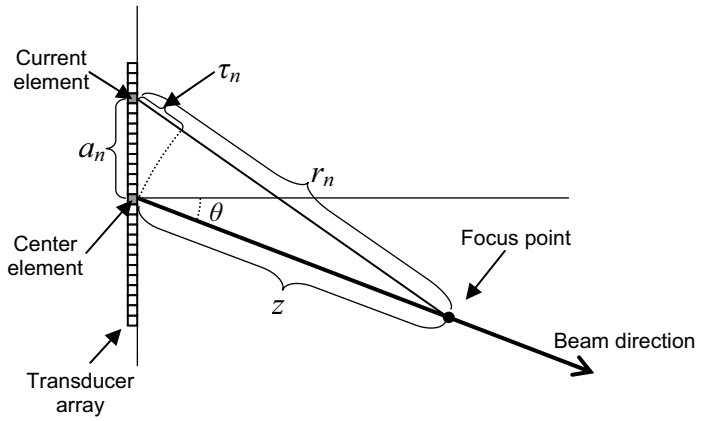

Figure 4.5: Geometry for calculating the delay.

4.3.1 Geometry of the problem

The focus point moves along the beam as the signal is being received and is therefore dependent on time and the angle of beam. The delay is also dependent of the position of the transducer array elements[1]. It is therefore neccessary to recalculate the delay for every element at every sample.

The delay for element n is defined as the time difference of arrival of an echo pulse originating from the focus point to the center element and element n.

Assuming that the speed of the pulses are constant, the delay can be calculated with simple geometry. 4.5 illustrates the geometry of the problem where:

z**:** The distance from the center element to the focus point.

r_n**:** The distance from the focus point to element n.

τ_n**:** The delay.

a_n**:** The distance from the center of the center element to the center of element n.

θ**:** The angle of the beam.

Furthermore we need to define:

t**:** The time since the pulse was transmitted (from the center element).

c_0**:** The speed of the pulses(speed of sound).

[1]It is assumed that the array is a straight line. If the array were to be curved, the calculation of the delay would be slightly different.

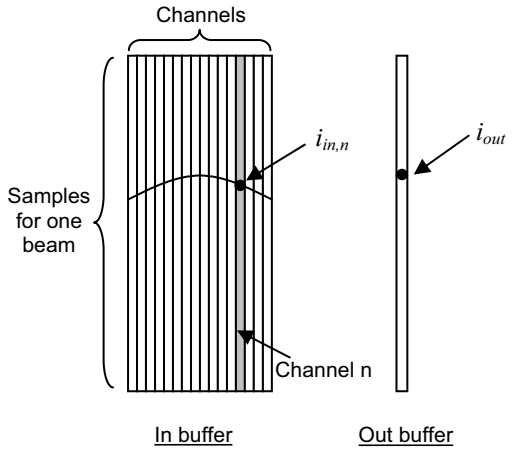

Figure 4.6: The input and output buffer

The relationship between all these are:

$$t = \frac{2z}{c_0} \tag{7}$$

$$t + \tau_n = \frac{z + r_n}{c_0} \tag{8}$$

$$r_n^2 = z^2 + a_n^2 - 2za_n sin\theta \tag{9}$$

4.3.2 The discrete version

In the implementation the signals from each channel will be sampled and stored in a buffer. The time will then just be a discrete index in that buffer(sample number) and the delay will be an offset from that index.

The sum can be formulated like this:

$$data_{out}(i_{out}) = \sum_{n=0}^{N} data_{in}(i_{in,n}, n) \tag{10}$$

Where:

$data_{out}(sample)$**:** The buffer for resulting sums.

$data_{in}(sample, channel)$**:** The sample-buffer for all the channels as a two-dimensional array.

N**:** Number of channels.

i_{out} is the known value (all values from 0 to number of samples in a beam) therefore $i_{in,n}$ has to be defined as a function of i_{out}. The relationships between the indices and t are as follows:

$$t = \frac{i_{out}}{f_s} \tag{11}$$

$$t + \tau = \frac{i_{in,n}}{f_s} \tag{12}$$

By combining equation 7, 8, 9, 11 and 12 we get:

$$i_{in,n} = \frac{i_{out}}{2} + \sqrt{\frac{i_{out}^2}{4} + \frac{a_n^2 f_s^2}{c_0^2} - i_{out}\frac{a_n f_s sin\theta}{c_0}} \tag{13}$$

4.6 illustrates the relationship between $i_{in,n}$ and i_{out}.

4.3.3 Constants (version 1)

It is not necessary to calculate the whole expression 13 every time. Some of the constants can be combined to make new constants like this:

$$C_{1,n} = \frac{a_n^2 f_s^2}{c_0^2} \tag{14}$$

$$C_{2,n} = \frac{a_n f_s sin\theta}{c_0} \tag{15}$$

The new expression will then be:

$$i_{in,n} = \frac{i_{out}}{2} + \sqrt{\frac{i_{out}^2}{4} + C_{1,n} - i_{out}C_{2,n}} \tag{16}$$

The constants can be stored in constant space which is cached and therefore costs nothing to load after the first time.

4.3.4 Constants (version 2)

When beamforming several beams at once, each with a different θ, the constant space easily gets full since $C_{2,n}$ is dependent of both channel number(n) and θ. By seperating them, it is possible to calculate more beams at the same time, which might increase performance. The new constants would be:

$$C_{3,n} = \frac{a_n f_s}{c_0} \tag{17}$$

$$C_4 = sin\theta \tag{18}$$

The new expression will be:

$$i_{in,n} = \frac{i_{out}}{2} + \sqrt{\frac{i_{out}^2}{4} + C_{1,n} - i_{out}C_{3,n}C_4} \tag{19}$$

4.4 Verification of the Kernels

All the kernels have been verified by creating a MATLAB script that creates a reference result which is compared with the results from the CUDA code.

4xUP was tested using white noise as input. Beamforming was tested using simulated input signals. Table 2 lists the maximum deviation between the results from the MATLAB scripts and the CUDA implementation. The deviation is normalized by dividing with highest absolute value.

$$deviation = \frac{max(abs(matlabOutput - cudaOutput))}{max(abs(cudaOutput))}$$

The MATLAB scripts operate on 64-bit floats while the GPU use 32-bit floats for calculations. And when data is transferred to the CUDA version, the data is converted to 16-bits which causes more deviation. The texture unit also does some approximations which contributes to the deviation. Therefore there will always be some deviations.

The deviation results shown in the table are small, which indicates that the CUDA code is working.

	Max deviation
TEXF	3.46×10^{-4}
4xUP	5.7×10^{-5}
1/4 TEXF	1.04×10^{-4}
TEXF MLA	6.09×10^{-4}
1/4 TEXF MLA	1.07×10^{-4}
TEXF APO	6.47×10^{-4}
1/4 TEXF MLA APO	1.77×10^{-4}

Table 2: Verification of kernel code: Maximum normalized deviation from reference.

4.5 Code Performance Analysis

The performance of the implemented kernels is analysed to determine if it possible to utilize the GPU more efficiently.

4.5.1 TEXF

There are several parameters that affect the performance of TEXF. The parameters for TEXF are:

Channels Number of channels

Beams Number of beams to be calculated per kernel launch

Samples Number of samples per beam

BlockSize Number of threads per block.

Figure 4.7 shows a plot of the performance of TEXF with different values for the Samples and Beams parameters. There is a clear pattern in the local maximums(dark blue) which corresponds with white lines. These lines mark where the number of blocks is a multiple of 96. The number of blocks launched per kernel is given by $Beams \times Samples/BlockSize$. The block size in this case is 128, which means that the maximum number of concurrently active blocks per SM is 6. The number of SM's is 16, giving a total of $6 \times 16 = 96$ maximum active blocks. This explains the drop in performance when the number of blocks exceeds 96.

Example: If 97 blocks were to be launched, 96 of the blocks would fully occupy the GPU. These blocks would run in parallel and therefore finish at about the same time. The last block will not start until the one of the first 96 blocks is finished. This block will then eventually be the only one running, and the next kernel launch will have to wait for it to finish.

Note that in future versions of CUDA it might be possible for several kernels to run simultaneously, thus eliminating this problem.

Table 3 shows the performance of TEXF using different block sizes. Performance is measured in cycles per thread per SP, i.e. the number of cycles it takes for one processor to execute one thread. It is calculated like this:

$$Cycles = 128 \times 1.35 \times 10^9 \times \frac{wallTime}{Samples \times Beams \times Repetitions}$$

where 128 is the number of SPs and 1.35×10^9 is clock speed (1.35 GHz). $Repetitions$ is the number of times the kernel is launched, this is to get a better average time[1]. $wallTime$ is the total time it takes to run all repitions.

[1]With 1000 repitions the results can still vary with about 0.1%

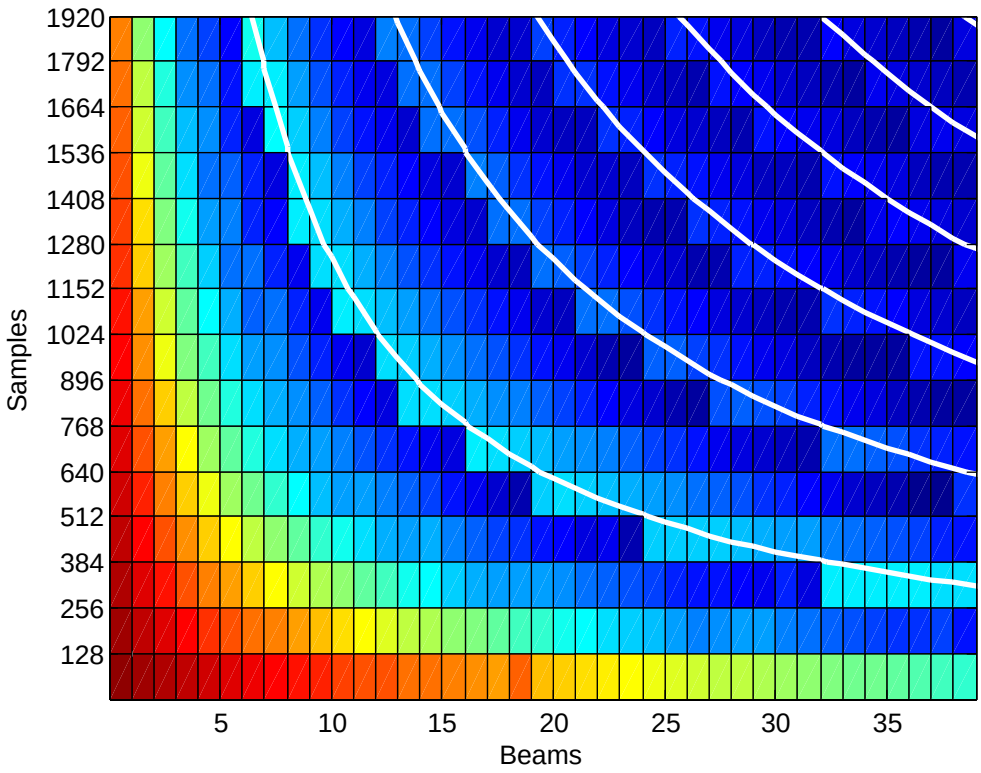

Figure 4.7: TEXF performance. Block size = 128. (Blue is good performance, red is poor performance.)

$Samples$ and $Beams$ are 1536 and 32 respectively which gives full occupancy for all block sizes.

Block size:	96	128	192	256	384
Cycles:	1889.1	1591.8	1571.6	1563.5	1559.8

Table 3: Block size performance test

We can observe that performance increases as the block size increases. This could be because of better use of texture cache for large blocks as discussed in section 4.2.1 but it does not explain the drop in performance when block size is 96. However, it is said that there is a higher risk of register bank conflicts when block size is not a multiple of 64(3), this could be the case here.

By increasing the number of beams and samples the per thread performance can increase even more. Using these parameters:

Test Parameters:	
Channels:	128
Samples:	6144
Beams:	40
Block size:	384
Repetitions:	1000

we got a performance reading of 1545.7 cycles per thread. By analyzing the disassembled kernel code we can see if there is any room for further optimization. The disassembled code can be found in C.1.2 and contains:

- 22 instructions outside of loop
- 13 instructions inside loop

  10 normal instructions (of which 1 is half)

  2 SFU instructions: RSQRT and RCP (of which RCP is half)

  1 texture fetch instruction

The loop iterates once per channel (channels = 128). Assuming that normal instructions take one cycle and SFU instructions take 4 cycles we get

$$cycles = 22 + 10 \times channels + 4 \times 2 \times channels = 22 + 1280 + 1024 = 2326$$

which is almost twice of what was measured. If we assume that the 22 instructions outside the loop are executed in 22 cycles, then the loop is executed in no more than 1545.7 - 22 = 1523.7 cycles, which means that each iteration of the loop is executed in no more than 1523.7/128 = 11.9 cycles. This implies that:

- More than one instruction is issued and executed per cycle. This is discussed in section 3.1.4
- The SFU execute instructions in parallel with the SPs

It is also possible that the GPU clock is a bit higher than the statet 1.35GHz but it would have to more than 10% higher to account for the performance measured in this test, which is highly unlikely.

Anyway, this shows that the kernel is not slowed down by bandwidth or latency of any kind and that the delay calculation is very fast. In other words: it is unlikely that it is possible to optimize further.

4.5.2 4xUP

C.1.2 shows the disassembly of the 4xUP code with filter length = 11. The code consists of:

- 34 instructions independent of filter length(of which 8 are half)
- 66 instructions dependent of filter length(6×length)

  6 MUL instructions

  60 MAD instructions

If one instruction is executed per clock cycle, one thread would execute in 100 cycles.

Test parameters:

Filter length: 11

Channels: 128

Samples: 32768

Block size: 256

Result:

Cycles/thread: 135

The test shows that the kernel runs on 35 more cycles than optimal. In order to discover the reason for this several more tests have been done:

Removing synchronization: Uses 1 less cycle. Indicating that `__syncthreads()` has no impact on performance.

Removing loading from global memory(15 instructions): Uses 19 less cycles. Indicating that neither memory latancy nor bandwidth are limiting factors.

Trying different block sizes(96, 128 and 384): No effect (less than 1 cycle difference).

Trying filter length = 22: Uses 234 cycles. 1.5 added cycles per added MAD instruction.

Trying filter length = 31: Uses 316 cycles. 1.5 added cycles per added MAD instruction.

Trying filter length = 50: Uses 516 cycles. 1.6 added cycles per added MAD instruction.

It seems that each MAD instruction takes on average 1.5 cycles to execute. Assuming that no instructions can execute in a fraction of a cycle, this means that some of the MAD instructions are executed in one cycle and some are delayed (e.g. every second instruction uses two cycles).

A possible source of the delay could be the loading of the operands. Each MAD instruction reads three operands, one operand from constant cache, one operand from shared memory and one from the registers. The program is written so that there are no shared memory bank conflicts and the constants are all read from the same address by all threads in the warp at the same time and therefore should not cause any delays. The only hypothesis left is register bank conflicts, which The Guide states that there are no direct way of controlling.

It should be possible to execute a MAD each cycle, which means that it might be possible increase the performance by about 20%.

4.5.3 1/4 TEXF

Tunning the 1/4 TEXF kernel and the TEXF kernel with the same parameters resulted in:

TEXF : 1544.3 cycles per thread

1/4 TEXF : 11805.1 cycles per thread

Which means that 1/4 TEXF takes 7.6 times longer to complete. This can be explained by the fact that 1/4 TEXF has 4 times more input samples which are 32-bit instead of 16 i.e. 8 times more data to read.

But if we assume that the data is read only once, the bandwidth used would be 27.97 GB/s which is less than half of what should be possible.

The kernel code for 1/4 TEXF is identical to TEXF in the loop. This means that the texture fetch is the only possible reason for the degraded performance.

4.5.4 1/4 TEXF MLA

MLA is implemented by placing an outer loop around the 1/4 TEXF code. When running 1/4 TEXF MLA with just 1 MLA it should do exactly the same as 1/4 TEXF. You would expect that it would run in about the same time, probably a little bit slower, but:

1/4 TEXF : 11805.1 cycles per thread

1/4 TEXF MLA : 6455.6 cycles per thread

It finishes in 55% of the time that 1/4 TEXF does!

The main difference is in how the constants are implemented. 1/4 TEXF MLA uses the constants as discussed in 4.3.4. The new constants are implemented as:

```
__constant__ float C1_gpu[NO_CHANNELS];
__constant__ float C3_gpu[NO_CHANNELS];
__constant__ float C4_gpu[BEAMS][NO_MLA];
```

which with 128 channels, 40 beams and only 1 MLA takes only:

$$4 \times (128 \times 2 + 40 \times 1) = 1184 bytes$$

which easily fits in the 8Kb cache.

The old version was implemented like this:

```
__constant__ float C_gpu[BEAMS][2][NO_CHANNELS];
```

which gives

$$4 \times (40 \times 2 \times 128) = 40960 bytes$$

which does not fit in the cache. But it does not really explain the huge difference in performance.

4.5.5 TEXF MLA

TEXF MLA uses the same constants as 1/4 TEXF MLA which caused significant improvement in performance. Let's see how this affects TEXF:

TEXF (Blocksize=384) : 1545.7 cycles per thread

TEXF MLA(MLA=1, Blocksize=384) : 1533.7 cycles per thread

TEXF MLA(MLA=1, Blocksize=128) : 1503.6 cycles per thread

TEXF MLA is slightly faster than TEXF. It also seems that TEXF MLA runs better with smaller blocks, which indicates that it is not limited by bandwidth. Since the reasen for using large blocks in TEXF was to reduce required bandwidth it must mean that TEXF was bandwidth limited.

By looking at the disassembly (C.4.1) we see that compared to TEXF there are

- 28 intructions outside loop

  6 more than TEXF

- 14 instructions in loop

  one MOV has been changed to MUL

  an additional ADD

Blocksize:	384	128	96
Max active blocks:	1	5	7
Max active threads:	384	640	672
Cycles per thread:	4048.1	2974.5	3398.6
Cycles per MLA:	2024	1487	1699

Table 4: TEXF MLA performance with different blocksizes

There is one more instruction in the loop, yet each iteration of the loop runs in only 11.5 cycles:

$$\frac{1503.6 - 28}{128} = 11.5$$

Note that the compiler has removed the outer loop since MLA is 1. When MLA is more than 1, the code is altered. One alteration is that two more registers are used:

MLA = 1 : Registers used = 10

MLA > 1 : Registers used = 12

With 12 registers per thread it is only possible to have $8192/12 = 682$ active threads, which makes a blocksize of 384 a poor choice as shown in table 4. Note that even though there are more active threads with a blocksize of 96 there is better performance with a blocksize of 128. This could be because of register bank conflicts.

4.5.6 1/4 TEXF MLA APO

When adding apodization the number of required registers increased to 16. This means that only 512 threads can be active per SM. The optimal blocksize was found to be 64.

Apodization also increases the required bandwidth.

4.6 Performance Summary

Table 5 shows the performance of the kernels measured in number of samples processed per second. This means that to achieve real-time performance this number must be greater than the number of channels$\times$sampling rate of the transducer.

Example: With 128 channels and 40MHz sampling rate, 1/4 TEXF MLA APO (16 MLA) would run in:

$$\frac{0.20 \times 10^9}{128 \times 40 \times 10^6} = 0.04 \text{ x realtime}$$

	Performance	(per MLA)
TEXF	14.32×10^9	-
4xUP (TAPS=7)	3.43×10^9	-
4xUP (TAPS=11)	2.55×10^9	-
4xUP (TAPS=15)	2.02×10^9	-
1/4 TEXF	1.87×10^9	-
TEXF MLA (1 MLA)	14.71×10^9	14.71×10^9
TEXF MLA (4 MLA)	2.83×10^9	11.32×10^9
TEXF MLA (16 MLA)	0.73×10^9	11.67×10^9
1/4 TEXF MLA (1 MLA)	3.43×10^9	3.43×10^9
1/4 TEXF MLA (4 MLA)	0.95×10^9	3.81×10^9
1/4 TEXF MLA (16 MLA)	0.24×10^9	3.92×10^9
TEXF APO	8.32×10^9	-
1/4 TEXF MLA APO (1 MLA)	2.96×10^9	2.96×10^9
1/4 TEXF MLA APO (4 MLA)	0.77×10^9	3.10×10^9
1/4 TEXF MLA APO (16 MLA)	0.20×10^9	3.15×10^9

Table 5: Performance of kernels. (samples/sec)

Which means that you would need 25 GPUs in order to run in real time. But it is also necessary to run 4xUP (here with 11 taps):

$$\frac{2.55 \times 10^9 \times 10^9}{128 \times 40 \times 10^6} = 0.50 \text{ x realtime}$$

Which would require two additional GPUs in order to run in real time.

4.7 Possible Optimizations

- Utilize cache better when upsampling and MLA is used. This was discussed in 4.2.4. This could reduce the required bandwidth by a factor of number of MLA. Possible improvement: about 300% added performance.
- Convert 4xUP output to 16-bit. Possible improvement: up to 100% added performance.
- Find out how to make 4xUP perform 1 MAD operation per cycle as discussed in 4.5.2. Possible improvement: about 20% added performance on 4xUP.

If the cache where utilized better when using upsampling and MLA it is possible that when there are enough MLAs it will no longer be limited by bandwidth but by instruction throughput. This means that 1/4 TEXF MLA could run as fast TEXF MLA, which is about 3 times faster.

5 Conclusion

Even though beamforming is easily adapted for computation on a GPU, we have showed that there are many ways to accomplish this and that finding an optimal solution requires good knowledge of the architecture and much tweaking.

A fully functional beamforming algorithm has been implemented that performs:

- Delay and sum on an arbitrary number of channels, where the delay is calculated from a given angle and transducer element positions[1].
- Linear interpolation for fractional delays.
- Apodization using pre-calculated coefficients.

But it has been shown that the implementation is not optimal so there is still some work to do. Section 4.7 gives some suggestions as to where to start.

Linear interpolation gives poor results at low sampling rates. An upsampling filter has therefore been implemented to counter this. The upsampling filter uses three 11-tap pre-calculated FIR fraction delay filters to produce a four times higher sample rate.

To compare the performance of the implementations we use an example case:

- 128 channels
- 40 MHz sampling frequency
- 5120 samples per beam i.e. depth $\approx$ 10cm.

If we perform beamforming on all the incoming data, i.e no pause between pulses, we can measure the speed as:

$$Speed = \frac{\text{computed data rate}}{\text{incoming data rate}}$$

If the speed is more than 1 then the GPU can perform the beamforming in real-time. If the speed is less than one it is necessary to use several GPUs to achieve real-time performance.

Table 6 shows the speed and GPUs needed to perform real-time beamforming using the implemented algorithms.

Note that:

[1] Array elements must be positioned in a straight line

	Implemented		Optimized	
	Speed	GPUs needed	Speed	GPUs needed
No upsampling:	1.62	1	1.62	1
Upsampled:	0.27	4	0.41	3
Upsampled and 2xMLA:	0.18	6	0.32	4
Upsampled and 4xMLA:	0.12	9	0.24	5
Upsampled and 8xMLA:	0.07	16	0.15	7
Upsampled and 16xMLA:	0.04	28	0.09	12

Table 6: Number of GPUs needed to perform beamforming in real-time.

- The optimized performance is hypothetical since it has not been implemented yet[1].
- The time it takes to get the data in and out of the GPU card is not accounted for.
- The current version of CUDA does not support writing to textures from the kernel. It is therefore necessary to copy the data output from the upsampler into the texture used by the beamforming kernel. This introduces an extra delay which is not included in the speed calculation.

From the table it seems that you would need a lot of GPUs to manage MLA beamforming. But keep in mind that if the current rate of development of GPU performance continues, it is not unlikely that we will see GPUs with 4 times the performance within the next couple of years.

[1]It is assumed that if the optimization suggested in section 4.7 are implemented, that when 4 or more MLA is used, the performance is as good as without upsampling divided by number of MLA.

6 Bibliography

[1] http://en.wikipedia.org/wiki/medical_ultrasonography. Internet. Downloaded 29/May/2008.

[2] http://pc.watch.impress.co.jp/docs/2006/1109/kaigai316.htm. Internet. Downloaded 5/May/2008.

[3] NVidia. *NVIDIA CUDA (Compute Unified Device Architecture): Programming Guide*, 1.1 edition, 2007.

[4] S. S. Stone S. S. Baghsorkhi D. B. Kirk S. Ryoo, C. I. Rodrigues and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[5] Matti Karjalainen Timo I. Laakso, Vesa Välimäki and Unto K. Laine. Splitting the unit delay. *IEEE Signal Processing Magazine*, 1996.

APPENDIX

A Tesla C870 physical requirements

A.1 Size matters

The Tesla C870 card is full length and dual slot, in other words, huge. Therefore it does not fit into any PC case, mostly because of the length. Many modern cases have harddrive bays in the front and leaves too little room for a full length card.

Figure A.1: Dimensions of the C870 board.

A.2 Cooling

The card produces a lot of heat and for that reason has a large cooling fan that sucks in air which is expelled at the back of the PC.

An old PC case was used in the first attempt to install the card. It was the only easily available case which had room for the full length card. Unfortunately it was not designed for air circulation and had no built in fans or ventilation holes. To allow air to get into the case it had to be left open.

But, even with the cover off, the card still did not get enough air circulation! When running heavy simulations[1] the temperature[2] would rise and the simulation would eventually hang at about $75\,^{\circ}\mathrm{C}$.

A new case was acquired that had better air circulation. It is a good idea to have more case fans blowing in than sucking out. This creates a higher pressure on the inside which forces the more air to exit through the fan on the Tesla GPU.

[1] The N-Body application from the CUDA SDK
[2] As reported by the NVIDIA Monitor application

B Overview of Attached Files

There was no time to clean up the files so it really is a mess, but this overview should help you find the important files.

Note that the beamforming kernels exists in two versions: one that runs from matlab and one that runs as a normal executable.

/CUDA beamforming/ Contains all CUDA files and related MATLAB scripts

/CUDA beamforming/CUDA beamforming project.sln Visual Studio Solution that contains all the non MATLAB beamforming kernels

/CUDA beamforming/MATLAB Contains all the MATLAB beamforming kernels and other related MATLAB scripts

[TEXF.cu] TEXF kernel

[TEXF_MLA.cu] TEXF MLA kernel

[TEXF_APO.cu] TEXF APO kernel

[quarterTEXF.cu] 1/4 TEXF kernel

[quarterTEXF_MLA.cu] 1/4 TEXF MLA kernel

[quarterTEXF_MLA_APO.cu] 1/4 TEXF MLA APOkernel

[Verify???.m] Scripts to verify the kernels

/CUDA beamforming/MATLAB/README.TXT Instructions on how to compile CUDA MEX(MATLAB executable) files.

/CUDA beamforming/MATLAB/Generate data/ Scripts to generate simulated received ultrasound data.

[TestSweep] Generates data. Performs beamforming and displays resulting image.

/CUDA beamforming/MATLAB/ReceiveBeamAmp/ Script that uses the beamforming kernels to generate the receive beam shape.

/MATLAB/ Contains some matlab scripts used for illustrations

/MATLAB/interpolator test/ Scripts to generate delay filters and the filter comparison script refered to in section 2.6.4

/MATLAB/illustrations/ Some illustrations.. example beam_amplitude_plot.m plots the transmit beam shape.

C Source Code Snippets

C.1 TEXF

C.1.1 Kernel Source

Listing 2: TEXF Kernel Source

```
1
2
3  texture<float,2,cudaReadModeNormalizedFloat> tex;
4
5  __constant__ float C_gpu[MAX_BEAMS][2][MAX_CHANNELS];
6
7  __global__ void delay_and_sum_kernel( float *out_data, int channels, int samples )
8  {
9      int idx = blockIdx.x*BLOCK_SIZE+threadIdx.x;
10     if(idx < samples){
11         float i = idx;
12         float out = 0.0f;
13         int beam = blockIdx.y;
14         float y = 0.5f + channels*beam;
15         float y_max = channels*(beam+1);
16         int n = 0;
17         do{
18             out += tex2D(tex, 0.5f + i/2 + sqrt(i*i/4 + C_gpu[beam][0][n] - i*C_gpu[beam][1][n]),y);
19             y += 1.0f;
20             n += 1;
21         } while( y < y_max );
22         idx += beam*samples;
23         out_data[idx] = out;
24     }
25 }
```

C.1.2 Disassembly

Listing 3: TEXF Disassembly

```
1  000000: 41002c05 00000013 mul24.lo.u32.u16.u16 $r1, s[0x000c], 0x0100
2  000008: a0000001 04000780 cvt.u32.u16 $r0, $r0.lo
3  000010: 20000011 04004780 add.u32 $r4, $r0, $r1
4  000018: 000a4e05 c0200780 movsh.b32 $ofs1, s[0x001c], 0x0000000a (shared memory operand type mismatch)
5  000020: 10004e01 0023c780 mov.b16 $r0.lo, s[0x000e]
6  000028: 10008009 00000003 mov.b32 $r2, 0x00000000
7  000030: 40008101 0000000b mul24.lo.s32.s16.s16 $r64, $r0.lo, 0x0080
8  000038: 1000801d 03f00003 mov.b32 $r7, 0x3f000000
9  000040: a0000005 44014780 cvt.rn.f32.s32 $r1, $r0
10 000048: 20008001 0000000b add.b32 $r0, $r0, 0x00000080
11 000050: b000020d 03f00003 add.rn.f32 $r3, $r1, 0x3f000000
12 000058: a0000819 44014780 cvt.rn.f32.s32 $r6, $r4
13 000060: a0000015 44014780 cvt.rn.f32.s32 $r5, $r0
14 000068: c0060c01 00000780 mul.rn.f32 $r0, $r6, $r6
15 000070: e0000c1d 03f00003 mad.rn.f32 $r7, $r6, 0x3f000000 (No operand 4 in this instruction)
16 000078: c0000021 03e80003 mul.rn.f32 $r8, $r0, 0x3e800000
17 000080: b5001001 00000780 label0: add.rn.f32 $r0, $r8, c0[$ofs1+0x0000]
18 000088: 14010005 2400c780 mov.b32 $r1, c0[$ofs1+0x0200]
19 000090: e0010c01 04000780 mad.rn.f32 $r0, -$r6, $r1, $r0
20 000098: 90000001 40000780 rsqrt.f32 $r0, $r0
21 0000a0: 90000000          rcp.half.f32 $r0, $r0
22 0000a4: 10008604          mov.half.b32 $r1, $r3
23 0000a8: b0000e01 00000780 add.rn.f32 $r0, $r7, $r0
24 0000b0: f2400001 00000784 tex.2d.b32.f32 {$r0,_,_,_}, $tex0, {$r0,$r1}
25 0000b8: b000060d 03f80003 add.rn.f32 $r3, $r3, 0x3f800000
26 0000c0: b0030bfd 600107c8 set.gt.u16.f32.f32 $p0|$o127, $r5, $r3
27 0000c8: b0000009 00008780 add.rn.f32 $r2, $r0, $r2
28 0000d0: d4000805 20000780 add.b32 $ofs1, $ofs1, 0x00000004
29 0000d8: 10010003 00000280 @$p0.ne bra.label label0
30 0000e0: 10004e01 0023c780 mov.b16 $r0.lo, s[0x000e]
31 0000e8: 40008101 00000083 mul24.lo.s32.s16.s16 $r64, $r0.lo, 0x0800
32 0000f0: 20000801 04000780 add.u32 $r0, $r4, $r0
33 0000f8: 30020001 c4100780 shl.u32 $r0, $r0, 0x00000002
34 000100: 2000c801 04200780 add.u32 $r0, s[0x0010], $r0
35 000108: d00e0009 a0c00781 mov.end.u32 g[$r0], $r2
```

C.2 4xUP

C.2.1 Kernel Source

Listing 4: 4xUP Kernel Source

```
1  __constant__ float c_coeffs[3][TAPS];
2
3  texture<float2, 1, cudaReadModeNormalizedFloat> tex_short2;
4
5  __global__ void upsample4x_kernel( float4 *out_data, short channel_pitch_in, short channel_pitch_out )
6  {
7      extern __shared__ float shared[];
8
9      const unsigned int ti = threadIdx.x;
10     const unsigned int i_out = blockIdx.y*channel_pitch_out + blockIdx.x*BLOCK_SIZE*2 + ti;
11     const unsigned int i_in = blockIdx.y*channel_pitch_in + blockIdx.x*BLOCK_SIZE + ti;
12     float4 out;
13
14     // load data into shared memory
15     float2 in = tex1Dfetch(tex_short2, i_in);
16     shared[ti*2]   = in.x;
17     shared[ti*2+1] = in.y;
18
19     if(ti < TAPS-1){
20         in = tex1Dfetch(tex_short2, BLOCK_SIZE + i_in);
21         shared[BLOCK_SIZE*2 + ti*2]   = in.x;
22         shared[BLOCK_SIZE*2 + ti*2+1] = in.y;
23     }
24
25     __syncthreads();
26
27
28     out.x = shared[ti + TAPS/2];    // D = 0
29     out.y = 0.0f; out.z = 0.0f; out.w = 0.0f;
30 #pragma unroll
31     for( int n = 0; n<TAPS; n++)
32         out.y += c_coeffs[0][n]*shared[ti+TAPS-1-n]; // D = -0.25
33 #pragma unroll
34     for( int n = 0; n<TAPS; n++)
35         out.z += c_coeffs[1][n]*shared[ti+TAPS-1-n]; // D = -0.5
36 #pragma unroll
37     for( int n = 0; n<TAPS; n++)
38         out.w += c_coeffs[2][n]*shared[ti+TAPS-1-n]; // D = -0.75
39     out_data[i_out] = out;
40
41     out.x = shared[BLOCK_SIZE + ti + TAPS/2];   // D = 0
42     out.y = 0.0f; out.z = 0.0f; out.w = 0.0f;
43 #pragma unroll
44     for( int n = 0; n<TAPS; n++)
45         out.y += c_coeffs[0][n]*shared[BLOCK_SIZE + ti+TAPS-1-n]; // D = -0.25
46 #pragma unroll
47     for( int n = 0; n<TAPS; n++)
48         out.z += c_coeffs[1][n]*shared[BLOCK_SIZE + ti+TAPS-1-n]; // D = -0.5
49 #pragma unroll
50     for( int n = 0; n<TAPS; n++)
51         out.w += c_coeffs[2][n]*shared[BLOCK_SIZE + ti+TAPS-1-n]; // D = -0.75
52
53     out_data[BLOCK_SIZE + i_out] = out;
54 }
```

C.2.2 Disassembly

Listing 5: 4xUP Disassembly

```
1  000000: 11005408           mov.half.b16 $r1.lo, s[0x0014]
2  000004: 11002e04           mov.half.b16 $r0.hi, s[0x000e]
3  000008: 41002c09 00000013 mul24.lo.u32.u16.u16 $r2, s[0x000c], 0x0100
4  000010: 40010405 00000780 mul24.lo.u32.u16.u16 $r1, $r1.lo, $r0.hi
5  000018: 20000205 04008780 add.u32 $r1, $r1, $r2
6  000020: a0000009 04000780 cvt.u32.u16 $r2, $r0.lo
7  000028: 2001840c           add.half.b32 $r3, $r2, $r1
8  00002c: 10008600           mov.half.b32 $r0, $r3
9  000030: 00030405 c0000780 movsh.b32 $ofs1, $r2, 0x00000003
10 000038: f7000001 00000784 tex.1d.s32 {$r0,$r1,_,_}, $tex0, {$r0}
11 000040: 04001001 e4200780 mov.b32 s[$ofs1+0x0020], $r0
12 000048: 04001201 e4204780 mov.b32 s[$ofs1+0x0024], $r1
13 000050: 308005fd 644107c8 set.gt.u32 $p0|$o127, $r2, c1[0x0000]
```

```
000058: 21000601 04404500 @$p0.equ add.u32 $r0, $r3, c1[0x0004]
000060: 00030405 c0000500 @$p0.equ movsh.b32 $ofs1, $r2, 0x00000003
000068: f7000001 00000504 @$p0.equ tex.1d.s32 {$r0,$r1,_,_}, $tex0, {$r0}
000070: 04041001 e4200500 @$p0.equ mov.b32 s[$ofs1+0x0820], $r0
000078: 04041201 e4204500 @$p0.equ mov.b32 s[$ofs1+0x0824], $r1
000080: 861ffe03 00000000 bar.sync.u32 0x00000000
000088: 11005600          mov.half.b16 $r0.lo, s[0x0016]
00008c: 11002e04          mov.half.b16 $r0.hi, s[0x000e]
000090: 41002c05 00000023 mul24.lo.u32.u16.u16 $r1, s[0x000c], 0x0200
000098: 40010000          mul24.half.lo.u32.u16.u16 $r0, $r0.lo, $r0.hi
00009c: 20018000          add.half.b32 $r0, $r0, $r1
0000a0: 00020405 c0000780 movsh.b32 $ofs1, $r2, 0x00000002
0000a8: 20000011 04008780 add.u32 $r4, $r0, $r2
0000b0: c480e405 00200780 mul.rn.f32 $r1, s[$ofs1+0x0048], c0[$ofs1+0x0000]
0000b8: c48be409 00200780 mul.rn.f32 $r2, s[$ofs1+0x0048], c0[$ofs1+0x002c]
0000c0: c496e40d 00200780 mul.rn.f32 $r3, s[$ofs1+0x0048], c0[$ofs1+0x0058]
0000c8: 1400da01 0423c780 mov.b32 $r0, s[$ofs1+0x0034]
0000d0: e481e205 00204780 mad.rn.f32 $r1, s[$ofs1+0x0044], c0[$ofs1+0x0004], $r1
0000d8: e48ce209 00208780 mad.rn.f32 $r2, s[$ofs1+0x0044], c0[$ofs1+0x0030], $r2
0000e0: e497e20d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0044], c0[$ofs1+0x005c], $r3
0000e8: e482e005 00204780 mad.rn.f32 $r1, s[$ofs1+0x0040], c0[$ofs1+0x0008], $r1
0000f0: e48de009 00208780 mad.rn.f32 $r2, s[$ofs1+0x0040], c0[$ofs1+0x0034], $r2
0000f8: e498e00d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0040], c0[$ofs1+0x0060], $r3
000100: 30040811 c4100780 shl.u32 $r4, $r4, 0x00000004
000108: e483de05 00204780 mad.rn.f32 $r1, s[$ofs1+0x003c], c0[$ofs1+0x000c], $r1
000110: e48ede09 00208780 mad.rn.f32 $r2, s[$ofs1+0x003c], c0[$ofs1+0x0038], $r2
000118: e499de0d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x003c], c0[$ofs1+0x0064], $r3
000120: 2000c811 04210780 add.u32 $r4, s[0x0010], $r4
000128: e484dc05 00204780 mad.rn.f32 $r1, s[$ofs1+0x0038], c0[$ofs1+0x0010], $r1
000130: e48fdc09 00208780 mad.rn.f32 $r2, s[$ofs1+0x0038], c0[$ofs1+0x003c], $r2
000138: e49adc0d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0038], c0[$ofs1+0x0068], $r3
000140: e485da05 00204780 mad.rn.f32 $r1, s[$ofs1+0x0034], c0[$ofs1+0x0014], $r1
000148: e490da09 00208780 mad.rn.f32 $r2, s[$ofs1+0x0034], c0[$ofs1+0x0040], $r2
000150: e49bda0d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0034], c0[$ofs1+0x006c], $r3
000158: e486d805 00204780 mad.rn.f32 $r1, s[$ofs1+0x0030], c0[$ofs1+0x0018], $r1
000160: e491d809 00208780 mad.rn.f32 $r2, s[$ofs1+0x0030], c0[$ofs1+0x0044], $r2
000168: e49cd80d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0030], c0[$ofs1+0x0070], $r3
000170: e487d605 00204780 mad.rn.f32 $r1, s[$ofs1+0x002c], c0[$ofs1+0x001c], $r1
000178: e492d609 00208780 mad.rn.f32 $r2, s[$ofs1+0x002c], c0[$ofs1+0x0048], $r2
000180: e49dd60d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x002c], c0[$ofs1+0x0074], $r3
000188: e488d405 00204780 mad.rn.f32 $r1, s[$ofs1+0x0028], c0[$ofs1+0x0020], $r1
000190: e493d409 00208780 mad.rn.f32 $r2, s[$ofs1+0x0028], c0[$ofs1+0x004c], $r2
000198: e49ed40d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0028], c0[$ofs1+0x0078], $r3
0001a0: e489d205 00204780 mad.rn.f32 $r1, s[$ofs1+0x0024], c0[$ofs1+0x0024], $r1
0001a8: e494d209 00208780 mad.rn.f32 $r2, s[$ofs1+0x0024], c0[$ofs1+0x0050], $r2
0001b0: e49fd20d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0024], c0[$ofs1+0x007c], $r3
0001b8: e48ad005 00204780 mad.rn.f32 $r1, s[$ofs1+0x0020], c0[$ofs1+0x0028], $r1
0001c0: e495d009 00208780 mad.rn.f32 $r2, s[$ofs1+0x0020], c0[$ofs1+0x0054], $r2
0001c8: e4a0d00d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0020], c0[$ofs1+0x0080], $r3
0001d0: d4084005 20000780 add.b32 $ofs1, $ofs1, 0x00000420
0001d8: d00e0801 a0a00780 mov.b128 g[$r4], $r0
0001e0: c480d405 00200780 mul.rn.f32 $r1, s[$ofs1+0x0028], c0[$ofs1+0x0000]
0001e8: c48bd409 00200780 mul.rn.f32 $r2, s[$ofs1+0x0028], c0[$ofs1+0x002c]
0001f0: c496d40d 00200780 mul.rn.f32 $r3, s[$ofs1+0x0028], c0[$ofs1+0x0058]
0001f8: 1400ca01 0423c780 mov.b32 $r0, s[$ofs1+0x0014]
000200: e481d205 00204780 mad.rn.f32 $r1, s[$ofs1+0x0024], c0[$ofs1+0x0004], $r1
000208: e48cd209 00208780 mad.rn.f32 $r2, s[$ofs1+0x0024], c0[$ofs1+0x0030], $r2
000210: e497d20d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0024], c0[$ofs1+0x005c], $r3
000218: e482d005 00204780 mad.rn.f32 $r1, s[$ofs1+0x0020], c0[$ofs1+0x0008], $r1
000220: e48dd009 00208780 mad.rn.f32 $r2, s[$ofs1+0x0020], c0[$ofs1+0x0034], $r2
000228: e498d00d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0020], c0[$ofs1+0x0060], $r3
000230: e483ce05 00204780 mad.rn.f32 $r1, s[$ofs1+0x001c], c0[$ofs1+0x000c], $r1
000238: e48ece09 00208780 mad.rn.f32 $r2, s[$ofs1+0x001c], c0[$ofs1+0x0038], $r2
000240: e499ce0d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x001c], c0[$ofs1+0x0064], $r3
000248: 20008811 00000103 add.b32 $r4, $r4, 0x00001000
000250: e484cc05 00204780 mad.rn.f32 $r1, s[$ofs1+0x0018], c0[$ofs1+0x0010], $r1
000258: e48fcc09 00208780 mad.rn.f32 $r2, s[$ofs1+0x0018], c0[$ofs1+0x003c], $r2
000260: e49acc0d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0018], c0[$ofs1+0x0068], $r3
000268: e485ca05 00204780 mad.rn.f32 $r1, s[$ofs1+0x0014], c0[$ofs1+0x0014], $r1
000270: e490ca09 00208780 mad.rn.f32 $r2, s[$ofs1+0x0014], c0[$ofs1+0x0040], $r2
000278: e49bca0d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0014], c0[$ofs1+0x006c], $r3
000280: e486c805 00204780 mad.rn.f32 $r1, s[$ofs1+0x0010], c0[$ofs1+0x0018], $r1
000288: e491c809 00208780 mad.rn.f32 $r2, s[$ofs1+0x0010], c0[$ofs1+0x0044], $r2
000290: e49cc80d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0010], c0[$ofs1+0x0070], $r3
000298: e487c605 00204780 mad.rn.f32 $r1, s[$ofs1+0x000c], c0[$ofs1+0x001c], $r1
0002a0: e492c609 00208780 mad.rn.f32 $r2, s[$ofs1+0x000c], c0[$ofs1+0x0048], $r2
0002a8: e49dc60d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x000c], c0[$ofs1+0x0074], $r3
0002b0: e488c405 00204780 mad.rn.f32 $r1, s[$ofs1+0x0008], c0[$ofs1+0x0020], $r1
0002b8: e493c409 00208780 mad.rn.f32 $r2, s[$ofs1+0x0008], c0[$ofs1+0x004c], $r2
0002c0: e49ec40d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0008], c0[$ofs1+0x0078], $r3
0002c8: e489c205 00204780 mad.rn.f32 $r1, s[$ofs1+0x0004], c0[$ofs1+0x0024], $r1
0002d0: e494c209 00208780 mad.rn.f32 $r2, s[$ofs1+0x0004], c0[$ofs1+0x0050], $r2
0002d8: e49fc20d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0004], c0[$ofs1+0x007c], $r3
0002e0: e48ac005 00204780 mad.rn.f32 $r1, s[$ofs1+0x0000], c0[$ofs1+0x0028], $r1
0002e8: e495c009 00208780 mad.rn.f32 $r2, s[$ofs1+0x0000], c0[$ofs1+0x0054], $r2
0002f0: e4a0c00d 0020c780 mad.rn.f32 $r3, s[$ofs1+0x0000], c0[$ofs1+0x0080], $r3
0002f8: d00e0801 a0a00781 mov.end.b128 g[$r4], $r0
```

C.3 1/4 TEXF MLA

C.3.1 Kernel Source

Listing 6: 1/4 TEXF MLA Kernel Source

```
1  #define DOWNSAMPLE 4
2
3  texture<float,2,cudaReadModeElementType> tex;
4
5  __constant__ float C1_gpu[NO_CHANNELS];
6  __constant__ float C3_gpu[NO_CHANNELS];
7  __constant__ float C4_gpu[BEAMS][NO_MLA];
8
9  __global__ void delay_and_sum_kernel( float *out_data )
10 {
11     int idx = blockIdx.x*BLOCK_SIZE+threadIdx.x;
12     float i = DOWNSAMPLE*idx;
13     int beam = blockIdx.y;
14     idx += beam*NO_MLA*SAMPLES;
15     for(int mla = 0; mla < NO_MLA; mla++)
16     {
17         float out = 0.0f;
18         float y = 0.5f + NO_CHANNELS*beam;
19         float y_max = NO_CHANNELS*(beam+1);
20         int n = 0;
21         do{
22             out += tex2D(tex, 0.5f + i/2 + sqrt(i*i/4 + C1_gpu[n] - i*C3_gpu[n]*C4_gpu[beam][mla]),y);
23             y += 1.0f;
24             n += 1;
25         } while( y < y_max );
26         out_data[idx] = out;
27         idx += SAMPLES;
28     }
29 }
```

C.4 TEXF MLA

C.4.1 Disassembly

Listing 7: TEXF MLA Disassembly

```
1  000000: 41002c09 0000000b mul24.lo.u32.u16.u16 $r2, s[0x000c], 0x0080
2  000008: a0000005 04000780 cvt.u32.u16 $r1, $r0.lo
3  000010: 10008001 00000003 mov.b32 $r0, 0x00000000
4  000018: 20000211 04008780 add.u32 $r4, $r1, $r2
5  000020: 00000005 c0000780 movsh.b32 $ofs1, $r0, 0x00000000
6  000028: 10008001 00000023 mov.b32 $r0, 0x00000200
7  000030: a0004e05 04200780 cvt.u32.u16 $r1, s[0x000e]
8  000038: 00000009 c0000780 movsh.b32 $ofs2, $r0, 0x00000000
9  000040: 0002020d c0000780 movsh.b32 $ofs3, $r1, 0x00000002
10 000048: 10008009 00000003 mov.b32 $r2, 0x00000000
11 000050: 10004e01 0023c780 mov.b16 $r0.lo, s[0x000e]
12 000058: 10008021 03f00003 mov.b32 $r8, 0x3f000000
13 000060: 40008101 0000000b mul24.lo.s32.s16.s16 $r64, $r0.lo, 0x0080
14 000068: 1c02001d 2400c780 mov.b32 $r7, c0[$ofs3+0x0400]
15 000070: a0000005 44014780 cvt.rn.f32.s32 $r1, $r0
16 000078: 20008001 0000000b add.b32 $r0, $r0, 0x00000080
17 000080: b000020d 03f00003 add.rn.f32 $r3, $r1, 0x3f000000
18 000088: a0000819 44014780 cvt.rn.f32.s32 $r6, $r4
19 000090: a0000015 44014780 cvt.rn.f32.s32 $r5, $r0
20 000098: c0060c01 00000780 mul.rn.f32 $r0, $r6, $r6
21 0000a0: e0000c21 03f00003 mad.rn.f32 $r8, $r6, 0x3f000000 (No operand 4 in this instruction)
22 0000a8: c0000025 03e80003 mul.rn.f32 $r9, $r0, 0x3e800000
23 0000b0: b4801200          label0: add.half.rn.f32 $r0, $r9, c0[$ofs1+0x0000]
24 0000b4: c8800c04          mul.half.rn.f32 $r1, $r6, c0[$ofs2+0x0000]
25 0000b8: e0010e01 04000780 mad.rn.f32 $r0, -$r7, $r1, $r0
26 0000c0: 90000001 40000780 rsqrt.f32 $r0, $r0
27 0000c8: 90000000          rcp.half.f32 $r0, $r0
28 0000cc: 10008604          mov.half.b32 $r1, $r3
29 0000d0: b0001001 00000780 add.rn.f32 $r0, $r8, $r0
30 0000d8: f2400001 00000784 tex.2d.b32.f32 {$r0,_,_,_}, $tex0, {$r0,$r1}
31 0000e0: b000060d 03f80003 add.rn.f32 $r3, $r3, 0x3f800000
32 0000e8: b0000009 00008780 add.rn.f32 $r2, $r0, $r2
33 0000f0: b0030bfd 600107c8 set.gt.u16.f32.f32 $p0|$o127, $r5, $r3
34 0000f8: d8000809 20000780 add.b32 $ofs2, $ofs2, 0x00000004
```

```
000100: d4000805 20000780 add.b32 $ofs1, $ofs1, 0x00000004
000108: 10016003 00000280 @$p0.ne bra.label label0
000110: 10004e01 0023c780 mov.b16 $r0.lo, s[0x000e]
000118: 40008101 00000183 mul24.lo.s32.s16.s16 $r64, $r0.lo, 0x1800
000120: 20000801 04000780 add.u32 $r0, $r4, $r0
000128: 30020001 c4100780 shl.u32 $r0, $r0, 0x00000002
000130: 2000c801 04200780 add.u32 $r0, s[0x0010], $r0
000138: d00e0009 a0c00781 mov.end.u32 g[$r0], $r2
```