

Department of Computer and Information Science

Final thesis

**Implementation of a real-time Fast
Fourier Transform on a Graphics
Processing Unit with data streamed
from a high-performance digitizer**

by

Jonas Henriksson

LIU-IDA/LITH-EX-A-14/026-SE

January 2015



Linköping University

Final thesis

**Implementation of a real-time Fast
Fourier Transform on a Graphics
Processing Unit with data streamed
from a high-performance digitizer**

by

Jonas Henriksson

LIU-IDA/LITH-EX-A-14/026-SE

January 2015

Supervisors: Usman Dastgeer, Anders Kagerin, Martin
Olsson

Examiner: Christoph Kessler

Abstract

In this thesis we evaluate the prospects of performing real-time digital signal processing on a graphics processing unit (GPU) when linked together with a high-performance digitizer. A graphics card is acquired and an implementation developed that address issues such as transportation of data and capability of coping with the throughput of the data stream. Furthermore, it consists of an algorithm for executing consecutive fast Fourier transforms on the digitized signal together with averaging and visualization of the output spectrum.

An empirical approach has been used when researching different available options for streaming data. For better performance, an analysis of the introduced noise of using single-precision over double-precision has been performed to decide on the required precision in the context of this thesis. The choice of graphics card is based on an empirical investigation coupled with a measurement-based approach.

An implementation in single-precision with streaming from the digitizer, by means of double buffering in CPU RAM, capable of speeds up to 3.0 GB/s is presented. Measurements indicate that even higher bandwidths are possible without overflowing the GPU. Tests show that the implementation is capable of computing the spectrum for transform sizes of 2^{21} , however measurements indicate that higher and lower transform sizes are possible. The results of the computations are visualized in real-time.

Acknowledgments

First of all I would like to thank my examiner, professor Christoph W. Kessler for his guidance and support during this project. Also a big thanks to my university supervisor Usman Dastgeer for his support and helpful advice throughout the project.

I would also like to thank my supervisors at SP Devices Sweden AB, Anders Kagerin and Martin Ohlsson, for all the help and support they have given me during my stay at the company. Many thanks also to Per Löwenborg for the support shown throughout the project and for always providing helpful advice and valuable insights.

Finally I want to thank my family for the support and encouragement they have given me throughout my thesis project and during my entire life.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem formulation	2
1.3	Thesis outline	3
1.4	Definitions	3
2	Theory	4
2.1	Discrete Fourier transform	4
2.2	Fast Fourier transform	5
2.2.1	Fast Fourier transform on graphics processing units	7
2.3	Window functions	8
2.4	Floating-point number representation	10
2.4.1	Errors in summation	12
3	Frameworks, utilities, and hardware	13
3.1	Compute Unified Device Architecture	13
3.1.1	Introduction	13
3.1.2	Memory architecture	15
3.1.3	Memory transfers and concurrency	16
3.1.4	Best practices	17
3.1.5	NVIDIA CUDA Fast Fourier Transform	17
3.2	ADQAPI	19
3.3	NVIDIA Nsight	19
3.4	NVIDIA Visual profiler	20
3.5	ADCaptureLab	20
3.6	Computer system	21
3.6.1	System	21
3.6.2	NVIDIA	22
3.6.3	SP Devices	22
4	Implementation	24
4.1	Feasibility study	24
4.1.1	Streaming	24
4.1.2	Performance	27

4.2	Software architecture	40
4.3	GPU implementation	44
4.3.1	Preprocessing	45
4.3.2	Fast Fourier transform	46
4.3.3	Post processing	46
4.3.4	Kernel optimizations	48
4.4	Visualization	49
4.5	Test set-up	49
5	Results and evaluation	52
5.1	Functional testing	52
5.2	Performance	54
5.2.1	Considerations and limitations	54
5.2.2	Measurements	56
6	Related work	58
7	Conclusion and future work	60
7.1	Conclusion	60
7.2	Future work	61

Chapter 1

Introduction

This chapter introduces the thesis and its context. First the project is related to the surrounding environment of the computer architecture industry. We describe what has been done and what is the reason for this thesis. After that the problem formulation of this thesis is presented. The goal and expected results of the thesis are discussed in more detail. Last we present a list of common abbreviations and words that are used frequently in this document.

1.1 Motivation

For many years the primary approach to achieving higher performance in applications and computer systems has been by means of hardware and different algorithms. It was only recently when heat issues and high power consumption, because of increasing power density, became a large issue, that the focus shifted towards increased parallelism at the hands of the programmer. In general purpose computing some degree of parallelism could be argued for by using threads and processes. On the processor however everything would be executed in a sequential fashion except on instruction level. With the introduction of multi-core architectures and the introduction of general-purpose computing on graphics processing units (GPGPU) a new era of programming started. Instead of trying to increase clock frequency the industry searched for other ways to increase performance. By decreasing frequency and adding more cores to the processor, the task of speeding up programs by means of parallelism was left to the programmers.

For many years the dominant platforms in real-time digital signal processing have been field-programmable gate arrays (FPGA), application specific integrated circuits (ASIC) and digital signal processors (DSP). Most of the development in these systems lies in programming with hardware description languages (HDL) like Verilog [27]. Programming in such languages requires knowledge in digital circuit design as they are used to model electronic components. For many applications they perform very well however

for certain types of problems different platforms could prove beneficial.

There has been much work done in the field of GPGPU when it comes to digital signal processing. There are several libraries, both proprietary and open source, that implements primitives and algorithms for signal processing. By reusing such code the development time for basic signal processing could be reduced by a huge factor. Also it is likely that there are more programmers comfortable with languages like C, than there are with hardware description languages (HDL). This makes development on a graphics processing unit (GPU) an interesting and viable alternative to other platforms which have been used for a long time within the signal processing field.

By using the GPU as an accelerator for signal processing, our hope is to achieve a high-performance alternative to the conventional hardware described above. This thesis is conducted on behalf of the company SP Devices Sweden AB in an effort to evaluate the possibility of joining of their technology of digitizers and A/D converters together with a GPU for real-time digital signal processing. In addition a platform for real-time spectrum analysis is to be created for their Peripheral Component Interconnect Express (PCIe) based units.

1.2 Problem formulation

The main problem of the thesis can be divided into two parts. Streaming data from the digitizer to the graphics processing unit and making sure that the graphics processing unit is capable of handling the data stream.

The real-time aspects of the streaming is one thing to be considered. The demands are on throughput, making sure that all samples are processed. Latency is not prioritized. This means that as long as the PCIe bus is a reliable channel and the graphics processing unit can process the stream without being overloaded, this condition is fulfilled.

The final application on the graphics processing unit shall consist of data formatting and windowing, a fast Fourier transformation and the possibility of averaging the frequency spectrum, all processed in real-time.

The performance of the graphics processing unit must be conforming to the real-time requirements of the application. It needs to process data faster than the bandwidth of the stream. This will be considered for both single- and double-precision. The difference is the bit representation which is 32 and 64 bits respectively. Questions that this thesis tries to answer are:

- What performance can be achieved when using the NVIDIA implemented fast Fourier transform for a fixed set of cards and a fixed transform size?
- For what frame sizes is single-precision sufficient in the context of this project?
- How shall the streaming of data be handled?

- How to map the problem onto the GPU architecture?

1.3 Thesis outline

The rest of the thesis has the following structure:

- Chapter 2 introduces the theory necessary for understanding the remainder of the thesis.
- Chapter 3 introduces different frameworks and equipment that has been used throughout the project.
- Chapter 4 contains a feasibility study and the implementation of the system.
- In chapter 5 results are presented and the system is evaluated.
- Chapter 6 discusses related work.
- Chapter 7 concludes the thesis and discusses future work.

1.4 Definitions

CPU Central processing unit

CUDA Compute unified device architecture

DFT Discrete Fourier transform

Digitizer Device used for the conversion and recording of analog signals in a digital representation

DMA Direct memory access

FFT Fast Fourier transform

FPGA Field-programmable gate array

GPGPU General-purpose computing on graphics processing units

GPU Graphics processing unit

Kernel Function executed on the GPU

PCIe Peripheral Component Interconnect Express

RAM Random access memory

Chapter 2

Theory

This chapter introduces the most important theoretical aspects of the system. It is necessary to understand both the properties and the implications of them to be able to implement and verify the system. First introduced is the discrete Fourier transform and the fast Fourier transform, with the latter being a group of algorithms for faster computation of the discrete Fourier transform. Then some considerations and theory behind windowing and floating-point number representation is introduced.

2.1 Discrete Fourier transform

The Fourier transform can be used to analyse the spectrum of a continuous analog signal. When used on a signal it is a representation of the frequency components of the input signal. In digital signal analysis the discrete Fourier transform is the counterpart of the Fourier transform for analog signals. It is defined in Definition 2.1

Definition 2.1 *Discrete Fourier transform*

$$X[k] = \sum_{n=0}^{N-1} x(n)W_N^{kn}, k \in [0, N-1] \quad (2.1)$$

where $W_N^{kn} = e^{-j\frac{2\pi kn}{N}}$ and is known as the twiddle factor [20].

For any N -point discrete Fourier transform the spectrum of the signal to which it is applied consists of a sequence of frequency bins separated by f_s/N where f_s is the sampling frequency [42]. This gives an intuitive way of understanding how the size of the computed transform affects the output. For large sizes the resolution gets better but the computation time increases because of more points processed.

Each bin has a different representation depending on what mathematical post-processing operation is done. For example the single-sided amplitude

spectrum of the signal can be extracted by first multiplying each bin by $\frac{1}{N}$, where N is the number of points in the transform. Then each bin except the DC component is multiplied by two to take into account the signal power residing at the negative frequencies in the double-sided spectrum. If the input signal is a sinusoidal the spectrum amplitude now corresponds to the amplitude of the input signal. This is discussed in [43, p.108].

The discrete Fourier transform has a time complexity of $\mathcal{O}(N^2)$ which makes it non-ideal for large inputs. This can be seen by looking at Equation 2.1 as there is a summation of N for each k and $k = 0, 1 \dots N - 1$. In Section 2.2 an algorithm for computing the DFT with a lower time complexity is presented.

The transforms mentioned above are functions that operate on complex input and output data. The DFT exhibits an interesting property called conjugate symmetry when supplied with a real-valued input [40]. Only $\frac{N}{2} + 1$ outputs hold vital information. The others are redundant.

2.2 Fast Fourier transform

Computing the discrete Fourier transform directly from its definition is for many applications too slow. When the fast Fourier transform was popularized in 1965 by J.W. Cooley and J.W. Tukey and put into practical use, it was a serious breakthrough in digital signal processing [5, 6]. It relies on using properties of the DFT to reduce the number of calculations.

The Cooley-Tukey FFT algorithm is a divide and conquer algorithm, which means that it relies on recursively dividing the input into smaller sub-blocks. Eventually when the problem is small enough it is solved and the sub-blocks are combined into a final result for the original input. Depending on how many partitions made during the division stage, the algorithm is categorized as a different radix. Let N be the number of inputs such that $N = N_1 N_2$, the next step is to do N_1 or N_2 transforms. If N_1 is chosen it is called a decimation in time implementation and if N_2 is chosen it is called a decimation in frequency. The difference between the two approaches is in what order the operations of the algorithm is performed. The most common implementation is the radix-2 decimation in time algorithm. For $N = 2^M$ the algorithm will do $\log_2 N = M$ divide steps with multiplications and summations resulting in a time complexity of $\mathcal{O}(N \log N)$. All the operations consist of a small operation called the butterfly and it is the input and output indexes that differs. Figure 2.1 show a radix-2 decimation in time butterfly and Figure 2.2 show a decimation in frequency butterfly. By using the properties of symmetry and periodicity of the discrete Fourier transform, some of the summations and multiplications of the DFT are removed. There are also implementations that have a time complexity of $\mathcal{O}(N \log N)$ where N is not a power of two. This and much more can be read e.g. in [20] by Duhamel et al.

When using a real-valued input to the FFT, the operation count as well

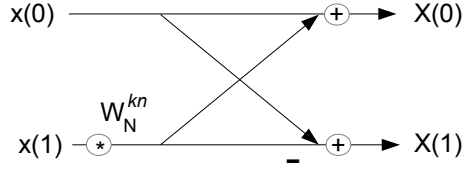


Figure 2.1: DIT butterfly.

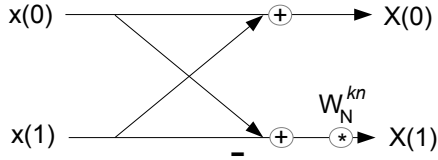


Figure 2.2: DIF butterfly.

as the memory storage requirements are essentially halved compared to a complex valued input sequence. This is because of the conjugate symmetry property of the DFT. This is explained in further detail in [39].

The outputs of an in-place FFT will be in reversed order. It is called bit-reversed order and is an inherent property of the operations done in the FFT algorithm. There are several ways how this can be fixed. The inputs can be placed in bit-reversed order or the transform can be executed out-of-place. Figure 2.3 shows the operations performed in an 8-point radix-2 decimation in time FFT.

It is important to note that the FFT produces the same result as the DFT. The only difference is the execution of operations.

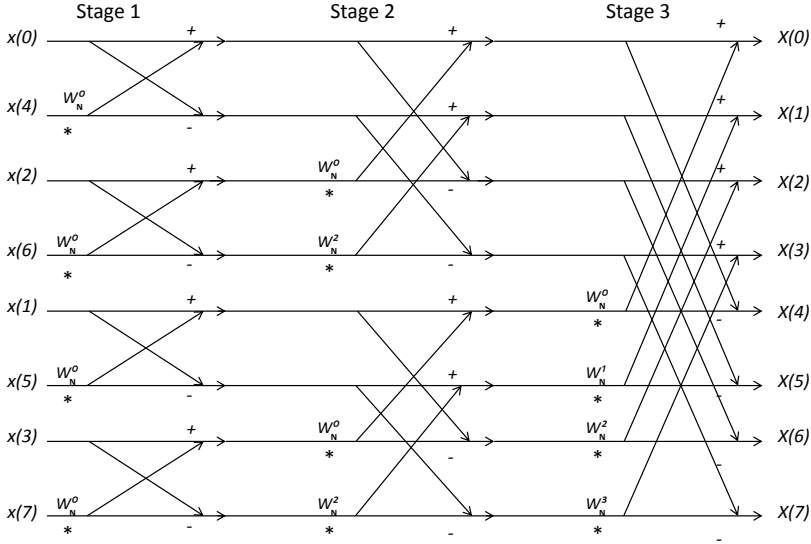


Figure 2.3: Signal flow graph of 8-point radix-2 DIT FFT.

2.2.1 Fast Fourier transform on graphics processing units

The last section showed that there are a number of independent operations during each stage in the FFT algorithm. This property can be exploited in a parallel architecture like the GPU. It is common to measure the performance of a floating-point computation unit with the unit of Floating-point operations per second (FLOPS). The definition of FLOPS is presented in Definition 2.2

Definition 2.2 *Floating-point operations per second*

$$FLOPS = n_{cores} n_{flop} f \quad (2.2)$$

where n_{cores} is the number of cores, f is the frequency, and n_{flop} is the average number of floating-point operations per cycle in one core.

When approximating performance for FFTs that implements the Cooley-Tukey algorithm with floating-point hardware it is common to use Equation 2.3 for complex transforms and Equation 2.4 for real input transforms.

$$GFLOP/s = \frac{5N \log_2 N}{t} 10^{-9} \quad (2.3)$$

$$GFLOP/s = \frac{2.5N \log_2 N}{t} 10^{-9} \quad (2.4)$$

where t is the measured execution time of the algorithm and N is the number of elements.

The difference between the two formulas is a consequence of the symmetric properties of real input transforms described in Section 2.2. The constants provide an estimation for the number of operations used in the executions. Cui et al. [18], Li et al. [33], and Govindaraju et al. [25] all use this metric to estimate the GFLOP count per second of FFT algorithms. The FFT used in this thesis, which is based upon the FFTW [23] implementation, also uses this metric.

2.3 Window functions

When processing a continuous signal in time it is necessary for every observation that we make that it is finite. It is possible to change the length of the observation but it will always be finite. This has implications when using transforms like the discrete Fourier transform. Because the DFT consider its inputs to be periodic, as discussed in [26], the expected value after N samples is the value of sample zero. This can be seen in the Figure 2.4.

We introduce the window function that is applied to an input signal by multiplying the input with a window function w as shown in Equation 2.5

$$y[n] = x[n]w[n] \quad (2.5)$$

where $w = 0 \ \forall n \notin [0, N - 1]$ and $[0, N - 1]$ is the observation window with N points.

Windowing is done implicitly when using the DFT on a continuous signal and is defined as the rectangular window.

Definition 2.3 *Rectangular window*

$$w[n] = 1 \ \forall n \in [0, N - 1] \quad (2.6)$$

A phenomenon called spectral leakage occur when the observed signal has discontinuities between the first and last sample. The discontinuities contribute to spectral leakage over the whole set of the transform frequency bins. The use of different types of window functions all deal with this fact. By reducing the importance of signal values closer to the edges of the observation window the leakage is reduced. There are many different windows used for different types of applications. The window function should be decided based on application context.

Another common window is the Hann window.

Definition 2.4 *Hann window*

$$w[n] = 0.5(1 - \cos \frac{2n\pi}{N}) \ \forall n \in [0, N - 1] \quad (2.7)$$

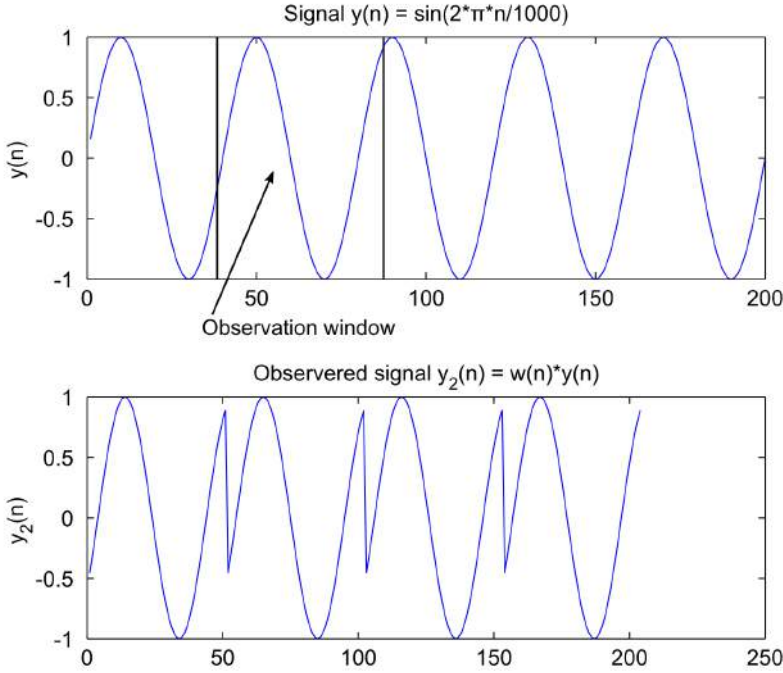


Figure 2.4: Observed signal with improper use of the discrete Fourier transform.

By examining the Hann window in the time domain it is obvious that it will bring the input signal values down close to zero at the edges of the window. Figure 2.5 shows the Hann window.

We will discuss two types of windowing effects that distort the frequency spectrum when a window other than the rectangular window is used. They are called coherent gain and equivalent noise bandwidth (ENBW) and are defined below. The coherent gain originates from the fact that the signal is brought to zero close to the boundaries of the window. This effectively removes parts of the signal energy from the resulting spectrum. The coherent gain can be computed from the window by the formula presented in Definition 2.5.

Definition 2.5 *Coherent gain*

$$CG = \sum_{n=0}^{N-1} w[n] \quad (2.8)$$

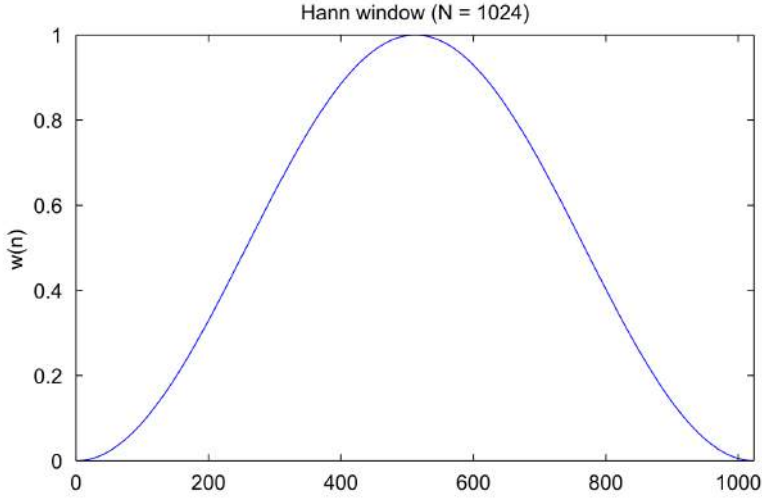


Figure 2.5: The Hann window function.

The coherent gain is also called the DC gain of the window. The frequency bins of the windowed transformed signal can be scaled to their correct magnitude by the normalized coherent gain. The coherent gain is normalized by division of N . Then the amplitude may be scaled by dividing each bin by the normalized coherent gain. A more thorough explanation of this process is described in [36, p.161].

The equivalent noise bandwidth comes from the fact that the window bandwidth is not unity. In this case the window acts as a filter accumulating noise over the estimate of its bandwidth. This results in an incorrect power of the spectrum.

Definition 2.6 *Equivalent noise bandwidth*

$$ENBW = \frac{\sum_{n=0}^{N-1} w^2[n]}{\left[\sum_{n=0}^{N-1} w[n] \right]^2} \quad (2.9)$$

The normalized equivalent noise bandwidth is the ENBW multiplied by the number of points in the window. The effects of coherent gain and equivalent noise bandwidth are discussed in [4, p.192-193].

2.4 Floating-point number representation

Most computer systems use the IEEE 754 standard [29] if they support floating-point operations. It defines the representation of floating-point numbers for binary and decimal number formats. Furthermore it defines

different types of rounding operations and arithmetic operations. It also defines exception handling for situations like division by zero and interchange formats used for efficient storage.

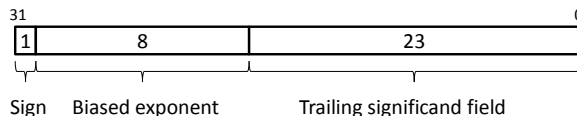


Figure 2.6: Single-precision format.

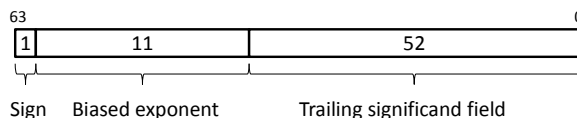


Figure 2.7: Double-precision format.

Figure 2.6 demonstrates the binary single-precision format and Figure 2.7 demonstrates the binary double-precision format defined by the standard. These are the most common floating-point number representations used, and from now on they will be referred to as single-precision and double-precision. What representation to use depends on the application context. If there is a need for high precision or the computation involves a lot of floating-point arithmetic operations, double-precision will most likely be required.

When using floating-point numbers it is important to be aware of some of the intrinsic properties related to its format. When representing the decimal number 0.1 in binary floating-point format it is an infinite sequence of zeros and ones. It cannot be represented as a finite sequence of digits. This is called rounding error and is discussed in [24]. Also floating-point numbers are not uniformly spaced. Numbers close to zero have a smaller distance to the next number possible to represent in floating-point format than large numbers. This issue is discussed in [38, p.514]. This fact also shows the additional errors introduced when using arithmetic operations on large floating-point numbers compared to small. For example, $(x_{small} + x_{small}) + x_{large}$ might not be equal to $x_{small} + (x_{small} + x_{large})$.

The results of arithmetic operations on floating-point numbers are different from that of integers. The mathematical law of associativity does not hold for floating-point operations. More information on the subject can be found in [24]. The implication of this is that the order of operation for instructions matter. By extension, for different hardware to compute the

same results in a floating-point operation they need to match each other precisely in the order of executed floating-point instructions. This is true for both algorithmic and hardware level.

In 2008 the standard was revised. One important addition is the fused-multiply-add(FMA) which basically removes one rounding step when doing a multiplication together with an addition. Normally without FMA the order of operations for $A * B + C$ would be $round(round(A * B) + C)$ effectively introducing a round-off error in the computation. In the white paper [44] on floating-point computations by N. Whitehead et al. there is an interesting example comparing a computation with FMA and without. It shows an error of $4 * 10^{-5}$ when using FMA and an error of $64 * 10^{-5}$ without FMA compared to the correct answer.

2.4.1 Errors in summation

D. Goldberg shows in [24] that errors in summation for a floating-point calculation are bounded by the term $ne \sum_{i=1}^n |x_i|$ where e is machine epsilon [45]. It is also stated that going from single-precision to double-precision has the effect of squaring e . Since $e \ll 1$ this is clearly a large reduction in the bound of the summation error.

Chapter 3

Frameworks, utilities, and hardware

This chapter describes the frameworks and tools that have been used to implement the software application. First we introduce the two major frameworks used: CUDA that is used to communicate with the GPU, made by NVIDIA, and ADQAPI used to communicate with the devices made by SP Devices AB. This is followed by debugging utilities from NVIDIA and SP Devices. Last the system used is described and some of the components it consist of.

3.1 Compute Unified Device Architecture

This section will discuss some of the features and functionality of GPGPU programming that have been used in the thesis. First a general overview is presented, then some discussion regarding the memory hierarchy. After that memory transfers and concurrency are discussed as well as best practices along with an introduction of the CUDA Fast Fourier Transform (cuFFT) library.

3.1.1 Introduction

Compute Unified Device Architecture or CUDA is used to communicate with graphics processing units manufactured by NVIDIA [13]. It is an extension of the C language with its own compiler. Nvcc is actually a wrapper for the supported C compiler with some added functionality. It adds support for all of the API calls used by CUDA to set-up and execute functions on the GPU. It supports executing kernels, modifications of GPU RAM memory, and the use of libraries with predefined functions among other things. To support concurrent execution between the GPU and CPU there are several tasks on the GPU that are non-blocking. This means that they will return

control to the CPU before they are done executing. These are listed in [16] and presented below:

- Kernel launches.
- Memory copies between two addresses on the same device.
- Memory copies from host to device of a memory block of 64 KB or less.
- Memory copies performed by functions that are suffixed with Async.
- Memory set function calls.

The graphics processing unit is fundamentally different in its hardware architecture compared to a normal central processing unit (CPU). While the CPU relies on fast execution in a serialized manner the GPU focuses on instruction throughput by parallel execution. Figure 3.1 shows the general idea of a CPU compared to a GPU. The use of less complex cores and more area of the chip die dedicated to processing cores makes it possible to implement chips with a huge amount of cores. The cores, or processing units, contain small arithmetic functional units. They are grouped into one or multiple streaming multiprocessors. A streaming multiprocessor also has functional units for floating-point computations. It is also responsible for creating, managing, scheduling and executing threads in groups of 32 called warps. Figure 3.2 shows the concept of the streaming multiprocessor. The figure also shows the extended programming model used to specify how work should be executed on the GPU. The CUDA programming model uses

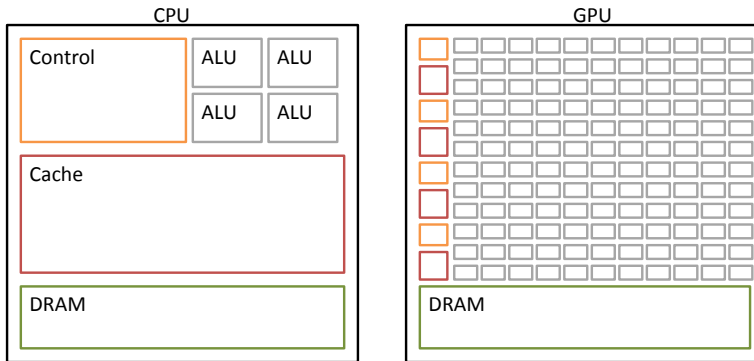


Figure 3.1: The difference between the general design of a CPU and the GPU.

threads mapped into blocks, which are mapped into grids. Figure 3.2 shows the model for the two-dimensional case however both grids and blocks can be extended into the third dimension. Each thread will be mapped to a processing unit and each block will be mapped to a streaming multiprocessor. Threads can communicate with each other within a block and shared memory can be used to coordinate memory loads and computations. Once a block is mapped to a multiprocessor the threads of the block are divided into warps that are executed concurrently. This means that best performance is attained when all threads within a warp execute the same control path and there is no divergence. If there is divergence each control path will be executed sequentially and eventually be merged together when both paths are done. The execution context for a warp is saved on-chip during its entire life time, meaning that there is no cost in switching between warps at every time of instruction issue. Each instruction issue the streaming multiprocessor will choose a warp with threads ready to execute their next instruction and issues them for execution.

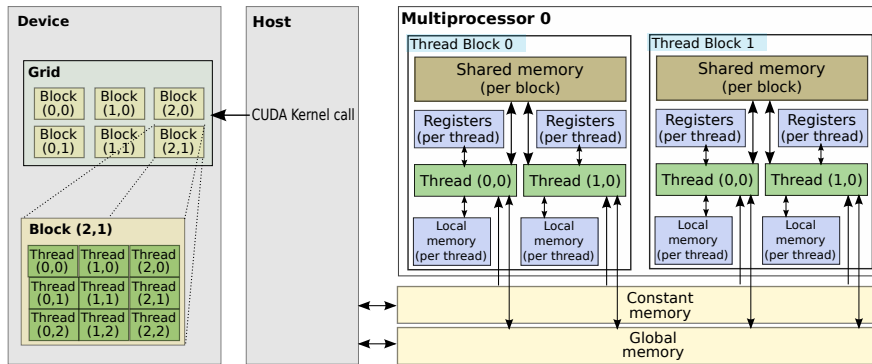


Figure 3.2: The CUDA model and how it is mapped into the architecture. Taken from [19].

As a means of classification NVIDIA based GPUs are divided into different classes called compute capability. Depending on the compute capability of a device, different functions are available. More information can be found in the CUDA documentation [16].

3.1.2 Memory architecture

There are three different types of memories, managed by the programmer, that are commonly used in GPGPU applications in CUDA. These are the global memory, shared memory, and constant memory. There is a fourth memory called texture memory, that is mostly used for imaging processing since it is optimized for such applications.

Memory access patterns of a CUDA application can severely cripple the

execution performance if done incorrectly. The most important part is to make sure that memory accesses to global memory are coalesced and to avoid bank conflicts in shared memory. The process of making coalesced memory accesses depends on the compute capability of the device; a higher number generally means that more of the complexity is handled in hardware. Also the introduction of a L2 cache for higher compute capability cards affects the severity of non-coalesced accesses. A general rule of thumb is to do aligned accesses, which means that the access should lie within the same memory segment; and the segment size depends on the word access size. Also the accesses should be to consecutive addresses, i.e. thread one should access memory element one etc.

The shared memory is a fast on-chip memory shared between threads in a block on the streaming multiprocessor. It is divided into equally sized banks that can be accessed concurrently, and with no conflict it performs at its maximum. If there are several accesses to the same memory bank they need to be serialized and performance is lost. As with coalesced accesses, the access patterns differ between different compute capability, however information regarding the subject can be found in the CUDA documentation [16].

The constant memory can be used when data does not change during kernel execution. It resides in global memory; however it is cached in the constant cache. If all threads in a kernel read the same data, all but one access will be to the fast constant cache, speeding up the global throughput.

3.1.3 Memory transfers and concurrency

CUDA supports different alternatives for transferring data from host memory to GPU memory: Direct memory copy, pinned memory, and mapped memory. The direct memory copy is the basic data transfer function. When transferring data of more than 64 KB it blocks further CPU execution until it is finished. It will query the operating system asking for permission to transfer data from or to the memory location specified in the function call. The operating system acknowledges the request if the memory specified belongs to the process from which the function call was made and the correct page is present in main memory. In some applications, mainly those that require a certain performance, this is not acceptable. It might be necessary that memory copies are non-blocking. Also the throughput of the memory transfers will be limited by the risk of data not being present in main memory. It is however possible to page-lock host memory, called pinned memory in CUDA terminology. By using CUDA API calls the operating system locks the memory locations that hold data that is going to be transferred from host to GPU or vice versa. The data can no longer be swapped from main memory to disk. By using the Async suffix, data transfers can be handled by the GPU DMA controller and the CPU can continue executing the host program. This also improves data transfer throughput. Mapped memory is

similar to the pinned memory approach with the exception of that there is no need for explicit memory copies. Instead the data resides in host RAM and when it is requested by a device kernel the data is transferred, transparently to the programmer.

CUDA also support concurrent memory transfers and concurrent kernel execution on some devices. Concurrent memory transfers, that is, writing and reading on the PCIe bus at the same time, is only supported on computing cards such as Tesla [17]. They have two DMA engines which make this possible. It is possible to query the "asyncEngineCount" to verify what type of execution is supported. If the response is 2 then the device is capable of concurrent memory transfers, i.e. Tesla computation cards; if it is 1 then kernel execution may overlap with one PCIe bus transfer. The usage of concurrent memory transfers require pinned memory. By querying "concurrentKernels" the response shows whether several kernels can execute at the same time on the GPU.

The concept of concurrent kernels and memory transfers is closely related to the concept of streams. The use of streams is necessary to achieve the concurrency mentioned above. When executing operations on the GPU in different streams they are guaranteed to be sequentially executed within their stream. But when executing kernel A into stream 1 and kernel B into stream 2, there is no guarantee as to which kernel will start first or finish first. Only that they both execute eventually. It is however possible to do synchronization between streams by using events created in CUDA. Events are also useful for measuring execution time on the GPU.

3.1.4 Best practices

When developing heterogeneous applications there are guidelines that the programmer is encouraged to follow. This is described in the Kepler tuning guide in the NVIDIA documentation [16]. The most important guidelines are as follows:

- Find ways to parallelize sequential code.
- Minimize data transfers between the host and the device.
- Adjust the kernel launch configuration to maximize device utilization.
- Ensure that global memory accesses are coalesced.
- Minimize redundant accesses to global memory whenever possible.
- Avoid different execution paths within the same warp.

3.1.5 NVIDIA CUDA Fast Fourier Transform

The CUDA Fast Fourier Transform library implements a framework to quickly start developing algorithms that involve time domain to frequency

domain conversion and vice versa. It implements transforms for complex and real data types in one, two, and three dimensions. The maximum data size for one-dimensional transforms is 128 million elements. Depending on the size of the transform the library implements different algorithms for computing the FFT or IFFT. For best performance however the input size should be in the form of $2^a * 3^b * 5^c * 7^d$ to use the Cooley-Tukey algorithm which implements decomposition in the form of radix-2, radix-3, radix-5 and radix-7. Transforms can be made in both single- and double-precision with in-place and out-of-place transforms. Also it is possible to do batched executions, which means that the input size is split into smaller blocks, each of them used as the input to the FFT algorithm [10].

Before executing a transform the user needs to configure a plan. The plan specifies what type of transform is to be done among other things. There are a couple of basic plans that set up the environment for the different dimensions as well as a more advanced configuration option named `cufftPlanMany`. The basic plans are created by setting how many dimensions the transform is to operate in and the type of input and output data. When using the function `cufftPlanMany()` the user can specify more detailed execution settings. Batched executions are configured in this mode as well as more advanced data layouts. The plan can be configured to use input data with different types of offsets and output data in different layouts as well.

The data layout of the transform differs depending on what data type that is used as input and output. The library redefines single-precision as `cufftReal` and double-precision as `cufftDoubleReal` coupled with `cufftComplex` and `cufftDoubleComplex`. The last two data types are structs of two floats or doubles respectively. Table 3.1 below shows the data layout for different types of transforms. The FFT between real and complex data type

Transform type	Input data size	Output data size
Complex to complex	N <code>cufftComplex</code>	N <code>cufftComplex</code>
Complex to real	$\frac{N}{2} + 1$ <code>cufftComplex</code>	N <code>cufftReal</code>
Real to complex	N <code>cufftReal</code>	$\frac{N}{2} + 1$ <code>cufftComplex</code>

Table 3.1: Table taken from NVIDIA CUDA cuFFT documentation [10].

produces the non-redundant $[0, \frac{N}{2} + 1]$ complex Fourier coefficients. In the same manner the inverse FFT between complex and real data type operates on the non-redundant complex Fourier coefficients. Also they take advantage of the conjugate symmetry property mentioned in Section 2.2 when the transform size is a power of two and a multiple of 4.

The library is modelled after the FFTW [23] library for CPU computation of fast Fourier transforms. For simplified portability there are features to make cuFFT work similar to FFTW, like using different output patterns. This is the default setting but can be modified with the function `cufftSet-`

CompatibilityMode().

When it comes to accuracy the Cooley-Tukey implementation has a relative error growth rate of $\log_2 N$ where N is the transform size as mentioned in the cuFFT documentation [10].

3.2 ADQAPI

ADQAPI is the framework developed by SP Devices AB to communicate with their digitizer products. There are two implementations, one in C and one in C++. All communication with the devices from SP Devices goes through the API calls. This involves finding different types of devices, creating their runtime software environment as well as much other functionality. The capabilities of different devices are reflected in the API. The configuration of different devices may vary slightly depending on their functionality. Using the streaming functionality of the digitizer involves a few technical details on how to set up the device. But the most important thing is how the data is handled in host RAM. When setting up the device it is encouraged to decide on a buffer size and how many buffers to allocate. This will create buffers in host memory RAM that the SP Devices digitizer will stream into without interaction from any program. This works in the same way as pinned memory explained in Section 3.1. To allow ADQAPI and CUDA to share buffers, pointers to the buffers created by the ADQAPI can be extracted with a function call. The streamed data must be handled by the API calls. If not, the buffers will eventually overflow and samples will be lost. The streaming procedure can be described as follows:

```

initialization;
while true do
    wait until buffer ready;
    get new buffer;
    do something with the data;
end
```

Algorithm 1: Streaming procedure in the host computer.

3.3 NVIDIA Nsight

The Nsight [14] software is the NVIDIA-supported add-on used to debug and trace execution of CUDA C/C++, OpenCL, Direct Compute, Direct3D and OpenGL applications. It is available as an add-on to Microsoft Visual Studio [8] for Windows and for Eclipse [22] in Linux and Mac OS X. Nsight adds several options that are useful for debugging and tracing application execution on the GPU. Features include but are not limited to:

- Set GPU breakpoints.
- Memory check to check for alignment error and bad pointers.

- Disassembly of the source code.
- View GPU memory.
- View the internal state of the GPU.

Nsight also adds features to support performance analysis to the IDE. It is possible to get a detailed report of the application. It shows GPU utilization, graphical execution traces, and detailed information about CUDA API calls. By looking at the graphical view of the application's execution it is possible to see if the application is behaving as expected, e.g. if the memory transfers and kernel executions are overlapping as expected, and other types of control flow operations.

3.4 NVIDIA Visual profiler

The NVIDIA Visual profiler is a standalone cross-platform profiling application. It includes, but is not limited to, a guided analysis tool that provides a step-by-step walk-through of some of the application's performance metrics. The analysis outputs a report of what parts of the application might benefit from improvements based on different performance metrics. It consist of an analysis of the total application in terms of bandwidth, achieved occupancy in the multiprocessors, and compute utilization. In addition to the first analysis it is also possible to analyse individual kernels on a detailed level. It is possible to do latency analysis, compute analysis and memory bandwidth analysis.

Furthermore it is possible, on a detailed level, to decide what type of metrics should be recorded on the GPU. There are many different types of metrics for categories such as: memory, instruction, multiprocessor, cache and texture cache. It is also possible to record many different types of low-level events like "warps launched" and "threads launched".

Like the Nsight debugger, the visual profiler includes a graphical trace of the execution time line of an application. It is also possible to view a unified CPU and GPU time line. Furthermore it is possible to place trackers in source code to manually decide when the profiler should collect traces.

3.5 ADCaptureLab

ADCaptureLab is a software program developed by SP Devices to use in conjunction with their digitizers. It is used as an analysis and capture tool. ADCaptureLab provides a graphical display of both the input signal and its frequency spectrum. The FFT size is capped at $2^{16} = 65536$ and is executed on the CPU. The software makes it possible to trigger on signal input values, different software triggers and external triggers. It is also possible to use different types of windows. In addition to this there are some

analysis measurements of the signals like SNDR and SFDR. Of course there are many device-specific options like specifying a sampling rate among other options. There is also an averaging feature implemented in the program. Figure 3.3 shows the main window view of ADCaptureLab.

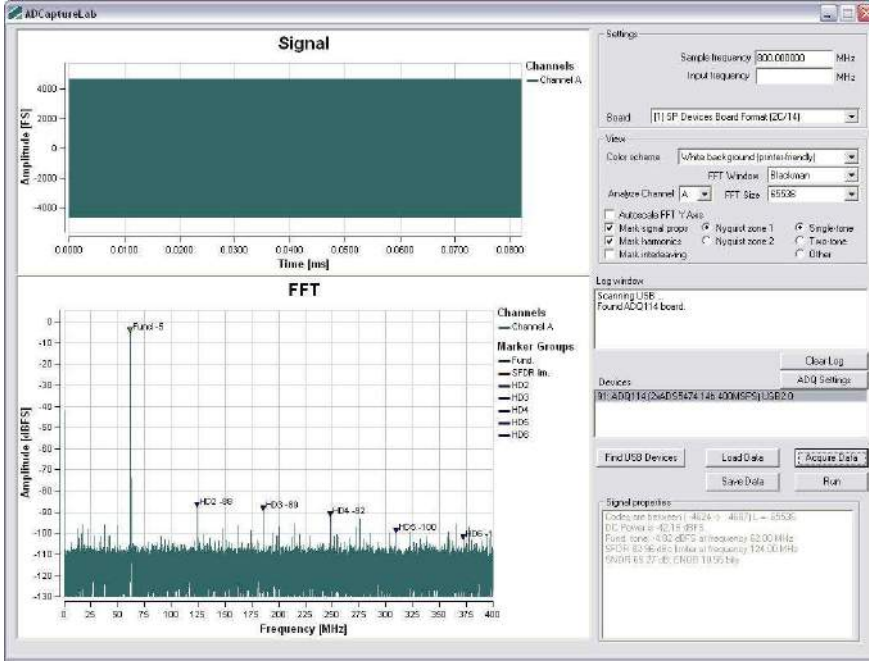


Figure 3.3: ADCaptureLab showing an input signal and its amplitude spectrum.

ADCaptureLab provides added security to verify correct functionality for the system. By providing another graphical representation of the input signal and the frequency spectrum it was used as reference to the visualization implemented in this thesis.

3.6 Computer system

This section describes the system that is used for the implementation part of the thesis. First the basic computer system is described with regards to: CPU, RAM, motherboard and interfaces used. Then we describe the graphics cards used followed by the devices provided by SP Devices.

3.6.1 System

The computer is under normal circumstances a test computer for SP Devices PCIe-based cards and is running Windows 7. SP Devices requested

that the application was created in this environment since many of their customers use Windows. Furthermore the application is created with CUDA 5.5. Below a brief overview of the system is presented in Table 3.2.

Components	Model
CPU	INTEL CORE i7-950
Motherboard	ASUS P6T7 WS SUPERCOMPUTER
RAM	PC3-16000 DDR3 SDRAM UDIMM (6 GB, CL 9 - 9 - 9 - 24, clocked at: 1333 MHz)
Graphics card	ASUS GTX 780 DirectCU II OC
Graphics card	NVIDIA GeForce GT 640 (GK208) [GIGABYTE]

Table 3.2: System specification.

3.6.2 NVIDIA

Geforce GTX 780

The GTX 780 is the main computation card used for the software implementation part of the thesis. It is a high-end gaming graphics card developed by NVIDIA. It is part of the Kepler architecture, described in a white paper [9] by NVIDIA. Because the card shares the chipset architecture with the Tesla computation card family they have some common characteristics. The GTX 780 possesses both the Dynamic Parallelism and HyperQ features of the Kepler architecture. However it is not capable of concurrent memory transfers. It does not have access to any of the GPUDirect [11] features or is capable of high-performance double-precision calculations. The single-precision performance is the same as the computation cards and it has a high memory bandwidth. It implements PCIe 3.0 x16 lanes.

Geforce GT 640

The GT 640 is a low-end card from the Kepler architecture family [9]. It is used as the graphics rendering card in the system.

3.6.3 SP Devices

ADQ1600

The ADQ1600 is the digitizer used for the final implementation in this thesis. It consists of, among other things, four A/D converters and SP Devices interleaving technology ADX that interleaves the results to one single sampled output. The digitizer produces 14-bit resolution at 1.6 GSPS. It is possible to send the samples as a 14-bit stream or pack each sample as 16-bit words, both of which are represented in two-complement. When packed as 16-bit words they are aligned to the most significant bit. When using

the packed data format and full sampling speed of 1600 MHz the digitizer is producing data at a rate of 3.2 GB/s. The ADQ1600 is available with several different interfaces to connect it with various systems. In this thesis the PCIe interface is used. The ADQ1600 implements PCIe 2.0 8x lanes. There is a 144 KB FIFO buffer memory residing in the FPGA in the digitizer that is used to store samples before they are sent on the PCIe bus. Additionally the digitizer is capable of keeping a record of 32 host RAM buffers at a time. This means that allocating more buffers will not help during the time that the operating system is occupied with other tasks not related to updating the digitizer's DMA engine. A picture of the ADQ1600 PCIe version is presented in Figure 3.4.



Figure 3.4: ADQ1600.

ADQDSP

The ADQDSP is a digital signal processing card created by SP Devices. It is used as a standalone device for calculations, signal recording and real-time digital signal processing. It has been used to test and verify parts of the design. More specifically it has been used for functional tests and data flow verification.

Chapter 4

Implementation

This chapter introduces the implementation part of the thesis. First the feasibility study is presented and explained. It concerns how the streaming of data is constructed and how to attain the required performance necessary for correct execution. Then the general structure of the implementation is described followed by the individual parts and optimizations. Last we present how the visualisation of the results and the integration of the test environment is implemented.

4.1 Feasibility study

Because of the demands placed on the application there are items to investigate before an implementation can be considered. This section presents the initial work done to decide how to proceed with the implementation.

First an investigation regarding the streaming capabilities of the digitizer in conjunction with a graphics card is discussed. Here we present the chosen approach and the rationale for the choice.

Second there are performance requirements to be considered when it comes to computations on the GPU. The chosen approach and result of the research is presented.

4.1.1 Streaming

In this thesis there is one data path that will be the same for all types of applications. It is the first destination of the acquired data samples measured by the digitizer. They shall be sent from the digitizer to the memory of the GPU. This is ideally implemented with a peer-to-peer solution. NVIDIA provides a feature called GPUDirect RDMA [11, 12] that implements a peer-to-peer protocol for their graphics cards. Also because of the requirements in performance it is reasonable to think that it might be necessary to implement a peer-to-peer solution to be able to match the sampling speed of

the digitizer. There are however some problems with the intended approach and it will be described in the following sections. Eventually the idea of implementing a peer-to-peer solution was rejected in favour of a solution that relies on buffering the acquired data in host RAM before relaying it to the GPU.

Peer-to-peer

The initial idea to get the samples from the digitizer to the GPU involved using the NVIDIA feature called RDMA [12]. Quite soon it became clear that RDMA [12] is a feature supported only on computation cards and not on regular gaming cards [30, 3, 11]. This was a problem because SP Devices required the solution to be supported on both gaming cards and computation cards.

The ideal implementation would be to pass a physical memory address pointing into GPU memory to the DMA controller of the FPGA in the digitizer. By doing this the FPGA will be able to initiate PCIe bus writes whenever enough samples have been collected and can be transmitted. PCIe bus writes experience lower latencies than bus reads as the device that initiates the transfer does not need to query its recipient in the same manner as during a read operation. By locking buffers in the memory of the recipient system, a write on the PCIe bus will be launched as soon as it is ready. In a read operation however there are two packets that must be sent, one asking for data at a specific memory location and the packet sent back with that data.

Other options were discussed and research made to find other strategies to be able to stream data directly into GPU memory. One other approach was found and investigated. A more thorough description of the implementation can be found in the paper [3] by Bittner et al. The end result would be almost the same as using RDMA, however this implementation relies on doing read cycles over the PCIe bus instead of writes. It involves mapping the memory available in the FPGA onto the PCIe bus to expose it to the rest of the system. By passing the address of the FPGA memory to functions in the CUDA API used for pinning memory, the memory will be page-locked and regarded as host memory from the GPU point of view. Pinned memory is explained in greater detail in Section 3.1. When this is done the GPU will be able to read contents directly from the FPGA memory by initiating PCIe read requests to the FPGA DMA controller.

This seems like a viable option at first but there are problems with this approach when used in conjunction with the ADQ1600 provided by SP Devices. When the GPU acts as a master in the transfer of samples, the control of transmitting samples as they are acquired by the digitizer is lost. There must be a way to buffer samples so that there is no risk of a sample being overwritten before being sent off for processing. The ADQ1600 does not have a large memory as it is not necessary for its normal application domain. The only buffering being done is in a small memory instantiated

within the FPGA that supports interrupts for approximately $45\ \mu\text{s}$. The memory is 144 KB in size according to SP Devices and the digitizer produces data at a rate of 3.2 GB/s. The memory would definitely need to be bigger for an approach like this to be realized.

Finally the possibility of using a computation card was investigated. By looking more closely into the documentation [12] it became clear that it does indeed support a way to extract physical memory locations directly into the GPU. It does not however provide physical addresses directly but instead it delivers a data structure that contains a list of physical addresses i.e. it delivers a page table. This is not a problem if the DMA controller used is capable of handling such structures, meaning that it can view a list of addresses as contiguous memory, directly writing to the different locations in order. The ADQ1600 DMA controller does not provide this feature. Implementing such a feature was deemed too time consuming for this thesis project and a different solution was necessary.

Host RAM

The result of the investigation of peer-to-peer streaming provided little options but to look for different alternatives. The option of streaming data from the digitizer to the host RAM of the system and then transfer it to the GPU is a possibility. There are however limitations with this approach and some concerns as well. The main concerns are:

- Is it possible to maintain the speed of 3.2 GB/s throughout the transfer?
- Is it possible to lock the same memory location from two devices at the same time?
- The data flow path becomes more complex and must be taken into consideration with regards to the physical placement of the cards. This is to make sure that some paths are not overloaded.

In their PCIe based digitizers, SP Devices has implemented a solution to stream data from their cards to the host RAM. By locking buffers in system memory via the operating system's system call interface, it is possible to stream data at the sample rate of the digitizer. The ADQ1600 implements the PCIe 2.0 x8. It provides a speed of 500 MB/s per lane which adds up to 4 GB/s in each direction. The encoding is 8b/10b which gives the maximum theoretical bandwidth of 3.2 GB/s. More information about the PCIe standard can be read in the FAQ [35] on the homepage of PCI-SIG who maintain the PCIe standard. This indicates that the GPU would need to fetch in at least 3.2 GB/s without any delays for the throughput through the entire system to be the same as the digitizer. This is obviously not realistic and there is a need for higher rate of data transfer from the host RAM to the GPU than from the digitizer to the host RAM. The system used in this

project supports PCIe 2.0 x16 which provides a max theoretical bandwidth of 6.4 GB/s. It is however difficult to conclude exactly how much extra bandwidth is needed since this depends on various parameters like: buffer size in host RAM, bandwidth of the host RAM, interrupts caused by the operating system, and delays in transfers and computations caused by the GPU.

To fetch data from host RAM to GPU memory at the maximum speed allowed by the motherboard or GPU PCIe interface, the need for using pinned memory arises. This means that both of these devices need to lock the same memory in host RAM for this solution to be viable. By testing this on a low-end graphics card, the GT 640, it was verified to work. Furthermore, an implementation measuring the bandwidth between the digitizer and the graphics card showed a rate close or equal to the maximum streaming rate even though they both implement the same PCIe interface. It is likely that with a card of higher bandwidth, it will be possible to stream data at the required rate.

There are a couple of ways the data flow can be handled and it depends on the end destination of the calculations. One thing is clear however, if all data is to be transferred off the GPU back to the host, one of two conditions must be fulfilled:

- The data rate to and from the GPU must be > 6.4 GB/s.
- The GPU must provide bi-directional data transfers.

One option is to employ a transfer that is time-multiplexed, meaning that data is transferred to the GPU and off the GPU at different points in time. Because the rate to the GPU must be higher than 3.2 GB/s to be able to handle the data rate of the digitizer, the rate from the GPU must conform to the same condition. The other possible option is that data is transferred in parallel to and off the GPU. The incoming data are samples to be processed, and the outgoing data is the result of the computations. This is possible if the GPU provides bi-directional data transfers.

The digitizer needs 3.2 GB/s write access on its path to host RAM and the GPU is capable of 6.4 GB/s reads and writes. This suggests that the card used for computation needs one PCIe switch by itself, which suggest that the graphics rendering card and ADQ1600 will reside under one switch. Results presented later in this thesis will however conclude that this will not work. The digitizer needs its own switch for stability reasons.

4.1.2 Performance

The real-time constraints coupled with the speed of the digitizer places requirements on the graphics card that must be investigated.

As for precision it is highly related to the performance of the computations. Using single-precision is faster for GPGPU computation. However

the reduced amount of bits produces a less accurate result. If it is possible to use single-precision instead of double-precision it will lead to a substantial performance improvement. It also opens up for the possibility of using a wide range of graphics cards primarily used for games available at less expensive prices than computation cards.

As for predicting performance on different cards there is, as far as we know, no definitive method for size N FFTs performed with the cuFFT library. Partly because it is closed source but also because it depends greatly on what architecture it is deployed upon. When predicting performance an empirical method has been used in conjunction with a measurement based approach.

First the possibility of using single-precision in the application is evaluated. Then a card is decided based upon the outcome of the evaluation.

Single- versus double-precision

When deciding what type of card to use in the final implementation it was necessary to make some judgements whether to use single-precision or double-precision. There are two parts of the application that this will be considered for; the FFT computation and the averaging part. The most important part when it comes to performance is the FFT computation. There is a possibility that for larger transform sizes the impact of less accurate results in single-precision could be too big to neglect. The impact of using single-precision in FFT computations was evaluated by creating a test implementation in Matlab and C. It involves executing several FFTs and inverse FFTs in both single- and double-precision to analyse the difference between the transformed signals with Matlab.

As for the summations made during the averaging part there is a discussion in Section 2.4 regarding the summation of floating-point numbers in single- and double-precision. It was decided to use double-precision instead of single-precision for the averaging part of the system. This is because the difference in performance is barely noticeable but the decrease in accuracy might be substantial when using single-precision.

Precision in fast Fourier transforms

When comparing the difference between single- and double-precision in CUDA FFTs, two measurements have been used. The first test calculates the squared difference of each point in time, of the transformed and inverse transformed signal, from single- and double-precision in CUDA. The result is summed and divided by the number of elements in the transform. Finally it is plotted as a measure of error for every transform size in the range. The formula used to calculate the error is shown in Equation 4.1

$$y_{err} = \frac{1}{N} \sum_{n=0}^{N-1} (x_{single}[n] - x_{double}[n])^2 \quad (4.1)$$

where N is the number of data values in the signal, x_{single} is the signal transformed in single-precision, and x_{double} is the signal transformed in double-precision.

The other test consisted of transforming all signals with Matlab's FFT to analyse the signals in the frequency domain. Let us consider the difference between the pure tone of the signal and the highest point of the noise introduced by finite precision. This can later be compared to the noise added by the ADQ1600 to the input signal.

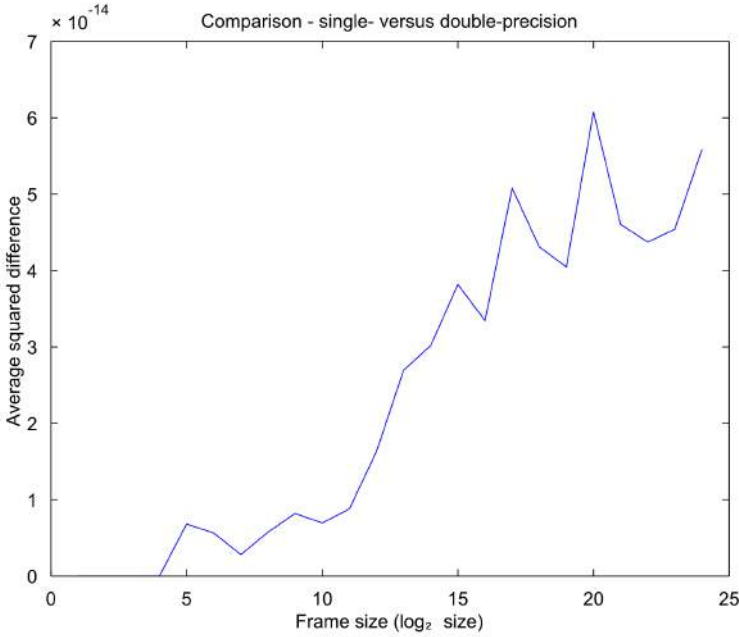


Figure 4.1: The average squared difference between transforms executed in single- and double-precision for transform sizes equal to powers of two.

Executions of an FFT and IFFT in sequence were made on a coherent sine in both single- and double-precision. The signal was created in C using the sine function in *math.h* and saved to a text file in double-precision together with the transformed signals from the GPU. Executions in CUDA were made in single- and double-precision with transform sizes ranging from 2^5 to 2^{24} . The purpose of using a coherent sine is to introduce as little noise as possible from other sources like windowing. At the same time we want to keep the spectral leakage to a minimum. As described in Section 2.3 windowing is used to reduce the spectral leakage.

The reason for doing several transform sizes is the idea that at some point the added operations for each stage in an FFT will start to have effect on the accuracy when using a single-precision representation. It is clear that

the accuracy does decrease with increasing powers of two but there is no definite size that displays a huge decrease in accuracy. This is an indicator that it might be possible to use single-precision for the intended application. In Figure 4.1 the error is plotted as a function of the FFT size.

It must however be compared to the input signal to see what information is being lost by using less bits to represent the result. For this the transform size of 2^{20} will be used considering that the highest difference lies there.

By using Matlab's FFT transform on the input signal and the transformed and inverse transformed single- and double-precision signals it is possible to reason about information being lost. In Figure 4.2 the amplitude spectrum of the single-precision signal is presented with the highest peak of the noise floor. Ideally all frequencies except the frequency of the input signal should be located in $-\infty$. This is clearly not the case and the highest peak in the noise show the maximum error compared to the expected result. Any information present below this point in a hypothetical signal, that is transformed with the algorithm, will be lost.

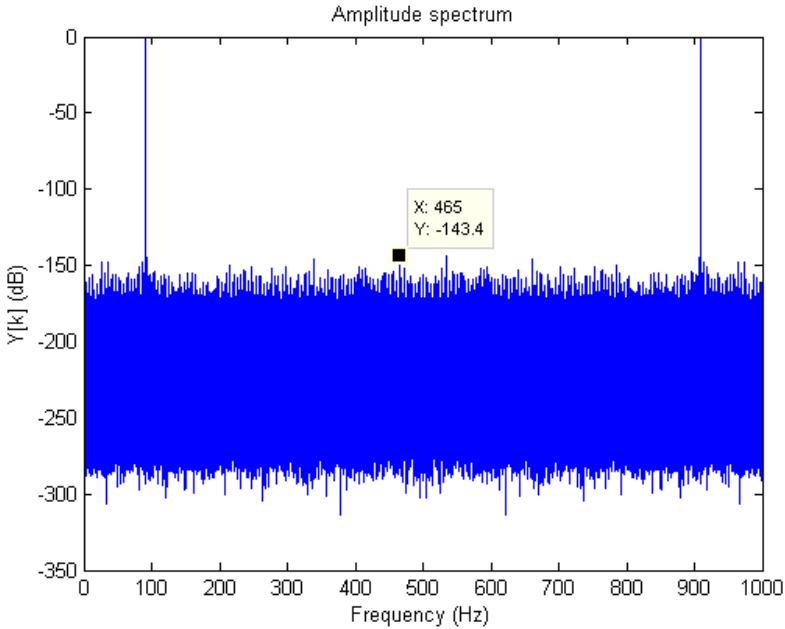


Figure 4.2: The amplitude spectrum in decibels of the transformed signal in single-precision. The highest peak in the noise floor is marked with its X- and Y-coordinate.

In Figure 4.3 the amplitude spectrum of the transformed signal using double-precision is presented. Matlab's transform of the original signal is very similar to the spectrum of the double-precision CUDA-transformed

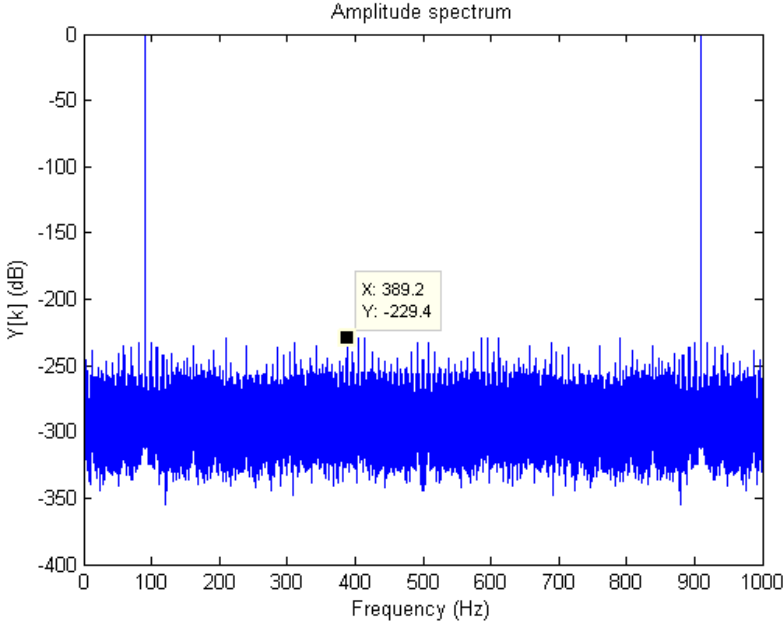


Figure 4.3: The amplitude spectrum in decibels of the transformed signal in double-precision. The highest peak in the noise floor is marked with its X- and Y-coordinate.

signal.

The maximum signal-to-noise ratio possible for a signal represented in 14-bits two-complement is $SNR_{max} = 20 \log_{10}(\text{dynamic range}) = 20 \log_{10} \frac{2^{14}-1}{1} \approx 78.3$ dB. Any information below -78.3 dB will be lost. In addition to the theoretical limit, the ADQ1600 also introduces noise to the signal and the practical limit may be even closer to zero than -78.3 dB. This means that a single-precision representation will work for FFT computations in the context of the intended application since the noise introduced by the algorithm is smaller than the noise introduced by the digitizer.

Graphics card

When predicting performance of the implementation several steps have been taken. By first making a rough model assuming that the FFT computation is the longest task to be performed in the algorithm an estimate of how many samples can be processed during a time frame was calculated. This gave an optimistic bound on the minimum performance required in GFLOP/s for an FFT calculation.

A low-end graphics card (GT 640) was used to test memory performance and polish the model created for performance. The core application includes

windowing, executing FFTs and creating an average of the results, which were all implemented in parts and measured for performance. By describing the execution times of the different parts of the program as a function of the FFT execution time the model was updated with measurement data. This gave better understanding of which computations take the most time. In the case of the GT 640 the execution time of the FFT compared to the other stages was approximately 50%. Also it was verified that the GFLOP/s for different transform sizes did not change much close to the intended goal. This indicates that depending on which card is chosen, it will either fail for all transforms or work for all transforms in close proximity to the goal size.

By using the model and replacing the execution times with the assumed GFLOP/s an estimate as to which GFLOP count will be necessary to run the application was extracted. To be able to match the speed of the digitizer approximately 130 GFLOP/s is necessary.

NVIDIA released a performance report [15] in January 2013 that discusses the performance for some of their library implementations. Among the results are the GFLOP/s for a K20X [17] card using a 1D complex transform for different sizes. The presentation indicates that using a K20X is more than enough to implement the application with a GFLOP/s rate of about 350 for the intended transform size. The K20X is a computation card from the NVIDIA Tesla family and one of their highest-performing cards at the moment. At the Department of Computer and Information Science, IDA, in Linköping University we were able to perform measurements on a K20c [17] which is very similar to the K20X. In comparison the K20X is a bit better on paper with more cores, higher memory bandwidth and more memory. In addition to the K20c, a m2050 computation card of the Fermi architecture was made available as well.

To understand what performance can be expected of the K20c a couple of measurements were made. The execution time of a host to GPU transfer was extracted as well as the execution time of the different kernels together: pre-FFT, FFT and post-FFT. Detailed information on the different kernels is presented in Section 4.3. Furthermore the execution time when overlapping data transfers and computations was extracted. Preliminary tests using the low-end graphics card had shown that performing computations on the GPU while doing memory transfers over the PCIe bus resulted in a performance decrease in both operations. The tests were executed on different transform sizes ranging from 2^{15} to 2^{23} . Every test result is based on an average of 300 executions. In Table 4.1 the measured bandwidths of the cards are presented, both with and without overlap. We are not sure why the memory bandwidths without overlap for small sizes are less than with overlap. For larger sizes the results tend to be more aligned to what is expected.

It is interesting to note that, while there is a decrease in throughput in the memory transfers, the difference between transfers with overlapping computations and without is much smaller when using computation cards. The difference on the GT 640 is significant where the overlapping transfers,

Transfer size ($\log_2(\text{size})$)	GT 640 (GB/s)	K20c (GB/s)	m2050 (GB/s)
Non-overlapping computations and memory transfers			
15	0,999	2,826	2,238
16	1,663	3,827	3,089
17	1,555	4,697	4,076
18	2,173	5,268	4,813
19	2,512	5,620	5,308
20	2,931	5,824	5,701
21	3,109	5,931	5,800
22	3,200	5,965	6,040
23	3,255	6,017	6,004
24	3,196	6,030	5,979
25	3,296	6,037	5,978
Overlapping computations and memory transfers			
15	2,088	3,383	2,639
16	2,425	4,316	3,617
17	2,642	4,953	4,992
18	2,850	5,452	5,452
19	2,805	5,632	5,771
20	2,873	5,852	5,933
21	2,911	5,910	5,997
22	2,921	5,980	5,991
23	2,927	5,997	5,996
24	2,951	6,006	5,988
25	2,944	6,016	5,992

Table 4.1: Memory bandwidth (GB/s) for different NVIDIA cards.

for the largest transfer, only attain $\sim 90\%$ of the bandwidth attained without overlaps. The computation cards should be able to transfer data at 6,4 GB/s; the reason for not attaining that speed is most likely because of the motherboard.

The number of threads per block will change the execution time of the different kernels. Three configurations were tested: 256, 512 and 1024 threads per block. Best performance overall was obtained when 256 threads per block were used. For this reason, if nothing else is stated, 256 threads per block will be used in the measurements from now on.

As for the measurements on execution time, with and without overlapping transfers and computations, the outcome is that there is most likely no difference. The deviation could be explained with the variance in different test executions.

When comparing the K20c performance versus the required execution time, measurements indicate that it will work for transform sizes of 2^{17}

and above. In Figure 4.4 the required execution time is plotted versus the execution time of the K20c.

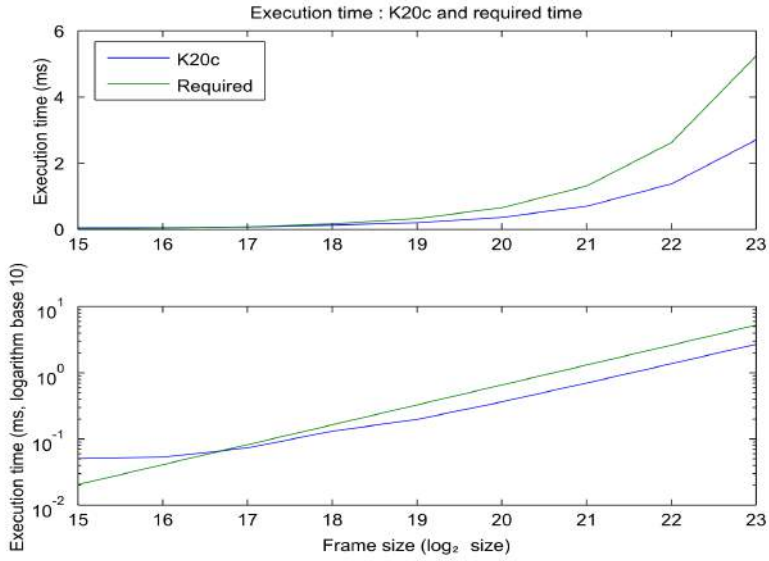


Figure 4.4: The execution time of the K20c computation card versus the required execution time.

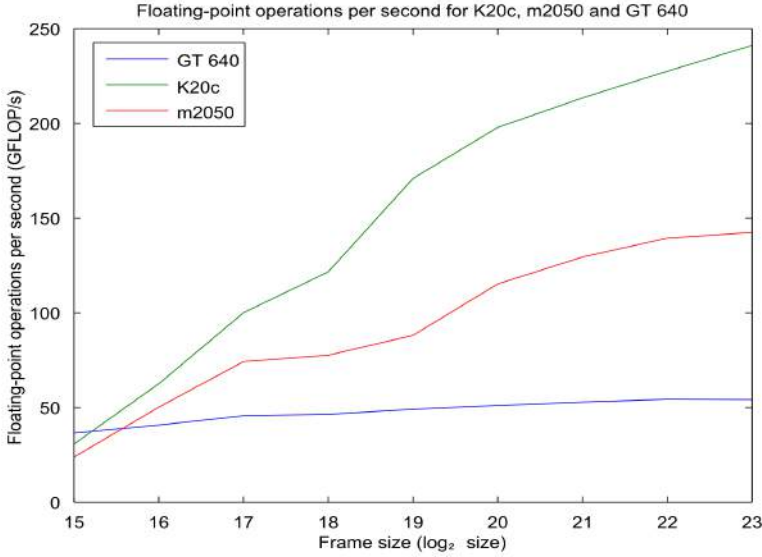


Figure 4.5: Floating-point operations per second for the FFT stage of the three cards.

The plot indicates that it will not be possible to execute transforms at the required speed for sizes below 2^{17} . This is quite unexpected when compared to the figures released by NVIDIA. Since the focus in this thesis is on large sizes and because of time limitations this issue has not been investigated further. Figure 4.5 shows the computed GFLOP/s of the FFT stage for each of the three cards. Both the execution time and GFLOP count indicate that a K20c card will work for the given application. It is interesting to note that while the execution time indicates that sizes of 2^{17} and above will work, the GFLOP/s indicates that only 2^{21} and above will work. The reason for this is that the execution time of the kernels are different between cards and it will be discussed in detail below.

There are gaming cards that use the same chip-set as the K20c card and have similar other features as well. Table 4.2 shows the comparison used when deciding what type of card to use for the final implementation.

The assumption is that these parameters will affect the performance of the application more than other architectural differences. Hopefully the results for one of these cards will be similar to the others used in the comparison. The theoretical maximum throughput is calculated directly from the definition of FLOP presented in Section 2.2.1. The results are presented with 3 significant digits.

The first three parameters have a direct link to the performance of any algorithm running on the GPU. And because of the problem sizes that are

Graphics card	Theoretical maximum throughput (GFLOP/s)	Memory bandwidth (GB/s)	L2 cache (MB)	Bi-directional data transfers
K20c	3500	208	1.5	Yes
GTX 780	3970	288	1.5	No
GTX 680	3090	192	0.5	No

Table 4.2: Comparison between three NVIDIA manufactured graphics cards.

expected to be executed in this case a three times bigger cache might have a big performance effect on both the computations on the cards as well as the data transfers on the PCIe bus. The bi-directional data transfer is only interesting if all processed data need to be transferred from GPU memory. For this application it was decided that transferring all processed data back to the host was not necessary. Instead data reduction is performed in the GPU by means of either skipping data or making the average count higher than the rate possible to send data back. It is actually the motherboard that sets the limitation in this case. There is a possibility that time-multiplexing would make it possible to transfer all results from the GPU with PCIe 3.0. Together with SP Devices a decision was made to use GTX 780 for the final implementation.

The GTX 780 was tested with the same tests as the other cards. Table 4.3 presents the memory bandwidth of the GTX 780. It is interesting to note that the GTX 780 experiences the same type of behaviour as the GT 640. The bandwidth is lower when overlapping transfers and computations and in this case the bandwidth with overlaps is $\approx 93\%$ of the bandwidth without overlap. We will present measurements below that indicates that there are even more differences between gaming cards and computation cards.

Figure 4.6 shows the execution time of the GTX 780 and the K20c compared to what is required by the digitizer. They are very close in performance. For reference the GFLOP/s for the GTX 780 versus K20c is presented as well in Figure 4.7.

One interesting observation is that, even though the execution time on GTX 780 and K20c are very much alike, the times spent in different kernels differs. Figure 4.8 shows the ratio between the pre-FFT stage and the FFT, and Figure 4.9 shows the ratio of the post-FFT stage and the FFT. By looking at the ratio of how much time is spent in the pre-FFT stage and the post-FFT stage two things can be concluded.

- For large frame sizes the ratio, between different stages of the algorithm, is almost the same.
- Computation cards spend much more of their time in percent in the FFT stage compared to the gaming cards.

Transfer size ($\log_2(size)$)	GTX 780 (GB/s)
Non-overlapping computations and memory transfers	
15	1,246
16	1,531
17	2,472
18	3,341
19	4,510
20	4,736
21	5,280
22	5,552
23	5,776
24	5,863
25	5,907
Overlapping computations and memory transfers	
15	3,395
16	4,170
17	4,875
18	5,281
19	5,160
20	5,363
21	5,448
22	5,488
23	5,504
24	5,493
25	5,524

Table 4.3: Memory bandwidth (GB/s) for GTX 780.

This is a very interesting observation. Without knowing how the underlying architecture looks like it is rather difficult to explain this behaviour. This also explains the discrepancy between execution time and GFLOP/s mentioned earlier for the K20c card. In the model used to predict necessary GFLOP/s for the FFT stage the ratio of the GT 640 was used. Since the ratio is much lower on the K20c, the model is unable to predict the correct GFLOP/s necessary. With the updated ratio it suggest that an implementation will work for ~ 90 GFLOP/s which is close to what has been measured. Since measurements on the card is necessary for extracting the correct ratio, the model is of little use when predicting performance in a new system.

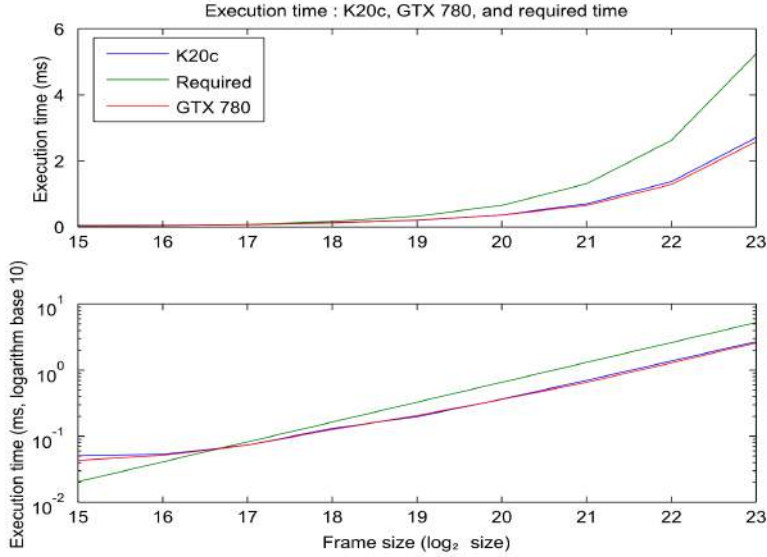


Figure 4.6: The execution time of the GTX 780 card and K20c computation card versus the required execution time.

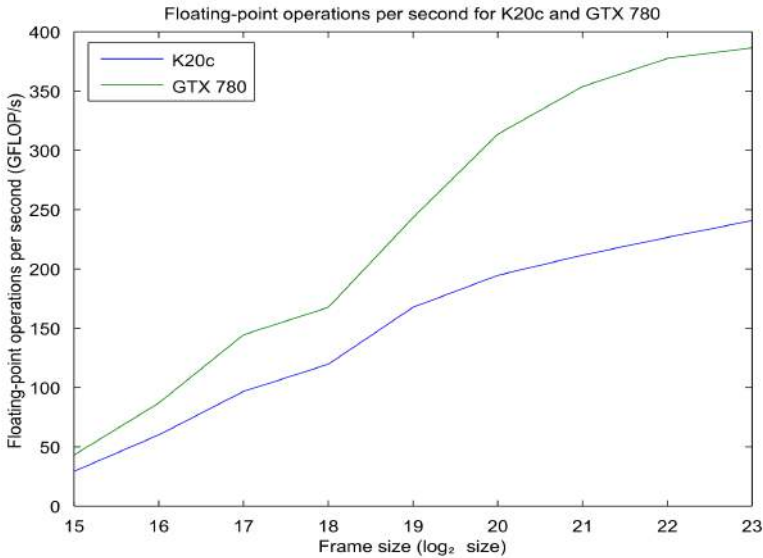


Figure 4.7: Floating-point operations per second for the FFT stage of the GTX 780 and K20c.

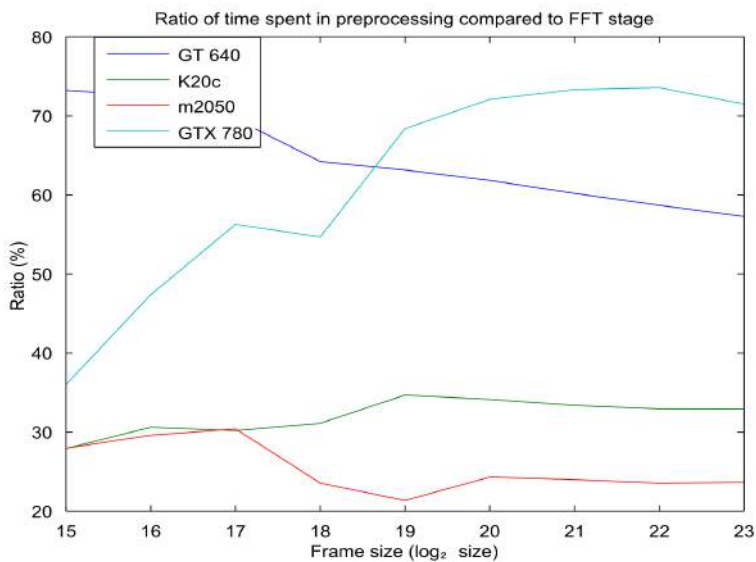


Figure 4.8: The ratio of time spent in the preprocessing stage compared to the FFT stage.

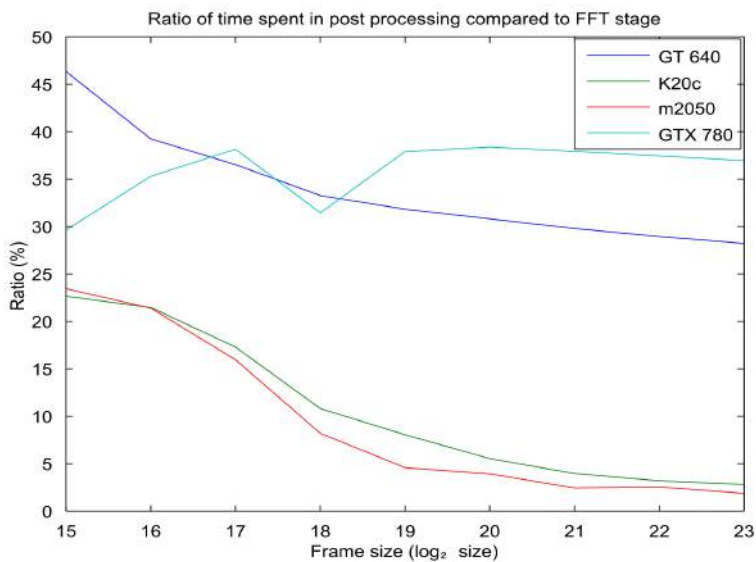


Figure 4.9: The ratio of time spent in the post-processing stage compared to the FFT stage.

4.2 Software architecture

This section describes the software architecture. It presents the high-level view of the implementation which is implemented with the C programming language. The only exception to this is the GPU kernels which are written in CUDA. However as mentioned in Section 3.1, CUDA is an extension to C and C++. Most of the files created in the project can be compiled with a normal C or C++ compiler. It is only files that contain GPU kernels that must be compiled with the NVIDIA compiler `nvcc`.

The overall structure of the program is initializing the environment and then start processing samples. Figure 4.10 shows the complete chain of events for the system. The control part of the system is executed on the

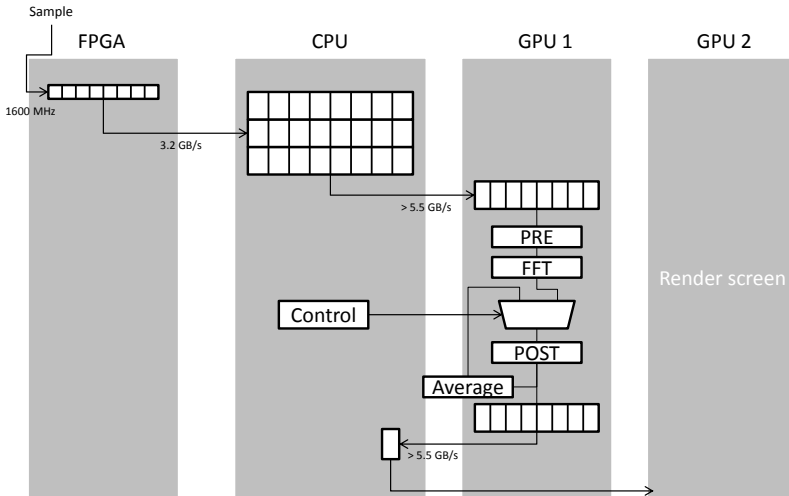


Figure 4.10: System overview.

CPU. Below the program flow is described in pseudo-code.

```

initialize(adq);
initialize(cuda);
initialize(local_variables);
while  $n\_sample < SAMPLES\_TO\_COLLECT$  do
    wait for new buffer;
    get new buffer;
    if overflow then
        break;
    end
    transfer buffer to GPU;
    if time to send data back then
        send last computed value back;
    end
    execute preprocessing;
    execute fft;
    wait on event record  $n\_sample - 1$ ;
    execute post-processing;
    record event  $n\_sample$ ;
    update local variables;
    change GPU stream;
end

```

Algorithm 2: Program flow.

There are a couple of software parameters that can be used to configure the application. The most important parameters are listed and explained below.

N_SEND Sets the size of how many samples each sample buffer should contain. This parameter also decides the size of the data transfers from host memory to GPU memory.

N_DRAM_BUFFERS Sets how many sample buffers to allocate in host memory.

N_TRANSFORM_SIZE Sets the size of the transform to be computed.

N_GPU_BUFFERS Sets how many buffers to allocate in the GPU.

N_SENDBACK Sets how often the event of sending data out of the GPU should occur.

N_AVG Sets how many samples should be averaged.

When arming the digitizer it will start to fill the buffers created in host memory. It will rely on the host operating system to provide new empty buffers. This happens when the application fetches new buffers to process and during operating system interrupt routines. It is essential that the

application empties these buffers faster than the digitizer is able to fill them. All of the commands sent to the GPU are of asynchronous nature. This means that the CPU will be in control most of the time fetching new sample buffers. The normal operation is that whenever a new buffer has been filled by the digitizer it will be sent off to the GPU for processing. All the asynchronous kernel launches are executed and local variables updated. Eventually the application will be stuck in a busy wait operation, waiting for the next sample buffer to be ready from the digitizer. The buffer allocation in host memory is clarified in Figure 4.11.

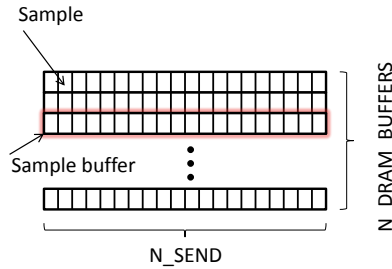


Figure 4.11: Buffer allocations in host memory.

The application is implemented with the concept of minimizing thread divergences in the GPU kernels as mentioned in Section 3.1.4. This is realized by executing control flow code on the CPU and throughput-oriented tasks on the GPU. Motivation for this can be found in NVIDIA Best Practices Guide in chapter Control Flow [15].

To be able to support transform sizes that deviate from the sample buffer size in host memory, the concept of batched execution, explained in Section 3.1.5, is extended to the whole implementation. The sample buffer size in host memory must be bigger than the size of the transform. When specifying a transform length that is less than the sample buffer size, the number of FFT executions done on the GPU is calculated with Equation 4.2

$$n_{batches} = \frac{s_{buffer}}{s_{fft}} \quad (4.2)$$

where $n_{batches}$ is the number of batches, s_{buffer} is the size of each sample buffer, and s_{fft} is the transform size. This will perform $n_{batches}$ number of FFTs executing in sequence on the entire memory area, as can be seen in the Figure 4.12.

An important feature of graphics cards is the possibility to execute memory transfers and at the same time make computations on the graphics card. It is described in Section 3.1. The application is relying on this feature.

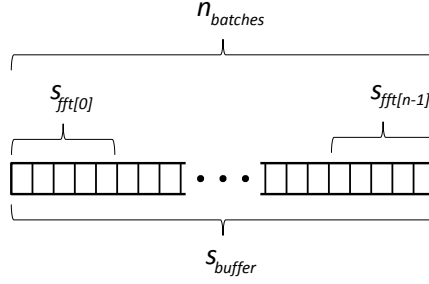


Figure 4.12: Batched execution model.

When the graphics card is using sample x as input for computations, the DMA engine on the graphics card is transferring sample $x + 1$ to the GPU. The normal behaviour of the application from the GPU's point of view is presented in Figure 4.13.

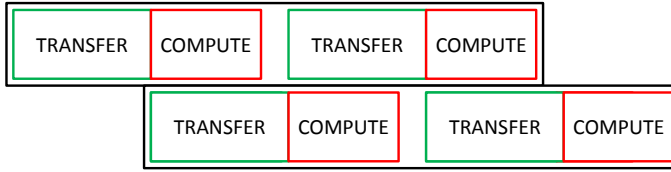


Figure 4.13: Overlapping data transfers and computations.

Because of the different formats used in the application there are buffers used to store different types of results in the GPU. In total there are four buffers, each with its own specific purpose. Figure 4.14 shows how the format is modified in the GPU and how data is exchanged between different buffers.

Data reduction is mentioned in Section 4.1.1. It is implemented with a parameter setting that specifies the rate to which data is sent back from the GPU to host RAM or another GPU. If the rate is set too high the GPU will not be able to handle the incoming computations and start to overflow the system. The rate is computed with Equation 4.3

$$n_{esb} = \left\lceil \frac{\max(n_{avg}, n_{sb})}{n_{batches}} \right\rceil \quad (4.3)$$

where n_{esb} is the effective send back rate, n_{avg} is the number of averages, n_{sb} is the send back rate specified, and $n_{batches}$ is the number of batches in

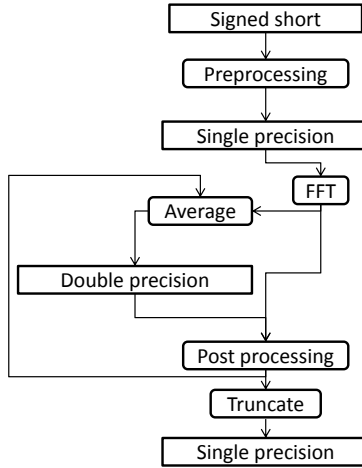


Figure 4.14: Different formats used in the GPU buffers.

each transfer.

The implementation is intended to ease switching between different devices from SP Devices. If the device supports streaming the only change necessary to the application code is the addition of code for initialization and destruction of the device registers. This is encapsulated in a C struct.

Last, the implementation is limited to a set of parameter configurations that have been tested and verified to work. The program will work when transfer sizes and transform sizes are powers of two. There are numerous reasons for this, including lower complexity, the fact that cuFFT achieves maximum performance for these sizes, and that the application was only required to handle such input. Moreover the transform size must be less than or equal to the send size. It was not required or requested to be able to do larger transform sizes than the send size.

4.3 GPU implementation

This section describes parts of the software implementation that are essential for a functional system. It describes how they are implemented and provides a rationale for different design decisions. Finally some general optimizations done in kernels are described and discussed.

4.3.1 Preprocessing

The preprocessing stage of the algorithm contains formatting of the data input conforming to the data layout required by the FFT. Additionally the window function is applied in this stage.

Formatting data input

This is the most simple function executing on the GPU. It is however essential for a working system and needs to be mentioned. The digitizer provides data samples with 16-bit width. This format is unsupported in the GPU except for storage. The formatting kernel is implemented in the GPU and each thread will sign-extend the 16-bit data input, first to a C type integer. After that, the integer is cast into a single-precision, floating-point number representation. Finally it is divided by 2^{15} to place it in the interval of $[-1, 1[$. The division is a combination of 2^{13} , to compensate for the 14-bit two-complement format used in the digitizer, and 2^2 , to compensate for the most significant bit alignment discussed in Section 3.6.3. Because of the non-uniformly spaced floating-point number representation described in Section 2.4, results will be more accurate by making the input signal range between -1 and 1. It is also natural that the maximum magnitude of the input signal is 1.

Windowing

Windowing is mainly implemented on the host system. The only part of windowing that is implemented in the GPU is the applying of the window to the input signal. This is done by loading the window from a predefined area and multiplying it with the input signal. Windowing on the GPU is done in the same kernel as formatting of the input described above.

When starting the application the window must be decided. It is computed on the host and finally placed in the global memory of the GPU. The reason for this implementation is that computing the window while processing samples is too compute expensive. It would make the computation longer for the kernel executed before the FFT execution. But also the type of window function would change the execution time for the kernel. This is unacceptable. With the current implementation the execution time is the same for all windows. This property makes reasoning about the performance of the system for different configurations in terms of different windows much easier. On a general note this property is sought after in all GPU kernels.

When it comes to batched execution this kernel does not change at all. The window loaded during the execution consists of smaller windows; with the window count equal to the number of batches.

To add a new window to the application the window function must be created with input parameters: memory location, transform size and number of batches. By adding the memory address of this function to the existing

array of window function pointers it can be accessed and used within the application.

When using windows the frequency spectrum is distorted. The compensation for this is handled in the post-processing stage.

4.3.2 Fast Fourier transform

This function is implemented by NVIDIA. The plan for executing the kernel is created at the start of the program. It will always be an in-place real to complex transform with native padding. This means that for both batched and non-batched execution the inputs are placed in a linear fashion in memory with a distance of transform size + 1 between each input to the transform. This will result in correct adjacent alignment for the output of the transform. The layout of the data used in cuFFT has been described in more detail in Section 3.1.5.

4.3.3 Post processing

This stage differs depending on what configuration is used. If there is no averaging the amplitude spectrum will be the output of this stage. If averaging is performed it will be the RMS spectrum that is the output. They are both modified and stored as values in decibel according to Equation 4.4.

$$Y_{dB}[k] = 20 \log_{10} X[k] \quad (4.4)$$

The difference in functionality will be described below. First we consider the case of no average, then with averaging enabled.

No average

For a signal with no windowing the spectrum is normalized and turned into the single-sided amplitude spectrum by Equation 4.5

$$Y[k] = \begin{cases} \frac{2|X[k]|}{N} & k \in [1, \frac{N}{2} + 1] \\ \frac{|X[k]|}{N} & k = 0 \end{cases} \quad (4.5)$$

where $Y[k]$ is the amplitude spectrum, $X[k]$ is the frequency spectrum, and N is the transform size.

If windowing is used then the spectrum is given by Equation 4.6

$$Y[k] = \begin{cases} \frac{2|X[k]|}{NW_{cg}} & k \in [1, \frac{N}{2} + 1] \\ \frac{|X[k]|}{NW_{cg}} & k = 0 \end{cases} \quad (4.6)$$

where W_{cg} is the normalized coherent gain discussed in Section 2.3.

Averaging

The averaging implemented in this thesis is called RMS averaging. It is performed in the frequency domain on the squared magnitude of the frequency spectrum. A detailed description on the mathematics behind the averaging process can be found in chapter 9 in [4]. It is the RMS spectrum that is the output of this part of the system.

The theoretical expression used to calculate the average of a fixed amount of spectrums is presented in Definition 4.1

Definition 4.1 *RMS spectrum*

$$x_{rms,avg}[k] = \sqrt{\frac{S}{M} \sum_{n=0}^{M-1} |X_n[k]|^2} \quad k \in [0, N-1] \quad (4.7)$$

where S is a scaling factor to account for windowing and transform size N , M is the number of averages and $X_n[k]$ is the n th frequency spectrum.

Finally to account for negative frequencies the implemented expression is derived in Equation 4.8

$$x_{rms,avg}[k] = \begin{cases} \sqrt{\frac{2}{M} \sum_{n=0}^{M-1} \left| \frac{X_n[k]}{NW_{cg}} \right|^2} & k \in [1, \frac{N}{2} + 1] \\ \sqrt{\frac{1}{M} \sum_{n=0}^{M-1} \left| \frac{X_n[k]}{NW_{cg}} \right|^2} & k = 0 \end{cases} \quad (4.8)$$

where W_{cg} is the normalized coherent gain.

On a practical level the average filter is implemented as a sequence of summations in time with the final step dividing the sum with the number of summations performed. For reasons explained in Section 2.4 the summations done in this part of the system are accumulated in a buffer with the double-precision format. Eventually when the result is saved it will be truncated into single-precision format. By doing this numerical errors from multiple summations with single-precision are avoided.

There are three different types of kernels used in the averaging part. One is used as a starting kernel which will overwrite the results at the buffer location. One will sum at the buffer location and the last one will use the buffer location, divide the results, and perform final processing. The control flow used to execute these kernels is performed in the host.

Averaging works slightly different than the other parts of the system when it comes to batched execution. For every batch there will be a new kernel launch on the GPU. It is also possible to implement this as one single kernel making all decisions on the GPU. There are pros and cons for both approaches. Making a new kernel for each batch produces less complex kernels. It also provides more determinism in kernel execution time as there are less control flow statements in kernel code. This simplified measurements when deciding on a graphics card. On the other hand it is necessary to create three similar kernels instead of one. The biggest problem however with

implementing only one kernel is that the averaging part needs synchronization between the summations and the divisions. The synchronization needs to be between blocks and that is not supported in the CUDA API. There are ways to implement it using the `__threadfence()` function, atomic operations, and the use of global memory. However, this approach was considered too risky, and because of other priorities the approach using one kernel was not implemented.

When implementing averaging as multiple kernels much of the synchronization issues are dealt with in the ordering of the kernel launches. Because executing kernels in the same CUDA stream guarantees sequential operation for operations sent to that stream, any averaging operation sent to the same stream will be implicitly synchronized. There is however a need to synchronize between streams. It can be implemented with events in CUDA. Basically every stream has its own event used to signal to other streams when it is ready with its averaging kernel part. All other streams must wait for that stream to complete. There is also a total ordering of when the operations will be executed and it is controlled by the CPU. It may be necessary to synchronize the GPU at regular intervals when using events. When plotting is not enabled the program stack will eventually overflow without device synchronization. By using `cudaDeviceSynchronize()` some performance is lost, however the synchronization that is necessary is not very frequent.

4.3.4 Kernel optimizations

During the implementation some micro-level optimizations were done to further decrease the execution time of single kernels. CUDA allows for fine-grained optimization on instruction level by giving the option of using intrinsic functions to the programmer. Intrinsic functions are special versions of normal mathematical functions such that they are mapped to fewer hardware instructions compared to the normal functions. However the precision is greatly reduced for some of the intrinsic functions so care must be taken when using them [16].

Because of the hardware architecture, multiplication is less time consuming than division. With this fact in mind, variables stored in local memory and function parameters used for division inside kernels are stored as the reciprocal of the denominator. The calculation is done in the CPU and the GPU is able to use multiplication instead of division.

For the application at hand, the decision of which optimizations to make is most of the time a trade-off between precision and performance. By using hardware functions for complex mathematical functions some precision is lost. One optimization that as far as we know gives a performance improvement without missing out on precision is dividing larger mathematical functions into smaller and less complex functions. This became evident when reducing the absolute operation for a complex number into smaller

and less complex mathematical operations. More detailed information on the subject can be found in the NVIDIA Best Practices Guide in the chapter on instruction optimizations [15].

4.4 Visualization

The visualization part of the program is implemented with the ArrayFire [7] GPGPU library which is used for plotting the resulting spectrum. It will however skip plotting if too much work is done on the GPU. We have not found more documentation on this issue. The rendering card is not extremely utilized and should be able to plot most of the time though.

4.5 Test set-up

The test environment consists of two parts. The high-level Matlab model of the system and the test-specific code integrated into the application source code. While developing the application, tests were gradually implemented and functionality verified. When one part of the system was completed the functionality for testing and verifying its correctness was functional as well.

First an implementation with the digital signal processor ADQDSP described in Section 3.6.3 was created. Using ADQDSP instead of a streaming device like ADQ1600 provides numerous advantages. It is easy to verify the PCIe transfer with simple mathematics depending on how much data we decide to collect and measure the time it takes. It is not possible to lose samples as data must be queried by the host system. Because it is programmed with a predefined pattern it is easy to verify that the GPU implementation does not lose data. It provides a stable system that can be used for functional testing.

Step one was to implement the streaming. There are two tests implemented to verify its correctness. One is implemented in the GPU that calculates and evaluates the data in GPU memory. The second test uses two storage buffers in host memory. The first one is used to hold the data from the ADQDSP. The other buffer holds data from the GPU. When a buffer has been transferred to host RAM it is placed in one of the buffers and also sent to the GPU. When all buffers are filled in the GPU they are flushed to the second buffer in host RAM. When all data has been transferred the two buffers are compared for equality.

The second step was to implement the actual computations and integrate the test environment with the algorithm. By supplying the `'-debug'` parameter to the application it is possible to decide parameters such as buffer size, transform size, type of window, and how many averages to compute. The application writes the output of the system to text files. By modifying the functional model created in Matlab it is possible to launch the application with specified parameters and collect the results from the text files when it

is done. The output of the system is compared to computations made in Matlab and the result is two comparisons. One is the average difference for the transform and the other is the maximum difference for any bin in the transform.

Definition 4.2 *Difference vector*

$$A_i = |x_i - y_i|, i \in [0, \frac{N}{2} + 1] \quad (4.9)$$

The vector x is the result from the computation on the GPU, the vector y is computed in Matlab on the same signal, and N is the transform size. The average difference is computed with Equation 4.10

$$C_{avg} = \frac{\sum_{i=0}^{\frac{N}{2}+1} A_i}{N} \quad (4.10)$$

and the maximum difference is computed with Equation 4.11.

$$C_{max} = \max(A_i) \quad (4.11)$$

Verifying that the correct RMS values have been computed is done by using the definition of RMS defined in Definition 4.3 and Parseval's relation [32] defined in Definition 4.4

Definition 4.3 *Root mean square*

$$x_{rms} = \sqrt{\frac{x_1^2 + x_2^2 \dots + x_N^2}{N}} \quad (4.12)$$

Definition 4.4 *Parseval's relation*

$$\sum_{n=0}^{N-1} x[n]^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2 \quad (4.13)$$

where N is the number of values used.

By combining them it into Equation 4.14 it is possible to calculate the RMS value for a signal in both time and frequency domain.

$$\sqrt{\frac{1}{N} \sum_{n=0}^{N-1} x[n]^2} = \sqrt{\frac{1}{N^2} \sum_{k=0}^{N-1} |X[k]|^2} \quad (4.14)$$

By using this property the RMS value of the signal is calculated in the time domain and compared towards the RMS value computed from the RMS spectrum computed in the application. The value calculated from the signal is matched towards Equation 4.15

$$x_{rms} = \sqrt{\frac{\sum_{k=0}^{\frac{N}{2}+1} x_{rms,avg}[k]^2}{W_{enbw}}} \quad (4.15)$$

where N is the signal length, $x_{rms,avg}$ is the averaged RMS spectrum, and W_{enbw} is the normalized ENBW that is used to correct the power spectrum [4, p.234].

It is possible to change the format and source of the input signal to the system. Possible configurations from devices from SP Devices are: short signed integer and unsigned integer. Synthetic signals created in C are in single-precision, floating-point format or short signed integer. This must be changed at compile time but adds flexibility to test different configurations towards the Matlab model.

Chapter 5

Results and evaluation

This chapter contains the results of testing the system. The functional tests and performance-related measurements are presented here.

5.1 Functional testing

The functional tests are mainly executed in a controlled environment without the real-time constraints. Especially when comparing the precision of the system with that of the Matlab model, it is not possible to keep the speed required by the system and at the same time evaluate the correctness of the implementation. Because of this, most tests are executed on synthetic signals created in the implementation. In addition to the tests performed on synthetic signals the ADQDSP is used to provide signals to the system. All measurements are done on signed short input signals.

First functional tests are performed to verify correct functionality. This means that the various parts of the system satisfy the intended function, which when supplied with an input signal, delivers a modified output that is: correctly aligned in memory space and modified by windowing and averaging. This is verified by supplying a constant DC voltage to the system. There are a number of useful properties of the FFT when supplied with a constant DC input in this context but the most important part is the easily interpreted output. It is simply the DC frequency bin or bin zero that has the same value as the constant input. This can be used to verify correct alignment, both with and without batches. When averaging it will be the same output as without averaging. All of these results will be bit exact as well. It is only when windowing is tested with this input that the results will be different between the Matlab implementation and the GPU implementation. It is however expected that the difference is the same on every transform made. It is the average difference presented in Table 5.1.

To verify that subsets of the input corresponds to the correct subsets of the output a test case using the ADQDSP is performed. The signal

generated in the ADQDSP is a counter of 32 bits. The high part and the low part will be treated separately resulting in a wave form that alternates between zero and a 16-bit counter.

The consistent properties of the functional test cases are: 10 sample buffers, the size of the sample buffers which is 2^{24} , and the amount of buffers allocated on the GPU which is limited to 8. This means that 10 transfers of 2^{24} samples each are transferred to the GPU and then verified for correctness. The results are presented in Table 5.1.

Transform size	Input	Average	Window	Expected output	Output
2^{15}	-1.0	0	No	No difference	No difference
2^{16}	-1.0	0	No	No difference	No difference
2^{17}	-1.0	0	No	No difference	No difference
2^{18}	-1.0	0	No	No difference	No difference
2^{19}	-1.0	0	No	No difference	No difference
2^{20}	-1.0	0	No	No difference	No difference
2^{21}	-1.0	0	No	No difference	No difference
2^{21}	-1.0	9	No	No difference	No difference
2^{21}	-1.0	0	Hann	Difference	$2,245 \cdot 10^{-12}$
2^{21}	ADQ-DSP	0	Hann	Difference	$4,688 \cdot 10^{-10}$

Table 5.1: Functional tests.

To verify that the system outputs reasonable results when processing signals of different types a couple of signals have been verified against the model. The consistent properties of these test cases are: sample buffer and transform length of size 2^{21} , Hann window is used for all signals but coherent sine and there is no averaging. Signal information describes the signal as A = amplitude and T = period. The results are presented in Table 5.2.

Signal	Signal information	Average difference	Maximum difference
Coherent sine	A = 1, T = 11.1134	$5,531 \cdot 10^{-11}$	$5,390 \cdot 10^{-08}$
Sine	A = 1, T = 500	$1,232 \cdot 10^{-11}$	$7,301 \cdot 10^{-08}$
Double sine	A = 0.5, T ₁ = 500, T ₂ = 250	$3,856 \cdot 10^{-10}$	$7,351 \cdot 10^{-08}$
Triangle wave	A = 1, T = 500	$2,654 \cdot 10^{-11}$	$4,588 \cdot 10^{-08}$
Square wave	A = 1, T = 500	$2,505 \cdot 10^{-09}$	$7,035 \cdot 10^{-08}$
Sawtooth wave	A = 1, T = 500	$1,679 \cdot 10^{-10}$	$3,262 \cdot 10^{-08}$
ADQDSP	-	$4,688 \cdot 10^{-10}$	$3,199 \cdot 10^{-08}$

Table 5.2: Precision tests.

For the same signals averages of 30 have been performed. The result is presented in Table 5.3. Furthermore, the RMS value of the input signals have been compared to the RMS value extracted from the spectrum. They match each other in at least 3 significant digits.

Signal	Signal information	Average difference	Maximum difference
Coherent sine	$A = 1, T = 11.1134$	$5,077 \cdot 10^{-11}$	$3,569 \cdot 10^{-08}$
Sine	$A = 1, T = 500$	$9,999 \cdot 10^{-12}$	$7,315 \cdot 10^{-08}$
Double sine	$A = 0.5, T_1 = 500, T_2 = 250$	$2,830 \cdot 10^{-10}$	$4,894 \cdot 10^{-08}$
Triangle wave	$A = 1, T = 500$	$3,064 \cdot 10^{-11}$	$5,264 \cdot 10^{-08}$
Square wave	$A = 1, T = 500$	$2,504 \cdot 10^{-09}$	$7,517 \cdot 10^{-08}$
Sawtooth wave	$A = 1, T = 500$	$1,488 \cdot 10^{-10}$	$2,770 \cdot 10^{-08}$
ADQDSP	-	$4,734 \cdot 10^{-10}$	$1,615 \cdot 10^{-08}$

Table 5.3: Precision tests on averaged samples.

5.2 Performance

This section describes the performance that is achieved by the system. It introduces some requirements and limitations, and presents performance results.

5.2.1 Considerations and limitations

There are a couple of things that limit the stability and performance of the system. The gravity of the problem and the difficulty of resolving it differs depending on what parameters are in focus. The system relies on the buffering of data in several stages. Whenever this link fails the system will fail. There are two buffers in the system subject to overflow. One resides in the digitizer and the other resides in host RAM. The buffer in the digitizer is very small and can only hold a small amount of sampled data. The buffer in host RAM can be modified by the user but is restricted to some constraints:

- The digitizer can only address up to 32 buffers until it needs to update its target registers that point into host RAM.
- There must be enough buffers in host RAM so that the GPU does not overflow, enough in this context translates into a number making the GPU capable of handling unforeseen delays in the operating system and GPU.

This essentially limits the different configuration options of the system to a specific set of configurations. The digitizer will overflow if the buffers are not large enough, even if streaming is the only task executed in the system. This is because the size and amount of buffers available to the digitizer decides the maximum response time it can handle before overflowing. The digitizer will overflow when Inequality 5.1 holds

$$t_{os} > \frac{\min(n_b, 32) * s_b}{d_{bw}} + t_{buf} \quad (5.1)$$

where t_{os} is the maximum response time of the host operating system, n_b is the number of buffers created in host RAM, s_b is the size of the buffers, d_{bw} is the digitizer bandwidth, and t_{buf} is the time it takes for the buffer in the digitizer to overflow, which is 45 μs . There are two events that are used to provide the digitizer with new buffers to fill. One of the events is when the application fetches a new buffer for processing and the other is during an interrupt routine of the operating system. What the expression states is that, when the time it takes for the digitizer to fill all of its available buffers is exceeded by the time it takes for the operating system to update the register of available buffers in the digitizer, the device will overflow. It is easy to see that increasing the buffer size and amount of buffers up until 32 will lead to a more interrupt tolerant system. However at some point the system will not be able to provide a sufficient amount of buffers of contiguous memory at the size specified by the application.

When increasing the size of the buffers and by doing this, effectively decreasing the amount of available buffers, the requirements on the GPU are increasing. The GPU can overflow in the trivial case that there is too much computation time with respect to the data transfer rate. It is also possible that the GPU will overflow if the response time of the operating system or response time of the GPU itself exceeds the time it takes for the digitizer to fill the buffers in host RAM. In the implementation created in this thesis the GPU overflows can be handled by tuning the parameter used for sending back data to the host RAM. If the application tries to send back data too fast it will eventually overflow. By using an operating system that is more real-time oriented than Windows the problems mentioned above are likely to be reduced.

In addition to the issues described above there are hardware constraints present. The speed of the host RAM is a critical parameter of the total system. Because of the high load placed on system RAM by both the digitizer and the GPU it is a parameter that most likely has a large effect on the stability of the application. This issue will be discussed more in Section 5.2.2.

Another hardware-related issue regards the digitizer's PCIe interface. The digitizer's maximum streaming rate is the same as the theoretical maximum bandwidth of the PCIe bus. This means that whenever there is a slight delay, it is a delay that is not possible to recover from. This delay will

eventually overflow the memory buffer in the digitizer. This is the reason for using a sampling rate of 1500 MHz on the digitizer which translates into 3.0 GB/s of data. It may be possible to reach a transfer rate of 3.2 GB/s by modifying the packet size of the PCIe packets. It will give a higher theoretical bandwidth as the header will take less space compared to the payload. This has not been verified however, as it is not supported on the system available.

When it comes to the PCIe set-up the digitizer must be placed on a separate PCIe switch. When used in the same switch as one of the graphics cards the stability of the system decreased drastically. This means that samples between the digitizer and host RAM were lost continuously with several samples being lost every minute.

5.2.2 Measurements

In this section the stability- and performance-measurements will be summarized. This section will also describe the configurations that have been used in stability testing as well as show how long the tests have been executed.

On rare occasions the system will lose samples. The reason for this is, as far as we understand, the congestion of the PCIe link coupled with the high load on host RAM. Samples are being lost even though the only part of the system activated is the streaming of data from the digitizer and the fetching and writing of data by the GPU. The observation when this happens is that the digitizer is experiencing overflow in the internal buffer and that it has several DMA transfers queued. For some reason the data is not transferred. Possible improvements that may lead to a stable system are suggested in Section 7.2. This involves upgrading the computer system to support new features and using high-performance hardware.

All of the configurations use a sample buffer size of $2^{25} \approx 33$ MB that corresponds to 2^{24} samples in each buffer. This is to minimize the risk of getting overflows in the digitizer. This however limits the amount of sample buffers that can be allocated. The tests will be performed on the maximum amount of buffers that can be allocated but it is not guaranteed to be the same for all test. The reason for this is that buffers are allocated at operating system start-up and the number of buffers possible to allocate differs from one start to the next. They are allocated with a tool developed by SP Devices. It would not be possible to allocate the same buffer size if the system has been running for a while since the RAM experience fragmentation during runtime. How many buffers that are used is specified in the test case and the test ends when an overflow is detected. Table 5.4 shows the stability tests that have been done.

It seems that when only doing read operations from the host RAM, the system experience a more stable behaviour than when transferring data back. All tests that overflow transfers parts of the computations back to the host RAM. This suggests that the loss of samples is occurring when both

Type of test	Transform size	Time (Hours)	Nr. buffers
CPU→GPU, Plot	2^{21}	13	18
CPU→GPU, GPU→CPU	2^{21}	12	26
CPU→GPU, GPU→CPU (no computations)	2^{21}	14	19
CPU→GPU, GPU→CPU (no computations)	2^{21}	10	19
CPU→GPU (no computations)	2^{21}	52 (no overflow)	9

Table 5.4: Stability tests.

the digitizer and the GPU is trying to write data to host memory.

In terms of performance the system can either be overloaded or manage the data stream. If the system is able to handle the data stream the buffers in host RAM will not overflow. This can be tested without running long tests. It has been tested for sizes ranging from 2^{15} to 2^{24} and the results are listed in Table 5.5. All tests are executed with 14 buffers for 2 minutes with averaging of 30 samples and the same send back ratio. Plotting is enabled.

Transform size	Overflow
2^{15}	Yes
2^{16}	Yes
2^{17}	No
2^{18}	No
2^{19}	No
2^{20}	No
2^{21}	No
2^{22}	No
2^{23}	No
2^{24}	Yes

Table 5.5: Performance tests.

The results correspond well to the results presented in the feasibility study. The only difference is that the system overflows for 2^{24} . This is because the plotting takes more time for each power of two while no other parameters change.

Chapter 6

Related work

Looking for related work in the topic of performing FFT computations on a GPU is not hard. The FFT is invaluable in digital signal processing, among other fields, and scientists and engineers are constantly trying to find ways to improve its performance on different platforms. In a paper [41] by D. Takahashi, experiments are performed on large one-dimensional transforms in GPU clusters. It is stated that FFT computations perform well on single graphics cards but because of the large number of memory accesses it poses more of a problem to get a fast solution in a cluster environment. The focus in the paper is to minimize the PCIe bus transfers because of the large memory bandwidth required by the FFT. For sizes close to the transform size used in this thesis the results are not very impressive and not even close to the GFLOP/s measured on a single GPU. However for a size of 2^{34} the measured operation count is ~ 764 GFLOP/s.

Another interesting project is described in a paper [37] by S. Qi et al that concerns the precision in the FFT. Parts of the computations in the FFT algorithm is computed on the CPU in double-precision and then transferred to the GPU. More precisely the twiddle factors are computed in double-precision as the iterative computation of them increases the accumulative error. When calculated, they are sent to the GPU for use in the appropriate butterfly operation. In our project some of these ideas have been implemented as well. Some offline computations are computed in double-precision on the CPU before transferring them to the GPU. Also the summation part in the averaging is implemented in double-precision instead of single-precision.

In 2010 in a paper [34] by Z. Lili et al. the performance of asynchronous data transfers in conjunction with their own implemented FFT algorithm is evaluated. By removing the copy time from the total execution time, the execution time of a transform size of 2^{20} is close to 30 ms. The measured execution time for the cuFFT algorithm on GTX 780 is around 0.17 ms. This is a speed up of about 170 times. Of course this depends on the imple-

mentation of the algorithm but it is likely that a big part of the difference comes from improvements in the architecture. Furthermore cuFFT is free for anyone to use who purchased a graphics card.

Another interesting aspect is the real-time constraints that comes with real-time streaming implemented in this thesis. The system is required to handle limited real-time constraints which in the general case is rather complicated in a GPGPU environment. Betts et al. brought forth this issue in [2]. There are several problems with real-time scheduling and GPGPU, and the paper argues that for the moment GPUs are only viable in firm, soft or probabilistic real-time systems. One reason for this is that GPU manufacturers do not give full disclosure to their products. As an example, pipeline depth and how threads are scheduled on a NVIDIA GPU remains a secret. This is vital to any static analysis. Also there are problems with branch divergence and the parallelism inherent to any GPU. Existing WCET analyses assume that a task is run with a single thread of control. This does not hold for a GPU. The paper [2] extends a hybrid technique used to estimate WCET from sequential code to be used on NVIDIA hardware. Two models are used to model concurrency; one based on measurements alone and one partly based on measurements and partly on calculations. The results show that the model based on measurements alone is much more accurate. This leads the authors of the paper to believe that how concurrency is integrated into the model largely affects the accuracy of the result.

Chapter 7

Conclusion and future work

This chapter concludes the thesis. First the results and achievements are discussed and wrapped up. The next section presents some interesting directions for extensions of the thesis topic.

7.1 Conclusion

A system has been created connecting a digitizer to a graphics card performing spectrum calculations and averaging. In addition basic plotting has been connected to verify and view calculations in real-time. Furthermore a test environment has been developed to verify correct behaviour and examine differences in precision.

At a rate of 3.0 GB/s and a transform size of 2^{21} the GPU performs about 715 FFT computations per second along with other operations like format change, windowing, averaging and post processing of the complex Fourier coefficients. The rate on the send and update visualisation may be changed but it is proven to work at an update rate of ~ 24 Hz.

The streaming of data has been implemented by means of buffering the data in host RAM. This approach was chosen because a peer-to-peer implementation was deemed too risky and/or time consuming. Investigations regarding this issue are presented in Section 4.1.1. With the current system, this implementation is however subject to loss of samples in rare occurrences which is described in Section 5.2.2.

All operations except the summation part in the averaging is executed in single-precision, floating-point format. By comparing the results from transforms executed in single-precision and double-precision it was concluded that single-precision is accurate enough in the application context. Because the averaging part is susceptible to numbers larger than the input of the trans-

form stage the summations are executed in double-precision.

The application is implemented with the CPU executing the control flow of the program. Tasks that are suitable for parallel processing is sent to the GPU for processing. Executions on the GPU follow a certain pattern each time frame. First a preprocessing stage that contains format change and windowing. Then the NVIDIA cuFFT Fast Fourier Transform is used to switch between time domain and frequency domain. After that a post-processing stage is introduced that differs depending if averaging is performed or not. Each average operation is a different kernel launch for synchronization reasons.

Some guidelines on using the application are as follow:

- Use buffers in host memory of at least 16 million samples and try to allocate at least 32 sample buffers.
- Make sure that RAM memories support the concurrent operations from streaming and GPU transfers.
- Most operations done on the computer will access RAM, keeping the accesses to a minimum while streaming will minimize the risk of the digitizer overflowing. This includes other programs as well.
- Reduce data transfers going from the GPU to the host RAM.

On a more general note, measurements have been performed on different graphics cards with the same code executed as the one implemented in the application. Measurements showed that even though cards have the same chip-set architecture, there might be differences in execution time between parts of an application. The tests indicate that there are differences between how gaming cards execute CUDA kernels compared to computation cards as well as how memory transfers are handled.

7.2 Future work

There are of course a lot of aspects of our work that can be improved and extended. Also there are different directions to take depending on what is desired.

As discussed in Section 5.2.2 the system eventually overflows. However, it seems to be a very rare occurrence that makes this happen. By using a system with features dedicated to this type of solution these effects may be suppressed. This new system should definitely include the following features:

- The possibility to change the payload of the Transaction Layer Packet (TLP) of the PCIe bus - *This may make it possible to stream data at 3.2 GB/s.*
- LGA-2011 socket - *Provides more PCIe lanes in the CPU for additional concurrency in the PCIe bus.*

- Support for PCIe 3.0 x16 lanes - *This will reduce the time the graphics card is using the RAM memories.*
- RAM with quad channel support and high clock frequency coupled with low latency - *This will increase the performance of RAM memories during runtime.*

By rearranging how work is executed both in the host program and in the GPU kernels there will most likely be performance gains. Also by implementing visualization directly in OpenGL it will most likely be possible to increase the rate at which the FFTs are visualized.

There are definitely more investigations required to make sure that the most stable configuration is used when it comes to streaming to RAM and then to the GPU. One direction may be porting the implementation to Linux and maybe even extending it with LITMUS [28]. Then it would be possible to place a high priority on streaming from the digitizer, which should produce a more deterministic behaviour in the link between the digitizer and host memory.

An interesting investigation could be to further investigate the difference in execution time between gaming cards and computation cards. Measurements in this report indicate that, when compared to a reference point, in this case the execution time of the FFT algorithm, the execution time of CUDA kernels share similar properties amongst gaming cards and computation cards. More on this can be read in Section 4.1.2.

The implementation is a low-level implementation in terms of program code. When it comes to code reproducibility, portability and comprehensiveness it lacks compared to more high-level alternatives. There are a number of different frameworks that can be used to elevate parts of the code to a format that is easier to comprehend, while still allowing for low-level manipulations in CUDA or even assembler.

Arrayfire [7] is a framework used to implement the visualization in the application. Mainly it is used for making computations and data transfers easier in terms of abstracting low-level code implementations. It also has other features like a graphics library.

Also the use of dynamic runtime systems like StarPU [1] could be used to abstract the entire work flow. By managing each operation as tasks they could be scheduled by the runtime system and executed dynamically when there is data to be processed. This approach can be investigated and evaluated in terms of the benefits and possible limitations of the approach.

Furthermore the use of skeletons made available by the SkePU [21] framework could be considered. This provides further abstractions and exploit parallelism while keeping performance of the application. An introduction to the subject can be found in a paper [31] by Kessler et al.


Bibliography

- [1] C. Augonnet, S. Thibault, R. Namyst, and P-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23(2):187–198, 2011.
- [2] A. Betts and A. Donaldson. Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis. *2013 25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 193 – 202, 2013.
- [3] R. Bittner and E. Ruf. Direct GPU/FPGA Communication via PCI Express. *2012 41st International Conference on Parallel Processing Workshops (ICPPW)*, pages 135 – 139, 2012.
- [4] A. Brandt. *Noise and Vibration Analysis : Signal Analysis and Experimental Procedures*. Wiley, 2010.
- [5] E.O. Brigham and R.W. Morrow. The fast Fourier transform. *IEEE Spectrum*, 4(12):63 – 70, 1967.
- [6] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.
- [7] ArrayFire Corporation. Arrayfire. <http://arrayfire.com/>, May 2014.
- [8] Microsoft Corporation. Microsoft Visual Studio. <http://www.visualstudio.com/>, May 2014.
- [9] NVIDIA Corporation. NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110. <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>, 2012.
- [10] NVIDIA Corporation. NVIDIA CUDA Fast Fourier transform library documentation. <http://docs.nvidia.com/cuda/cufft/>, April 2014.
- [11] NVIDIA Corporation. NVIDIA GPUDirect. <https://developer.nvidia.com/gpudirect>, May 2014.
- [12] NVIDIA Corporation. NVIDIA GPUDirect RDMA. <http://docs.nvidia.com/cuda/gpudirect-rdma/index.html>, May 2014.

- [13] NVIDIA Corporation. NVIDIA Homepage. <http://www.nvidia.com/>, May 2014.
- [14] NVIDIA Corporation. NVIDIA Nsight. <https://developer.nvidia.com/nvidia-nsight-visual-studio-edition>, April 2014.
- [15] NVIDIA Corporation. NVIDIA Performance report 5.0. https://developer.nvidia.com/sites/default/files/akamai/cuda/files/CUDADownloads/CUDA_5.0_Math_Libraries_Performance.pdf, April 2014.
- [16] NVIDIA Corporation. NVIDIA Programming guide. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, April 2014.
- [17] NVIDIA Corporation. NVIDIA Tesla summary. <http://www.nvidia.com/object/tesla-servers.html>, April 2014.
- [18] X. Cui, Y. Chen, and H. Mei. Improving Performance of Matrix Multiplication and FFT on GPU. *2009 15th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 42 – 48, 2009.
- [19] U. Dastgeer. *Performance-aware Component Composition for GPU-based systems*. PhD thesis, Linköping University, The Institute of Technology, 2014.
- [20] P. Duhamel and M. Vetterli. Fast Fourier transforms: a tutorial review and a state of the art. *Signal Processing*, 19(4):259 – 299, 1990.
- [21] J. Enmyren and C. Kessler. SkePU: a multi-backend skeleton programming library for multi-GPU systems. *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14, 2010.
- [22] Eclipse Foundation. Eclipse. <http://www.eclipse.org/>, May 2014.
- [23] M. Frigo and S.G. Johnsen. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216 – 231, 2005.
- [24] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5 – 48, 1991.
- [25] N.K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. *International Conference for High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008.*, pages 1 – 12, 2008.
- [26] F.J. Harris. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE*, 66(1):51 – 83, 1978.

- [27] IEEE. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005*, pages 1 – 560, 2006.
- [28] IEEE. LITMUS^ÂT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. *2006. RTSS '06. 27th IEEE International Real-Time Systems Symposium*, pages 111 – 126, 2006.
- [29] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1 – 58, August 2008.
- [30] H. Jeong, W. Lee, J. Pak, K-h. Choi, S-H. Park, J-s Yoo, J.H Kim, J. Lee, and Y.W Lee. Performance of Kepler GTX Titan GPUs and Xeon Phi System. *Contribution to proceedings of the 31st International Symposium on Lattice Field Theory*, 2013.
- [31] C. Kessler, U. Dastgeer, S. Thibault, R. Namyst, A. Richards, U. Dolinsky, S. Benkner, J.L. Traff, and S. Pllana. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1403 – 1408, 2012.
- [32] H. Lee. On orthogonal transformations. *IEEE Transactions on Circuits and Systems*, 32(11):1169 – 1177, 1985.
- [33] Y. Li, Y. Zhang, H. Jia, G. Long, and K. Wang. Automatic FFT Performance Tuning on OpenCL GPUs. *2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 228 – 235, 2011.
- [34] Z. Lili, Z. Shengbing, Z. Meng, and Z. Yi. Streaming FFT Asynchronously on Graphics Processor Units. *2010 International Forum on Information Technology and Applications (IFITA)*, pages 308 – 312, 2010.
- [35] PCI-SIG. PCIe FAQ. http://www.pcisig.com/news_room/faqs/pcie3.0-faq/#EQ2, May 2014.
- [36] K. M. M. Prabhu. *Window functions and their applications in signal processing*. CRC Press, 2014.
- [37] S. Qi, X. Wang, and S. Shi. Mixed Precision Method for GPU-based FFT. *2011 IEEE 14th International Conference on Computational Science and Engineering (CSE)*, pages 580 – 586, 2011.
- [38] S.W. Smith. *The scientist and engineer's guide to digital signal processing, 2nd edition*. California Technical Publishing, 1999.
- [39] H.V. Sorensen, D.L. Jones, M. Heideman, and C.S. Burrus. Real-valued fast Fourier transform algorithms. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(6):849 – 863, 1987.

- [40] D. Sundararajan. *Discrete Fourier Transform : Theory, Algorithms and Applications*. World Scientific, 2001.
- [41] D. Takahashi. Implementation of Parallel 1-D FFT on GPU Clusters. *2013 IEEE 16th International Conference on Computational Science and Engineering (CSE)*, pages 174 – 180, 2013.
- [42] L. Tan. *Digital Signal Processing : Fundamentals and Applications*. Academic Press, 2007.
- [43] W. van Drongelen. *Signal Processing for Neuroscientists : An Introduction to the Analysis of Physiological Signals*. Academic Press, 2006.
- [44] N. Whitehead and A. Fit-Florea. Precision Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. <http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf>, 2011.
- [45] C.J. Zarowski. *Introduction to Numerical Analysis for Electrical and Computer Engineers*. Wiley, 2004.

 Avdelning, Institution Division, Department Division for Software and Systems, Dept. of Computer and Information Science 581 83 Linköping		Datum Date January 2015
Språk Language <input type="checkbox"/> Svenska/Swedish <input checked="" type="checkbox"/> Engelska/English <input type="checkbox"/> _____	Rapporttyp Report category <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	ISBN ISBN <hr/> ISRN LIU-IDA/LITH-EX-A-14/026-SE <hr/> Serietitel och serienummer ISSN Title of series, numbering - <hr/> Linköping Studies in Science and Technology Thesis No. 14/026-SE
URL för elektronisk version http://XXX		
Titel Title Implementation of a real-time Fast Fourier Transform on a Graphics Processing Unit with data streamed from a high-performance digitizer Författare Author Jonas Henriksson		
Sammanfattning Abstract <p>In this thesis we evaluate the prospects of performing real-time digital signal processing on a graphics processing unit (GPU) when linked together with a high-performance digitizer. A graphics card is acquired and an implementation developed that address issues such as transportation of data and capability of coping with the throughput of the data stream. Furthermore, it consists of an algorithm for executing consecutive fast Fourier transforms on the digitized signal together with averaging and visualization of the output spectrum.</p> <p>An empirical approach has been used when researching different available options for streaming data. For better performance, an analysis of the introduced noise of using single-precision over double-precision has been performed to decide on the required precision in the context of this thesis. The choice of graphics card is based on an empirical investigation coupled with a measurement-based approach.</p> <p>An implementation in single-precision with streaming from the digitizer, by means of double buffering in CPU RAM, capable of speeds up to 3.0 GB/s is presented. Measurements indicate that even higher bandwidths are possible without overflowing the GPU. Tests show that the implementation is capable of computing the spectrum for transform sizes of 2^{21}, however measurements indicate that higher and lower transform sizes are possible. The results of the computations are visualized in real-time.</p>		
Nyckelord Keywords FFT, GPU, Digitizer, Real-time		

På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: <http://www.ep.liu.se/>