Signal Processing on a Graphics Card

An Analysis of Performance and Accuracy

Prepared By:
Arjun Radhakrishnan

Supervised By:
Prof. Michael Inggs

A dissertation submitted to the Department of Electrical Engineering,
University of Cape Town, in fulfilment of the requirements
for the degree of Bachelor of Science in Engineering.

Cape Town, October 2007

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Bachelor of Science in Engineering in the University of Cape Town. It has not been submitted before for any degree or examination in any other university.

Signature of Author . . . . . .

Cape Town

22 October 2007

Abstract

This report aims to investigate and test the application of graphics cards to common signal processing tasks. Recent developments in graphics processor technology have made it possible to use these processors for more general computing tasks than simply rendering graphics.

The report begins by introducing the graphics card and pipeline. It then moves on to an investigation of the tools that are available to facilitate general purpose computation from low level shader languages to high level software development kits. This is followed by an investigation of available libraries that perform the fast Fourier transform on the GPU.

Next, the CUDA Toolkit is used to test the fast Fourier transform on the GPU. The associated CUFFT library is used for the task and the CPU benchmark is run using the FFTW library's BenchFFT program. The GPU implementation was found to be significantly faster in execution speed than the CPU implementation for large transform sizes. The accuracy on the GPU was found to be good as well.

This is followed by the implementation of simple filters on the graphics card. The results obtained are compared with similar implementations on the CPU based on their performance and the accuracy. Here it was found that the CPU implementation performed far better than the GPU due to the slow memory accesses. Accuracy was good for finite impulse response filters but bad for infinite impulse response filters.

To conclude, graphics processors were found to be extremely fast when calculating FFTs and extremely inefficient for the filters. This dichotomy is caused due to the nature of programming for a GPU where memory accesses to device memory take significantly more clock cycles than to local registers. It is conceivable that an optimised GPU filter algorithm would surpass the CPU implementation for filters with a large number of taps. This brings into focus the importance of designing algorithms from the ground up to take full advantage of the power of the graphics card.

Acknowledgements

First and foremost I would like to thank my parents for their love, support and belief in me. They have given me the freedom to choose my path and learn from my mistakes and for that and everything else I wish to dedicate this report to them.

Thanks also to Professor Michael Inggs for keeping me on track with guidance when needed while allowing me to explore the topic in my own fashion. Additionally he allowed the purchase of a new PC and graphics card for testing which allowed me to use the CUDA toolkit.

For always guiding me when I was in doubt and his patience in dealing with some of the problems I faced I wish to thank Marc Brooker.

I also wish to express my gratitude to my friend and classmate Gregor George. When push came to shove he was right with me pushing back. Moreover his advice regarding both the content and structure of my final thesis was invaluable.

Last but not least, I want to thank the people I've met here at UCT who have made these years a pleasure - Aileen, Hugo, Joe, Lichklyn, Rhulani, Tapfuma, Thabang, Tsele and Vafa, thank you all!

Contents

List of Figures

List of Tables

Nomenclature

AGP Accelerated Graphics Port

ALU Arithmetic Logic Unit

API Application Programming Interface

CPU Central Processing Unit

DRAM Dynamic Random Access Memory

FFT Fast Fourier Transform

FIR Finite Impulse Response

FLOPS Floating point Operations per Second

GFLOPS GigaFLOPS (1 billion FLOPS)

GPGPU General-Purpose Computation on Graphics Processing Units

GPU Graphics Processing Unit

HAL Hardware Abstraction Layer

IIR Infinite Impulse Response

PCI Peripheral Component Interconnect

Raster Graphics This is the technique of representing a digital image on a rectangular grid of pixels for display, generally on a computer monitor

SIMD Single Instruction Multiple Data

SPU Stream Processing Unit

SRAM Static Random Access Memory

Warp The term is used metaphorically and comes from its use in the textile industry where it refers to a set of threads that make up the cloth. Here it refers to the threads that make up a process.

Chapter 1

Introduction

This thesis investigates the currently available set of tools and libraries that enable one to perform signal processing on a graphics processing unit. This introduction begins with a brief background to the topic following which the main objectives for the thesis are outlined. Next the scope of the work is stated, finishing with a short outline of the remainder of the thesis.

1.1 Background and Justification

In recent years, the Graphics Processing Unit (GPU) has gone from being a limited and specialised computer peripheral to one that has found application in a wide variety of fields from image processing to physics and biological simulations. This kind of use is termed General-Purpose Computation on Graphics Processing Units (GPGPU). It is mainly due to the inherent parallelism of these processors that they can operate on certain kinds of datasets at several times the speed of a CPU. A graph showing the increase in pure processing power over the past few years can be seen in Figure 1.1. The performance is measured in how many billions of floating point operations can be performed per second (GFLOPS).

Signal processing is one area that readily lends itself to implementation on the GPU because most of these algorithms deal with performing some transformation on a vector of input data. Here the parallelism of the graphics card allows far better throughput than even a modern multi-core CPU. However, there are some concerns that have made GPUs less appealing for general-purpose tasks in the past such as limited programmability of the graphics pipeline, lack of easy-to-use tools for creating optimized code and the high computational cost of branching operations.

New developments in GPU technology have made the first two objections obsolete as the current generation of graphics cards offer fully programmable pipelines and vendor supported GPGPU tools. The third limitation is an integral part of why the GPU is as

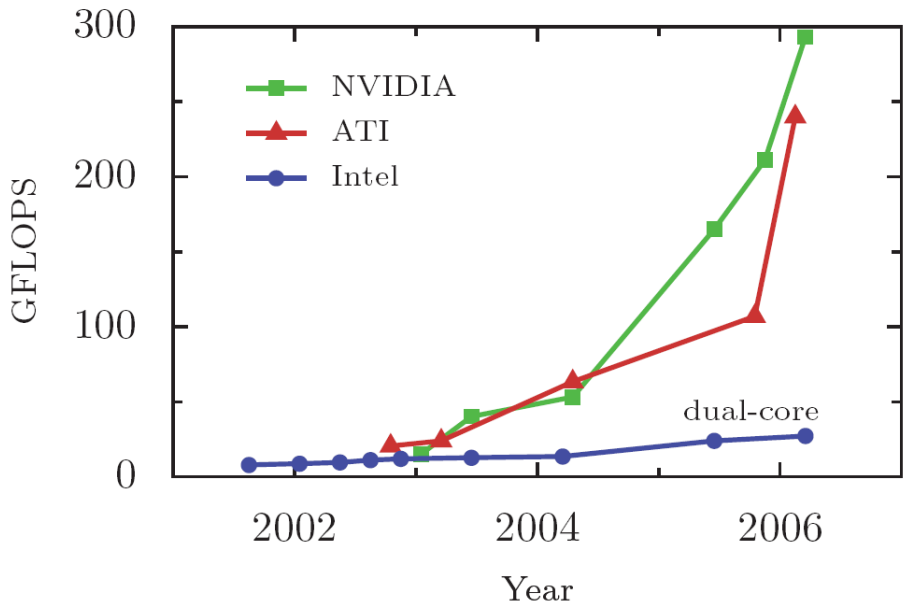

Figure 1.1: Comparative performance of current generation GPUs & CPUs [1]

fast as it is and as such, not likely to change any time soon. Thus one cannot envision a situation where the GPU makes the CPU obsolete, but it is now a viable co-processor that can take some of the routine number-crunching work from the CPU and allow it to focus on the tasks it is better suited for.

1.2 Problem Definition

This thesis examines the currently available software development kits and toolsets that allow one to implement common signal processing primitives such as the Fast Fourier Transform (FFT) and simple Infinite Impulse Response (IIR) and Finite Impulse Response (FIR) filters on a GPU. Following that, the accuracy and performance of a chosen GPU implementation is compared to those of a popular CPU implementation.

1.3 Thesis Objectives

The main objectives of this thesis are to:

- Introduce readers unfamiliar with the operation of the GPU to its basic workings as well as its relation to the graphics card.
- Provide a thorough review of the state of general purpose computing on graphics processing units including the various toolsets available.
- Investigate the existing libraries for implementing the signal processing primitives to be investigated on GPUs and comment on their suitability to the task.

- Test the Fast Fourier Transform on the GPU and compare the accuracy and performance of the results with a benchmark CPU implementation.
- Implement and test the response of Finite and Infinite Impulse Response filters on the graphics card using a variety of input signals. Compare the accuracy and performance of the results obtained with a benchmark CPU implementation.
- Conclude about the suitability of GPUs to be used in signal processing based on the results obtained.
- Identify possible future applications of these methods and avenues for further research in the field.

1.4 Scope and Limitations

This thesis deals specifically with testing existing FFT libraries and various filters on the GPU and comparing the accuracy of results and performance of execution with reference CPU-based implementations. It does not deal with the implementation of the actual FFT algorithm on a GPU, only describing previous attempts in the literature review. Neither will it consider the implementation of other general applications on GPUs.

The only graphics cards that will be tested in this thesis are NVIDIA graphics cards because the drivers available for AMD cards on Linux are lacking in several critical features. For the filters, a straightforward port of CPU code will be used. This is due to the lack of sufficient time to implement and test an optimised algorithm since the test hardware arrived with only two weeks to go before the deadline.

1.5 Plan of Development

The plan of development for the remainder of the thesis is presented below.

Chapter 2 begins with an overall background to graphics processors, discussing in some detail the most popular ones available as well as describing their hardware and capabilities. This is followed by a brief overview and comparison of the two competing graphics application programming interfaces (APIs) available - OpenGL and Direct3D. Next a detailed description of the typical pipeline of a consumer graphics card is given and their relation to the graphics APIs is explained. The chapter ends with a review of previous attempts at executing the FFT on the GPU and optimising the movement of data to and from the GPU.

Chapter 3 then moves the discussion on to GPGPU. This is the investigative part of the thesis and involves a thorough survey of the tools available - both high and low level. Depending on the level of control and performance needed the trade-off between ease of

implementation and speed of execution can be made. Since there are several optimised toolkits that allow one to program the GPU in a high level language the use of shader languages for general purpose computing is generally not necessary, but are briefly covered for completeness. There exist several implementations of the FFT on the CPU and GPU and next they are presented in brief. The chapter ends with a brief discussion of all the tools available and the choice of NVIDIA's CUDA toolkit and associated CUFFT library is justified for testing the GPU. The FFTW library will be used to test the CPU.

Chapter 4 deals with the implementation and testing of the CUFFT library. 1D, 2D and 3D in-place and out-of-place real and complex transforms will be investigated in a series of experiments on the NVIDIA GeForce 8600GTS. The same tests will be run on the Intel Core 2 Duo processor using the FFTW library and the results will be compared for accuracy and performance.

In the test on 1D performance, for small arrays processed individually the CPU performed better than the GPU. However the GPU can process in batches and as such achieves higher numbers of FLOPS (floating point operations per second). As the transform sizes increased, the performance of the GPU showed. For a transform of size 131072 elements, the GPU was able to process at over 4200 MFLOPS for batch sizes of 16 and 64. This is compared to around 2000 MFLOPS on the CPU - the GPU offers more than twice the performance. Similar trends were observed with the 2D and 3D FFTs. Starting out performing worse than CPUs for small transform sizes, as the sizes increased so too did the GPU's performance. In terms of accuracy, the effects on the output due to the single precision nature of the GPU was not significant - of the order of 10^{-5} across all the tests.

Performance measurement can take many forms aside from the common unit of FLOPS. In today's mobile and power conscious world, performance per watt is fast becoming a key measure. Also an interesting measure of performance would be performance per Rand or the price-performance ratio. All these are presented at the end of the chapter.

Chapter 5 deals with the testing and implementation of filters. Filter simulations with various input signals are to be run on the GPU and CPU and the results will be compared for accuracy and performance. Special note will be made of the effect on the result due to the lack of double-precision floating point capabilities on the GPU. It was expected that errors in filters with feedback (IIR filters) would be magnified and distort the output significantly while FIR filters would be more stable to errors due to rounding and do not experience this effect. However due to limited time, I was unable to test the IIR filter on the GPU to confirm this.

The FIR filter was designed as a Direct Form-I filter and implemented on the graphics card successfully. However, the performance observed was extremely slow compared to the CPU. For a filter with 500 taps, and an impulse as the input, the FIR filter calculated the output on the CPU in $3 * 10^{-6}$ seconds. The same algorithm on the GPU took 0.3 seconds. This huge difference in performance was due to the cache-inefficient nature of

the algorithm implemented and every time a memory access was made there was a very large performance cost associated with it.

Chapter 6 concludes that GPUs can offer large performance benefits when applied to signal processing tasks. However, one must be aware of the differences in the programming model between the CPU and GPU. A naive implementation of an algorithm that is efficient on the CPU will most probably give unsatisfactory performance as was seen in the case of the FIR filter. The main portion of the thesis concludes with my recommendations for further research in this field.

The Appendices contain information about the history of the GPU as well as a preview of some of the new initiatives being taken by the major players in the CPU and GPU market to enter the GPGPU field. Following that is some technical information about the hardware that was used in the tests. Tables of all the results obtained from the tests are included here as well.

Chapter 2

The Graphics Processor & Pipeline

In order to tackle the problem of running general-purpose programs on a specialised processor such as the GPU, one must first understand exactly how it works. All the terminology and programming paradigms for GPUs are aimed at making it easy to render 3D images on a 2D screen. In order to leverage the processing power available to the fullest extent, the data and code must be optimised for the graphics pipeline. In this section we will examine these elements in detail.

2.1 The Graphics Processing Unit

The GPU is the actual chip on a graphics card that performs the operations required to render pixels on the screen. Modern consumer GPUs are stream processing SIMD processors. This means that they are optimised to perform a single instruction, called a kernel function, on each element of a large dataset at the same time.

The driving force behind the rise of the GPU has been the multibillion-dollar video game industry. In fact the demand for graphics power is so great that GPU performance has been exceeding Moore's law by over 2.4 times a year [2]. The reason for this dramatic increase in power is the lack of control circuitry which occupies a majority of the space on a modern CPU. In contrast to this, GPUs devote almost all the space on the chip to the Arithmetic Logic Units (ALUs) thus tremendously increasing their pure processing power, but at the cost of severely penalising incorrect branch choices. For traditional purposes this is not an issue, but when attempting to use the GPU as a general purpose processor, this is a serious issue that must be taken into account. Figure 2.1 shows the relative amounts of transistors devoted to different tasks on the CPU as compared to the GPU.

Graphics cards are often marketed as being compliant with a specific graphics API and in that regard the industry is currently at the changeover point from the older Direct3D 9.0 based cards to the new Direct3D 10 compliant ones. A number of significant changes

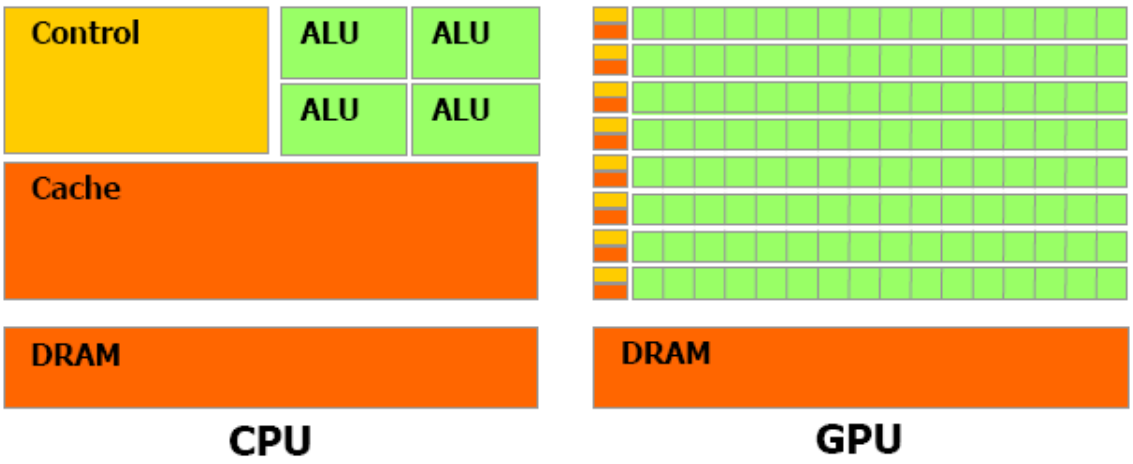

Figure 2.1: GPU & CPU Transistor Use Comparison [3]

have taken place between these two generations, specifically regarding the pipelines, that will be dealt with in Section 2.3.

The two main competitors in the consumer graphics arena are AMD and NVIDIA and their flagship graphics cards offer staggering performance.

The ATI Radeon HD 2900 XT from AMD has 320 stream processing units (SPUs), 128-bit floating point precision, 16 pipelines and boasts a theoretical memory bandwidth of 105.60 GB/s [4]. The NVIDIA GeForce 8800 Ultra meanwhile uses 128 SPUs, 128-bit floating point precision, 24 pipelines and has a theoretical memory bandwidth of 103.70 GB/s [5]. And for an indication of where the industry is headed, the next generation of ATI and NVIDIA graphics cards are touted to offer 1 TeraFLOPS as compared to the current 330 GigaFLOPS from NVIDIA and 450 GigaFLOPS from ATI [6].

2.2 Graphics APIs

At this stage I would like to briefly introduce the two main graphics APIs in use today and their respective advantages and disadvantages.

2.2.1 OpenGL

OpenGL (Open Graphics Library) was developed in 1992 by Silicon Graphics, Inc and is a platform-independent, multipurpose graphics API. Since it is an open standard, many companies contribute to its development, which is overseen by the OpenGL Architecture Review Board (ARB). The ARB specifies the features that must be present in any OpenGL distribution. It must be noted that the 'open' in OpenGL does not mean open source, and it is still a proprietary API.

OpenGL is currently at version 2.1 with version 3.0 having just been adopted by the ARB at the end of August 2007. OpenGL 3.0 is the first complete overhaul of the API and

makes it completely object based, not state based. It will include new developments in the GPU field as well as improve performance in existing technologies [7].

2.2.2 Direct3D

Direct3D is a part of the DirectX collection of APIs that provides libraries for creating 3D images on the Windows platform. It is a proprietary API and is developed and maintained by Microsoft. While in its initial release Direct3D was seriously flawed and lacked many of the functions provided by OpenGL, it has since evolved and is now at least its equal in performance and ease of use [8].

Direct3D is now up to version 10 which is available solely on Windows Vista as part of DirectX 10.0. Older Windows operating systems use Direct3D 9. There are several major improvements in Direct3D 10 that will be discussed in Section 2.3.2.

2.2.3 Comparing Direct3D and OpenGL

OpenGL and Direct3D are fundamentally different in their approaches to solving the problem of creating 3D graphics. OpenGL aims to specify a 3D rendering system that may be hardware accelerated. Direct3D on the other hand is derived from the features that the hardware supports [9].

This difference extends to the way they both work as well. Direct3D allows the application to manage the hardware resources, thus giving the creator more control. OpenGL on the other hand simplifies the API for the application developer by placing the burden of managing the hardware resources on the creator of the hardware driver.

The major advantage that OpenGL has over Direct3D is its platform independence. For this reason it is favoured by most researchers and graphic designers. However, due to the widespread use of the Windows operating system on PCs, Direct3D has become the preferred platform for game development which is the major driver of GPU progress.

Finally, OpenGL allows hardware vendors to create their own extensions to the standard and advertise them to the API via the driver. This allows new functionality to be exposed quickly, at the cost of some confusion when different vendors use different APIs to implement similar extensions. Direct3D, being developed and maintained solely by Microsoft, is more consistent in the features offered at the cost of denying access to vendor-specific features.

Thus it can be seen that the pros and cons of both of these APIs currently balance each other out and the choice of API must come down to the specific needs of the user and tools available.

2.3 The Graphics Pipeline

When discussing the graphics pipeline, it is important to recognise the relationship between the software, the API and the hardware. This is because the pipeline is stipulated by the specific API in use, be it OpenGL or Direct3D. Generally the graphics API defines the way the software interacts with the hardware and specifies certain minimum requirements for compliance. Because of this, it is often called a Hardware Abstraction Layer (HAL) [10].

2.3.1 The Simplified Pipeline

The generic graphics pipeline shown in Figure 2.2 is the pre-Direct3D 10.0 model which we shall examine first. Instructions are sent from the application to the GPU. The API handles the transformations that occour between and including the 'command' and 'fragment' stages [11] and the output is finally sent to the appropriate display.

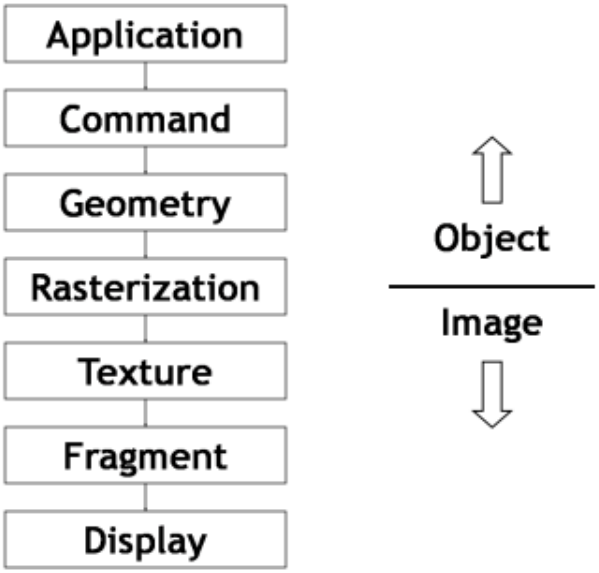

Figure 2.2: A Conceptual Pipeline [12]

I will now present an in-depth look at exactly what the the GPU does at each stage of the pipeline. The start of the pipeline is at the Application itself. It is here that the objects to be rendered are represented as basic geometric primitives - most commonly triangles. There are some visual optimisations that take place at this stage such as occlusion culling and visibility checking that does not play a big role in GPGPU and can be safely ignored for our purposes.

The Command stage buffers and interprets the application's call and translates it into a form understandable by the hardware.

Now on to the Geometry stage. In practice this stage is implemented in a module called the Vertex Shader Unit as it acts on the vertices of the triangles sent in from the application. The Vertex Shader Unit operates on the single instruction multiple data (SIMD) principle and has multiple execution units called Vertex Pipelines to allow high throughput. This stage is very useful in GPGPU applications as here is where an image would usually undergo transformations and projections to render the scene from the viewpoint of the virtual camera. These same transformations can be applied to any dataset and benefit from the parallel nature of the execution. The Vertex Pipeline is controlled by a program called a Vertex Shader, written in a shading language. These can be low-level (Assembler) or high-level (HLSL, GLSL, Cg). In general the performance gained by using the low-level language is offset by its complexity to learn and implement. Most modern graphics applications use high-level shader languages to control the execution in the Vertex Pipelines.

The next stage in the general pipeline is the Rasterisation stage. This is done in the Fragment Processing Unit. It is also called the Pixel Shader Unit. Like the Vertex Shader Unit, it operates on the SIMD principle and has a number of Fragment or Pixel Pipelines that can be programmed by means of a Pixel Shader. In general, pixel pipelines are grouped into 4 and can thus process 4 fragments at a time. These 'quads' are the smallest elements processed by the Pixel Shader Unit. As in the Vertex Shading Units, Pixel Shading Units can also be used to implement GPGPU.

The Texture stage is managed by the Texture Management Unit. They allow access to a number of predefined textures from within the Pixel or Vertex Pipelines. They fetch the appropriate texture from the memory, transform and project it, filter it and finally make it available to the fragment being processed.

The final stage of the API's pipeline is the Fragment stage. This takes place in the Raster Operation Unit, which is composed of several Raster Operation Processors or Z-pipes. The fragments that leave the Pixel Pipeline may still undergo some changes before they are ready to be displayed. Some tests they undergo are the scissor, alpha, stencil, depth and colour blending tests. Again, none of these are specifically important to GPGPU and thus will be omitted from this discussion. The final image is now ready to be written to the frame buffer and displayed on screen.

Figure 2.3 shows a complete graphics pipeline as it would be implemented in hardware.

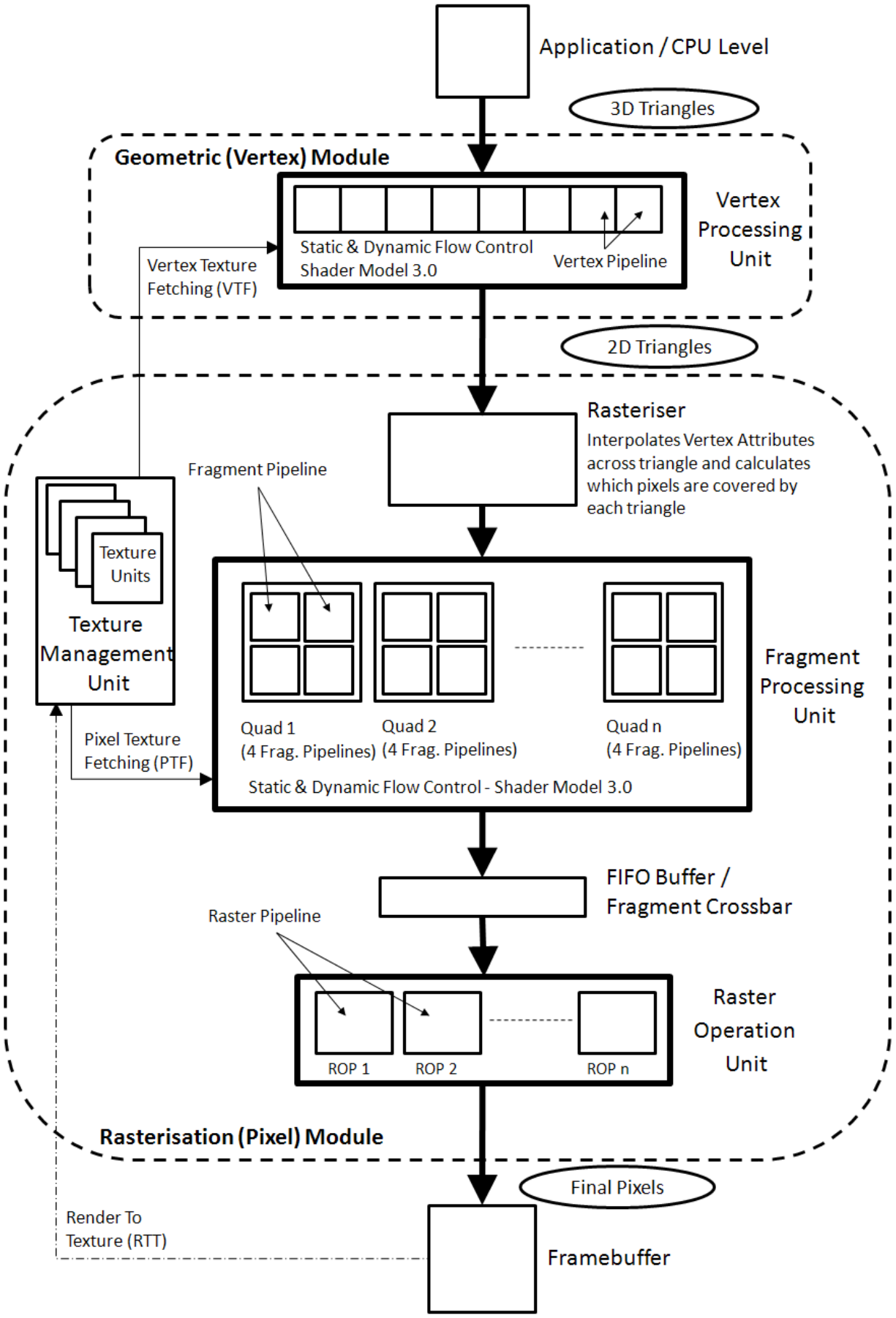

Figure 2.3: Hardware Pipeline [11]

2.3.2 The New Direct3D 10 Pipeline

Now that we have sufficient understanding of the previous generation of graphics cards, let us briefly examine the major aspects of the Direct3D 10 breed. With Direct3D 10, the 3D pipeline has become completely programmable, finally moving away from fixed function pipelines completely [13] as shown in Figure 2.4. While it appears similar to the older generation, and most of the stages perform similar functions, there are some significant differences namely the new Input Assembler, Geometry Shader and Output Merger stages.

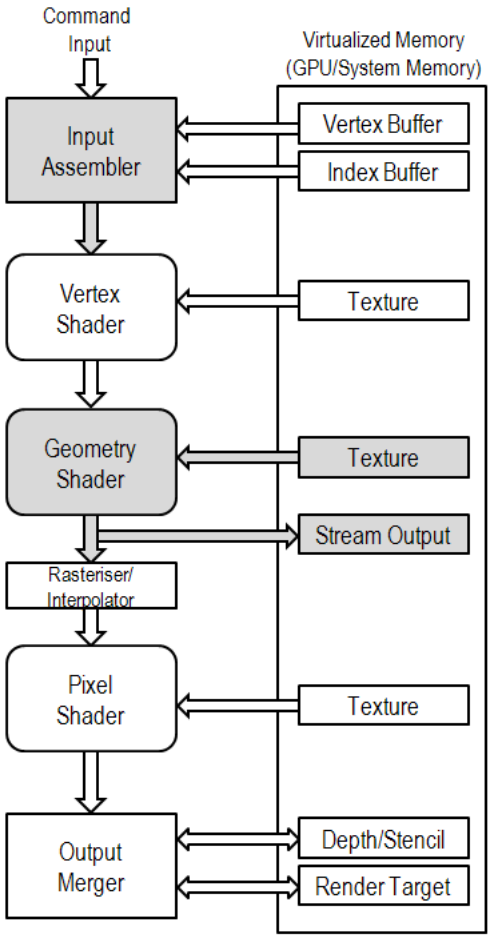

Figure 2.4: The Direct3D 10 Pipeline [13]

Firstly, the Input Assembler takes in the index and vertex streams and assembles them into the geometric primitives (triangles, line lists, etc.) that will be used in other stages of the pipeline. The secondary purpose of the Input Assembler is to make the shaders more efficient by attaching system-generated values called 'semantics'. These semantics can reduce processing to only those primitives, instances or vertices that have not been processed yet [14]. Finally, the Input Assembler also deals with Stream-Out data, which is fed back into the pipeline from the Geometry Shader.

The second new stage is the Geometry Shader, which allows for the execution of shader-code with multiple vertices as input and the generation of vertices as output. In addition to this, the output from the Geometry Shader can be fed back into the Input Assembler to

be re-processed. This allows multi-pass geometry processing with minimal intervention by the CPU which is good for parallelism.

Finally, we get to the Output Merger. This is the last stage of the pipeline and takes the results of the previous stages, combines it with the contents of the render target and produces the final pixel value that is displayed on the screen.

2.4 Review of Literature

2.4.1 The FFT on a GPU

Moreland and Angel have co-authored two papers dealing with the implementation of signal processing primitives on a GPU[15, 2]. In them, once they explain fundamental concepts such as the impulse response, convolution and the Fourier transform, the authors present the discrete Fourier transform (DFT) and the method for performing it efficiently on a computer - the fast Fourier transform. There are several algorithms that implement it for different kinds of input data. One of the most popular methods is the Cooley-Tukey algorithm. This algorithm reduces the operational complexity of performing the DFT on input transforms who's size is a power of two from $O(N^2)$ to $O(N \log N)$. This is the algorithm that Moreland and Angel implemented in their paper.

The algorithm works on the principle of divide and conquer. The input is split in two with one half being all the even indices and the other, the odd indices. The Fourier transform may be applied separately to each half and the two halves combine to form a full transform. This is applied recursively to get $N/2$ DFTs of 2 points. In practice instead of using recursion, the algorithm implements an iterative butterfly network. Each 2-point DFTs requires two complex multiplications and a complex addition. These results are then combined into the final output.

There are several optimisations to the basic algorithm to ensure that the execution is as fast as possible. One is to reverse the bits of the input indices to allow the program to treat the sub-sequences as continuous chunks in the array. Another is to use the conjugate nature of real samples to enable frequency compression.

Moreland and Angel implemented this algorithm on the graphics card by programming the fragment processing unit of the graphics pipeline in Cg, a shader programming language. The FFT was first applied to each row, then to each column. The results obtained from their tests showed their application performing on average at about 2.5 GigaFLOPS and taking 2.7 seconds on an NVIDIA GeForce FX 5800 Ultra as compared to 0.625 seconds (or 2 seconds if the transforms were not padded appropriately) on a 1.7GHz Intel Xeon processor using FFTW. This, the authors felt, was a competitive result, especially given that the Intel Xeon cost several times as much as the GeForce FX 5800.

2.4.2 Designing Algorithms for Efficient Memory Access

In the paper entitled "A Memory Model for Scientific Algorithms on Graphics Processors" [16], Govindaraju et al. present a cache-efficient algorithm for implementing memory intensive scientific applications, including the FFT, on modern GPU architectures.

The authors describe the memory model of GPUs and how it differs from the traditional CPU memory model. In essence, GPUs consist of small write-through caches near the fragment processors and use a dedicated high-bandwidth, high-latency memory subsystem as shown in Figure 2.5. This is in contrast to the low-latency and low-bandwidth memory that the CPU accesses. It is due to this high latency that the efficient use of the GPU's cache can significantly increase the speed of some memory intensive algorithms.

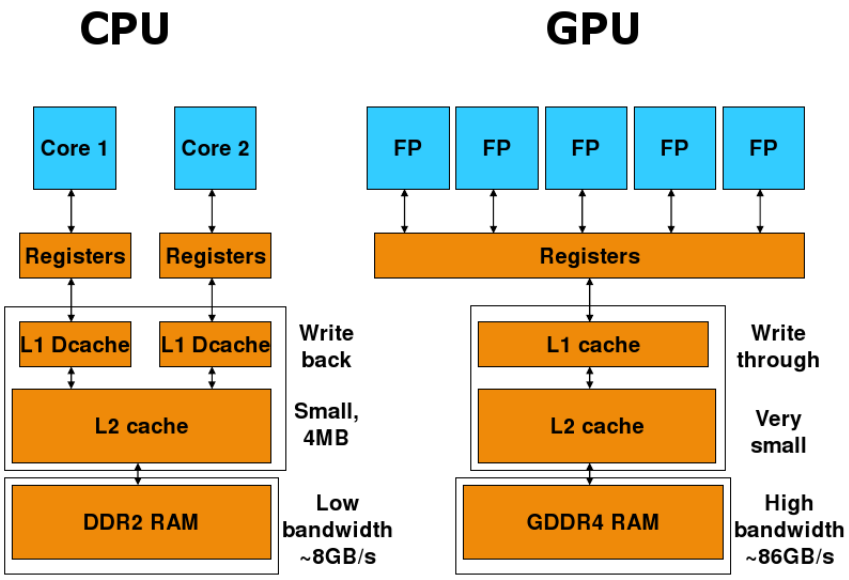

Figure 2.5: GPU and CPU Memory Models[17]

In order to mask the high latency, GPUs prefetch blocks of data values from the DRAM-based video memory into small, fast SRAM-based caches that are local to the fragment processors. This allows the GPU's bandwidth when accessing the main memory to be up to 10 times that of the CPU. Also, due to their parallelism and the fact that they have fewer pipeline stages than CPUs, GPUs have much smaller caches which means that only a few blocks can fit in the cache at a time.

When designing algorithms for the GPU, it is important to take these factors into consideration. For example, since GPUs are able to compute extremely fast, it is preferable to avoid copying any extra data to the GPU if it is possible to calculate it on the GPU during execution. Additionally, since GPUs are optimised to operate on 2D arrays, 1D arrays must be mapped onto a 2D array for optimum efficiency.

Chapter 3

Tools Available for General-Purpose Computation on a GPU

Given the enormous processing power of GPUs, it is easy to see why interest in graphics cards as general purpose processors is on the rise. Besides the increases in raw power, graphics processors have become increasingly programmable. Graphics cards support full IEEE-754 single-precision vectorised floating-point operations [1] and it has been possible to use high-level shader languages to program the vertex and pixel pipelines since the release of the NVIDIA GeForce 3 and ATI Radeon 8500 [18]. More recently, with the release of the Direct3D 10 compliant GPUs with their unified shaders and easy programmability, there have been attempts to make programming for the GPU even simpler with vendor supported APIs. This chapter will investigate some of the APIs and tools available that facilitate the application of GPUs to general processing tasks followed by an overview of the libraries dealing specifically with implementing the FFT.

3.1 High Level GPGPU APIs

With the introduction of their respective Direct3D 10 cards, both AMD and NVIDIA have released special APIs to facilitate their application in general purpose processing. These are NVIDIA's CUDA™ (Compute Unified Device Architecture) and AMD's CTM ™ (Close to Metal) standard development kits (SDKs). In addition to these official APIs, there are two other GPGPU oriented toolsets available - BrookGPU and Rapidmind. In this section, we will provide an overview of each of these.

3.1.1 CUDA

CUDA allows programmers to write code in C that will compile to run on the graphics card, exploiting its massively parallel nature. This technology is only available to the GeForce 8 Series graphics cards and upwards in the commercial line and Quadro Professional Graphics solutions. Older cards cannot be supported because of the somewhat limited programmability of the pipeline. It was made available to the public on both Windows and Linux on February 16, 2007 [19].

The CUDA software stack is as shown in Figure 3.1 with a lightweight driver and runtime layer running on top of the hardware. The higher-level mathematical libraries run on top of this runtime[3].

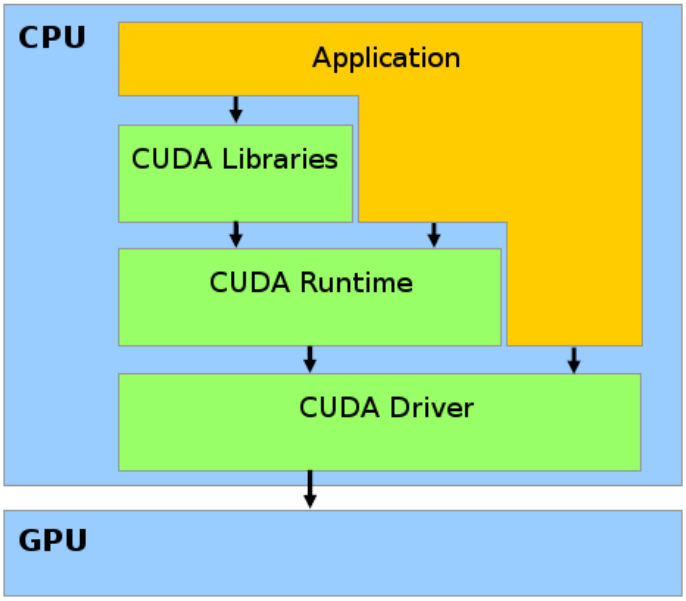

Figure 3.1: CUDA Software Stack [3]

The CUDA API consists of two parts:

- firstly, a minimal set of extensions to the C programming language that allow the programmer to target certain portions of the code to run on the GPU
- and secondly, a runtime library that consists of a host component, device component and common component. The host component runs on the host and allows for control and access to the GPU from there. The device component provides device-specific functions. And finally the common component provides a subset of the C standard library that is supported to run on the device.

A big advantage of using CUDA is that it is officially supported by NVIDIA and there is thorough documentation regarding its use. A drawback would be its limited portability and lack of compatibility with older graphics cards.

3.1.2 CTM

AMD's CTM is an open source project that aims to provide a thin hardware interface to the ATI graphics cards for GPGPU [20]. It can only be used with Radeon R580 based graphics cards and AMD Stream Processors. It is already being used in the University of Stanford research project – Folding@Home to provide 20 to 40 times the performance of a single CPU. Since we have discounted the use of ATI graphics cards for this study, a more detailed analysis will not be undertaken however more information is available on the AMD website relating to CTM.

3.1.3 Rapidmind

Rapidmind is the commercial spin-off of the LibSh programming language for GPUs. It provides libraries and headers that can be used to accelerate GPGPU and is available on both Windows and Linux. It is not limited to GPUs however and can be used to accelerate applications on multi-core CPUs or stream processing architectures such as the Cell Broadband Engine by IBM. Since it was born out of the free LibSh language, Rapidmind is freely available for non-commercial use.

One of the most impressive aspects of Rapidmind is its support of a wide variety of hardware and software. It can be used to accelerate all NVIDIA and ATI graphics cards that have been released in the past year. It is also officially supported on Windows XP, Windows Vista and several of the most commonly used Linux distributions. Finally it can be used in conjunction with either the GCC 4 compiler on Linux or Microsoft Visual C++ on Windows. This portability is very important and a great advantage of using Rapidmind [21].

3.1.4 BrookGPU

Brook is a stream programming language and BrookGPU is a compiler and runtime implementation of Brook for graphics hardware. They are both research projects at Stanford University [22]. Brook is described on its website in the following manner:

> "Brook is an extension of standard ANSI C and is designed to incorporate the ideas of data parallel computing and arithmetic intensity into a familiar and efficient language."
>
> - BrookGPU: Introduction [22]

3.2 Shading Languages

Apart from the high-level libraries and APIs for GPGPU, one can exploit the more direct control over the shaders that is offered by shading languages. These are still not as low level as assembly code but offer fine-grained control over the graphics pipeline.

3.2.1 Cg

Cg is NVIDIA's shading language and extends the C programming language to allow programming the vertex and pixel shaders in GPUs. Code written in Cg is portable to a wide range of platforms and hardware. It is compiled using the Cg compiler which optimises the code for the hardware on which it is to be run [23].

3.2.2 HLSL

HLSL (High Level Shading Language) is the proprietary language used by Microsoft's Direct3D API. It is similar to NVIDIA's Cg programming language [24]. In Direct3D 10 there are three shaders available – vertex, pixel and geometry. Geometry shaders are absent from previous versions.

3.2.3 GLSL

OpenGL Shading Language (GLSL or GLslang) is the C-based shader programming language that became a part of OpenGL from version 2.0. It offers direct, platform independent control of the graphics pipeline. The code written in GLSL is passed to the hardware driver to compile and implement, thus the GPU creator can optimize the code for their specific architecture [25].

3.3 Existing FFT Libraries

Currently the best and most commonly used library for implementing the FFT on CPUs is one called FFTW. On GPUs, GPUFFTW, CUFFT and libgpufft are some of the libraries that can be used.

3.3.1 CPU Based – FFTW

FFTW is an acronym for Fastest Fourier Transform in the West. It is an optimised library written in ANSI C that can compute Discrete Fourier Transforms (DFT) of data with arbitrary length, rank, multiplicity and general memory layout. It is faster than most

other implementations available [26]. FFTW uses a variety of algorithms and chooses the best one based on the hardware architecture it is being run on. The library analyses the hardware and then creates a 'plan' that it uses to perform the rest of the computation.

Some of the algorithms used are the Cooley-Tukey algorithm, the prime factor algorithm, Rader's algorithm for prime sizes, and a split-radix algorithm (with a variation due to Dan Bernstein) [27]. FFTW's code generator also produces new algorithms on the fly when creating the plans.

FFTW can compute:

- the DFT of both real and complex data,
- the DFT of even- or odd-symmetric real data (discrete cosine or sine transforms)
- the input data can have arbitrary length. FFTW employs O(n log n) algorithms for all lengths, including prime numbers.

3.3.2 GPU Based

Before beginning the discussion of the FFT libraries available for the GPU, it must be noted that the current generation of graphics cards from both NVIDIA and ATI are single-precision processors. Graphics cards that are capable of double-precision will be released at the end of this year [28]. There have also been some attempts at emulating double precision.

GPUFFTW

This is a freely available library that implements the FFT in a manner that is optimised for graphics processors. It is related to the CPU based FFTW in name only, not in terms of its implementation. The GPUFFTW algorithm uses Stockham autosort.

It is fairly limited at the present time because it can only handle 1D power-of-two single precision FFTs [29]. 2D and 3D versions are in development and may become available soon. A sample benchmark from the official website that shows the performance increase over CPUs is shown in Figure 3.2. It works on NVIDIA GPUs from the GeForce 6 series onwards.

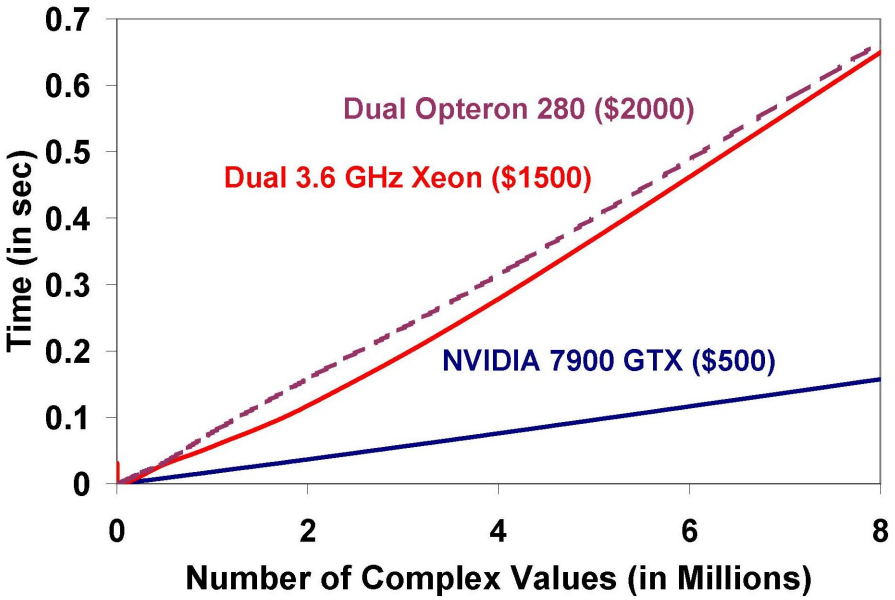

Figure 3.2: GPUFFTW Benchmarks [30]

CUFFT

This is NVIDIA's proprietary FFT implementation that only works with the newest generation (GeForce 8 series) of GPUs. It is a part of the CUDA library. Some of its features are [31]:

- 1D, 2D, and 3D transforms of complex and real-valued data.
- Batch execution for doing multiple 1D transforms in parallel.
- 2D and 3D transform sizes in the range [2, 16384] in any dimension.
- 1D transform sizes up to 8 million elements.
- In-place and out-of-place transforms for real and complex data.

The CUFFT library uses a technique similar to that employed by the FFTW library for CPUs whereby it first creates specific plans for the dataset, architecture and transforms in use and then implements it on the GPU.

libgpufft

This is an older library that also allows for GPU based FFT calculation on both ATI and NVIDIA hardware. It is implemented using the BrookGPU compiler and uses an improved version of Cooley-Tukey FFT algorithm. The newer GPUFFTW is as much as 3 times as fast for 1D FFTs [30]. However, this library does have some advantages [32]:

- libgpufft has an identical communication pattern for each stage, allowing the same fragment program to be used across all stages. Consequently libgpufft does not incur the cost of switching the fragment program between stages.
- libgpufft packs two complex floating point numbers into each fragment, allowing the 4-wide units to be used at maximum efficiency.
- libgpufft is written in Brook, making its code simple to understand and adapt.

3.4 Discussion and Comparison of the Tools for Use in This Thesis

All the tools presented in this chapter have been used successfully to implement various applications on the GPU. The question now turns to which of these is most suitable for the task at hand and which would provide the best results in the limited time available. This choice must be made based on the information gathered here as there is insufficient time to test each tool individually.

Firstly, I have decided against implementing the FFT algorithm using low-level shaders because there already exist several highly optimised libraries. Secondly, CTM must be removed from consideration due to the poor driver support for AMD graphics cards on Linux based operating systems. Now I will consider the other tools.

RapidMind and BrookGPU are both excellent tools that are being actively developed. Moreover they support a wide variety of graphics cards. RapidMind however, perhaps because it is a commercial product, seems to offer better documentation and support. Finally the CUDA toolkit is a proprietary API developed and maintained by NVIDIA and supports only its latest generation of GPUs. However this hardware is nearly a year old and widely available at a variety of price points. Given the factors discussed here, either CUDA or RapidMind would be good choices.

Now moving on to the FFT libraries, libgpufft is good because it supports a wide variety of hardware and offers many features, but it is a few years old and does not take advantage of the enhancements provided in the new hardware. GPUFFTW is extremely fast at calculating 1D FFTs and is optimised for the latest NVIDIA hardware, but is limited to just that. While it would be possible to implement a multidimensional FFT using a distributed single dimensional FFT, the limited time available makes that unfeasible. Finally we get to NVIDIA's CUFFT library which offers all the features required and has excellent documentation provided by NVIDIA.

Thus the final choice of toolkit and library that will be used for the rest of this investigation is NVIDIA's CUDA toolkit with the associated CUFFT library.

Chapter 4

The FFT on the GPU

We now move on to the implementation and testing of the chosen FFT library - the CUFFT library. 1D, 2D and 3D in-place and out-of-place real and complex transforms will be investigated.

4.1 Hardware and Environment

The hardware and operating system to be used in all the tests is shown in Table 4.1.

Type	Hardware
Processor	Intel Core 2 Duo E6550 Conroe 2.33GHz 4MB Unified Cache
RAM	Apacer 2048Mb DDR2 [DDR6400]
Hard Drive	Western Digital Caviar SE 160GB 7,200RPM SATA 8MB Buffer
Video Card	MSI NVIDIA GeForce 8600GTS 256MB
Power Supply	Superchannel 500W Switching Power Supply
Optical Drive	LG GSA-H44N DVD Writer
Motherboard	MSI Intel P35 Neo-F (16x PCIe bus for GPU)

Table 4.1: Standard Test Bed

The software and drivers used are shown in Table 4.2.

Software	Version
Operating System	Ubuntu Linux 7.10 Beta (Gutsy Gibbon)
Linux Version	2.6.22-14-generic
Make Version	GNU Make 3.81
Compiler	gcc 4.1.3
X Window System	X.Org 1.3.0
Video Drivers	NVIDIA Linux x86 100.14.11
Toolkit	NVIDIA CUDA v1.0
FFT (CPU)	FFTW 3.1.2
FFT (GPU)	NVIDIA CUFFT v1.0

Table 4.2: Software Environment

4.2 Factors affecting the FFT

Before delving into the actual tests that have been run, I will briefly explain what will be tested.

4.2.1 Multi-dimensional Datasets

The dimensionality of the transform affects the execution time because, at the lowest level, it means that there will need to be more multiplications and additions to get the result. Additionally the maximum transform size changes depending on the number of dimensions due to the limited memory on the GPU.

4.2.2 Power-of-two Transforms

All transforms that are to be tested will have sizes that are integer powers of two because calculating the FFT is most efficient at these sizes. While it is possible to calculate the FFT of non power-of-two and prime number sized transforms, it is generally advisable to pad the input data, of size n, with zeros or employ mirroring to achieve the smallest integer power of two greater than n [33].

4.2.3 In-place and Out-of place Transforms

An in-place FFT is calculated entirely in the original sample memory and does not require additional buffer memory. Out-of-place transforms however have distinct input arrays and output arrays. Out-of-place transforms are generally easier to implement and faster, however on many classes of hardware this extra space cannot be spared, hence the need for in-place algorithms.

4.2.4 Real and Complex Transforms

There is a computational advantage when dealing with purely real signals because then the n point real signal can be made into an $n/2 + 1$ point complex signal. The FFT can then be run on this signal and, once done, an untangling algorithm transforms the output into the complete FFT of the original real signal. This reduces the amount of memory required to run the FFT.

4.3 Testing Methodology

All tests measure the performance of the GPU including the bus transfer times. This is because unlike when processing on the CPU, data has to be sent to the GPU to be processed. This means that no matter how fast the GPU is at the number-crunching, the data transfer time will make the CPU more useful for smaller datasets as well as those times when the input data changes frequently. The overhead involved in copying the data to the GPU's memory and back also puts the burden of minimising the number of memory transfers that take place on the application programmer.

Table C.3 in Appendix C shows the speed of memory accesses between the main memory and the GPU cache for the test system used. This varies depending both on the GPU and the motherboard chipset used. Having said that, I will now describe the test for both the CPU and the GPU.

On the GPU, the chosen FFT is run and the maximum number of FFTs that can be performed per second is counted. To ensure that there is as little influence by the operating system as possible, the test is run 32 times in succession and the fastest run is recorded. The reason why the fastest run and not the average run is used is because preliminary tests showed that the variation in performance between successive runs can be very large due to random operating system interference. Thus the averages fluctuated greatly. To eliminate this randomness from the data the fastest run will be used. This number is then used to calculate the performance of the FFT in MFLOPS using formula 4.1 [26].

$$\frac{5 * numFFTs * nx * \log_2(nx)}{10^6} \tag{4.1}$$

where nx is the transform size and $numFFTs$ is the experimentally produced number of FFTs performed per second.

This same procedure is followed for several other transform sizes for each type of FFT being considered. Errors were calculated by having the program use a specified input file of real or complex numbers generated in Matlab. The FFT was run and the result was sent to a file. This was then compared to Matlab's output.

On the CPU, first the `fftw-wisdom` command is run to create optimised plans for the CPU and chipset. Next the benchFFT benchmarking utility provided by the library is used to run the same tests that were run for the GPU. To simplify the testing, a shell script was used to run the test 32 times and output the results to a file. Since the FFTW's performance was not prone to the random fluctuations caused by OS interference, the average of all the runs was computed. It must be noted that from the CPU usage monitors it appears that only one core of the dual core Intel CPU was actually being used as can be seen from Figure 4.1.

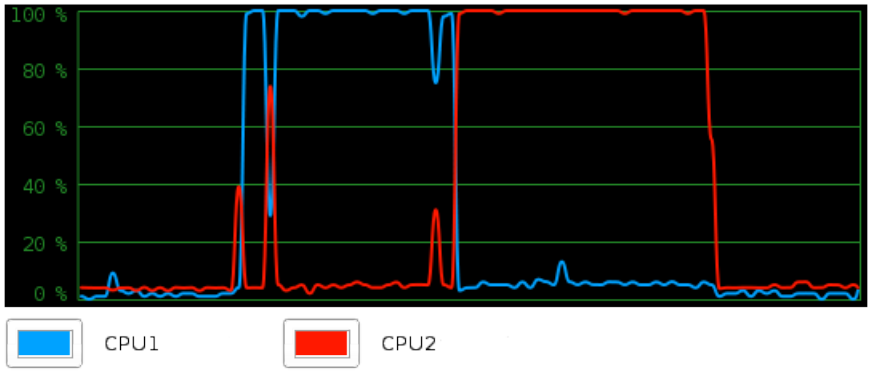

Figure 4.1: CPU Usage for FFTW benchmark

I will only present summarised results here along with the discussion. The complete results are tabulated in Appendix D.

4.4 Results of Tests

4.4.1 1D Transforms

The 1D transform size limit stipulated by the CUFFT library is 8 million elements. In addition to the test described in the last section, a feature provided by CUFFT for 1D transforms that is interesting to examine is the execution of the FFT in batches. This takes advantage of the parallelism of the GPU especially at lower transform sizes.

The graphs shown in Figures 4.2 and 4.3 are based on data captured while testing 1D in place complex-to-complex transforms, but the trends are the same across the board for these FFTs. The important results are:

- The performance of the CPU based FFTW algorithm, while competitive for smaller sized FFTs, falls behind the GPU based implementations as the size increases.
- The GPU is very inefficient at performing single FFTs on small datasets due to the high overhead incurred to transfer the data to and from the GPU memory.
- As the transform size increases, the difference in performance between the batched and single FFT methods shown in Figure 4.3 becomes very small until there is virtually no increase in performance when applying the FFT to 262144 elements.
- The GeForce 8600GTS cannot directly compute power-of-two transforms larger than 262144 elements due to the fact that it has only 256MB of memory on the card. The cufftComplex datatype is 8 bytes long and when using a batch size of 64 and transform size of 524288, the memory required is $524288 * 64 * 8 = 268435456$ bytes, or 256MB. Since some portion of the memory on the GPU is used by the X window system, the function cannot allocate sufficient memory process the data.

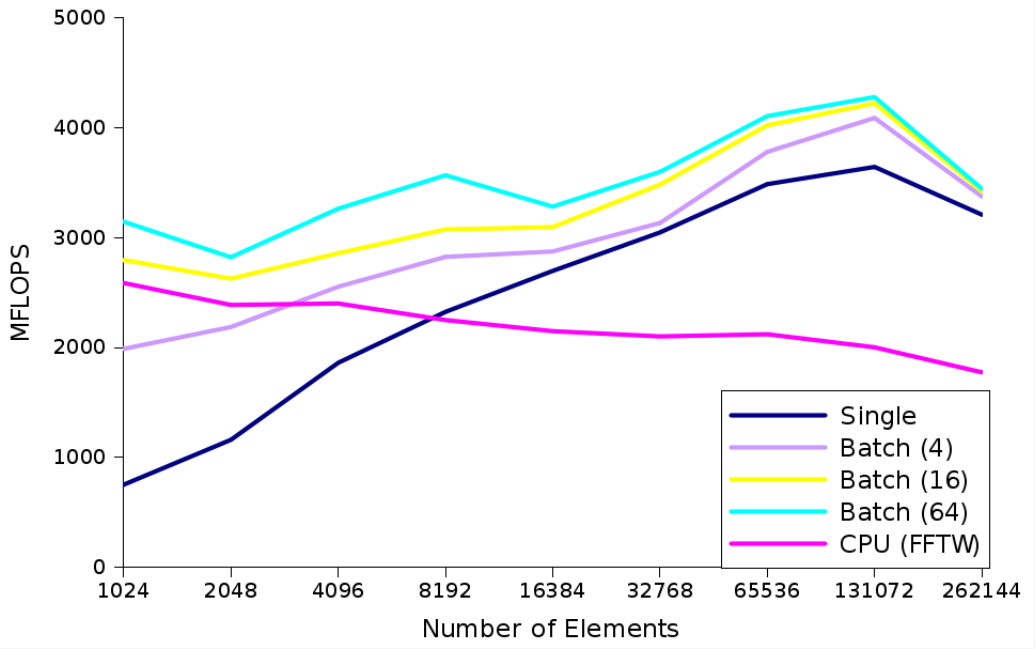

Figure 4.2: Performance vs Transform Size for 1D FFTs

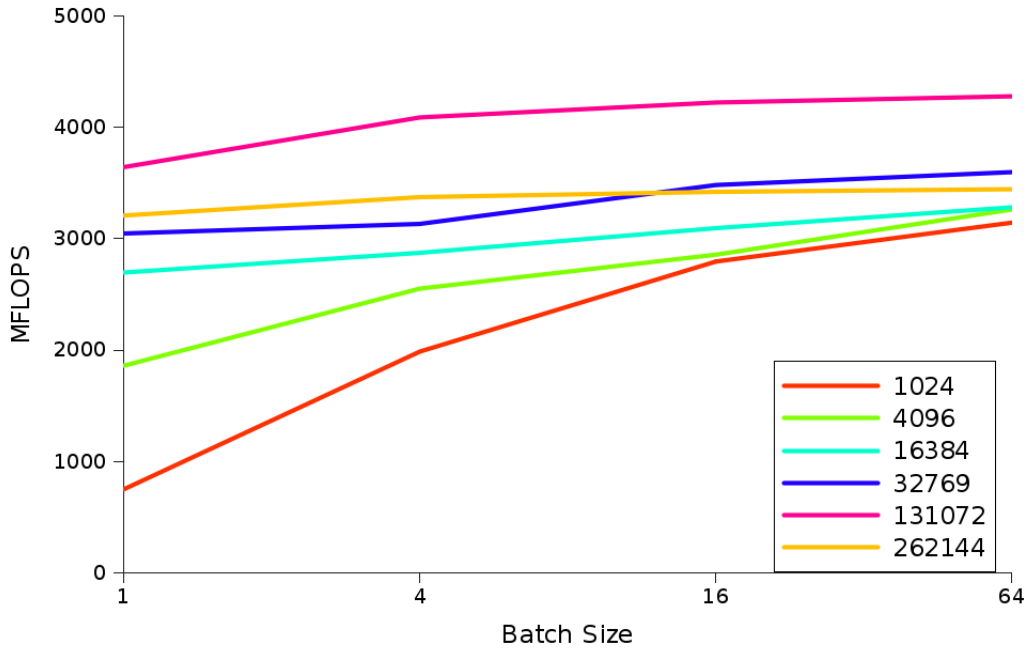

Figure 4.3: Performance vs Batch Size for 1D FFTs

- The 1D transform size at which the GPU FFT seems to be the most efficient is 131072 elements.
- There is almost no observed difference between the performance of the in-place and out-of-place transforms. Since there is a spike at the end of the in-place plot it is possible that the difference will begin to show for larger transform sizes but again, these cannot be tested due to the limited video memory available.
- For the out-of-place transforms shown in Figure 4.4, the difference in performance between the FFTW and CUFFT is only 13% for a 1024 size transform, but that increases until it is nearly 300% faster for transforms of size 262144.

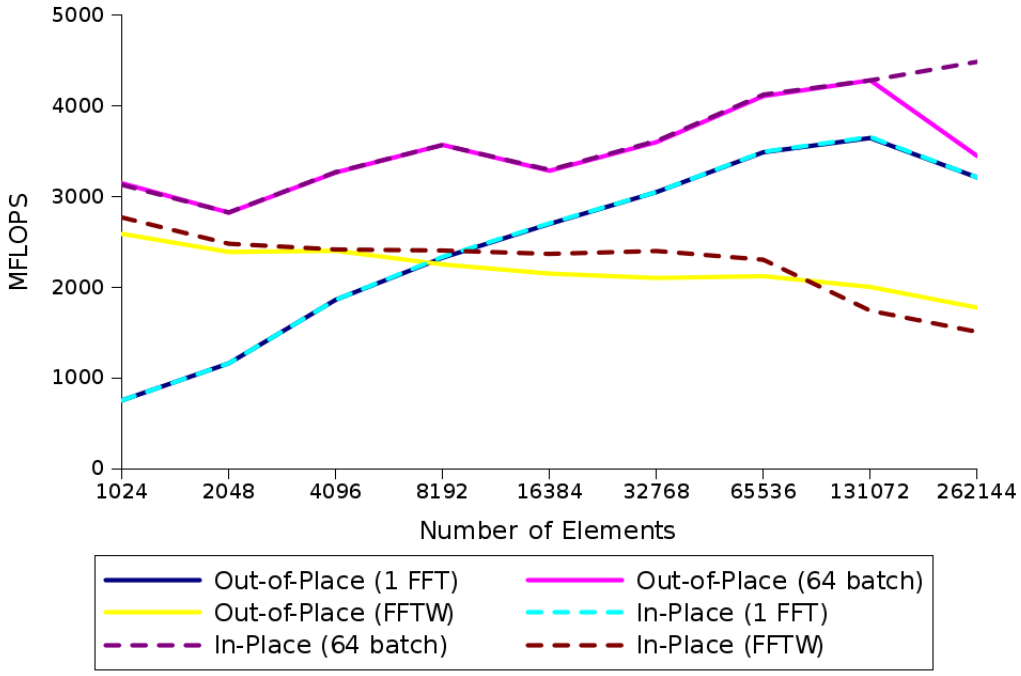

Figure 4.4: In-Place vs Out-of-Place for 1D FFTs

4.4.2 2D Transforms

For 2D graphs the maximum transform size specified by CUFFT is 16384x16384, but in testing the program crashed for transforms larger than 1024x1024. Thus I tested transforms from 8x8 up to 1024x1024 in order to obtain sufficient data. As can be seen from Figure 4.7, the effect of the high-latency memory accesses is very pronounced with low transform sizes but the performance increases exponentially as the size increases. At the same time the CPU's performance decreases roughly linearly.

There was very little difference in performance between the in-place and out-of-place versions of the algorithm for the complex-to-complex FFT, but complex-to-real FFTs showed a difference in performance for larger transforms.

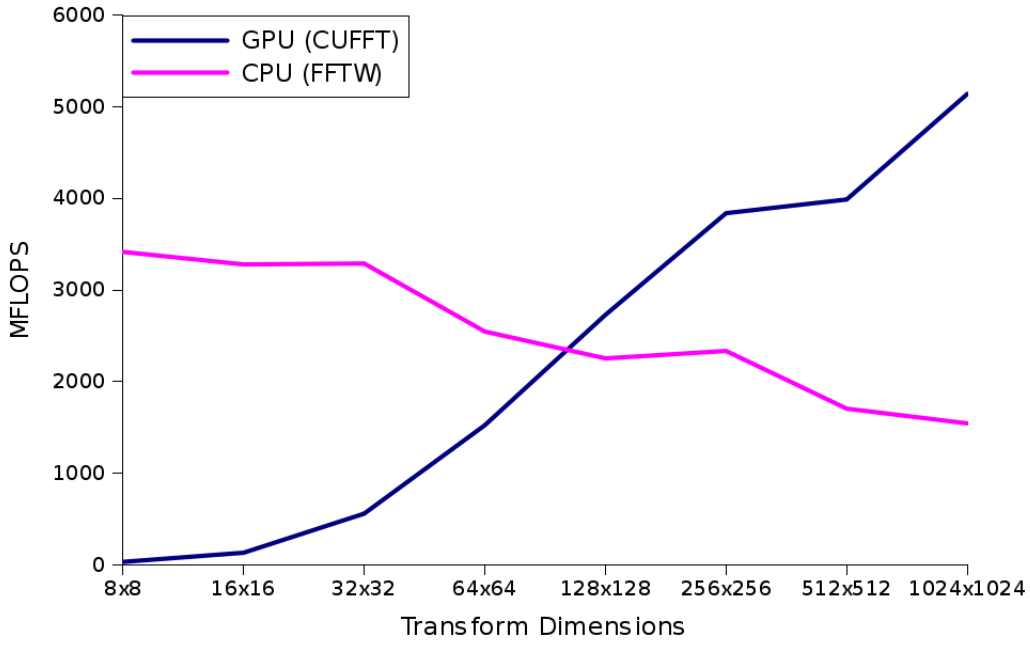

Figure 4.5: Performance vs Transform Size for 2D FFTs

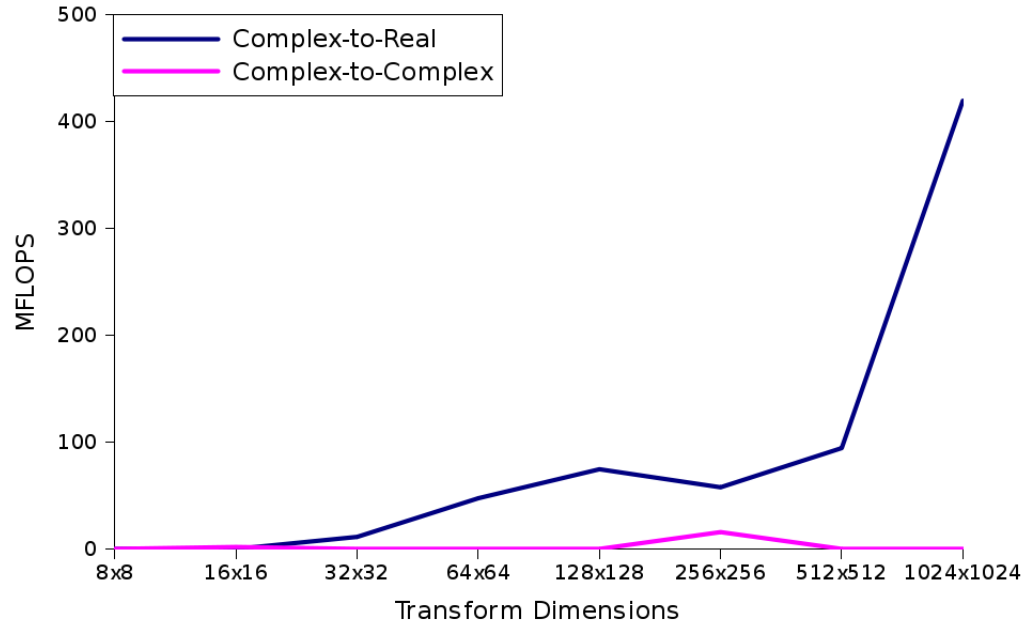

Figure 4.6: Increase in Performance Offered by Out-of-Place Algorithms for 2D FFTs

4.4.3 3D Transforms

3D transforms, like 2D ones, have a size limit of 16384 in all dimensions, but again this limit could not be reached due to limited memory on the GPU. Therefore, for testing, power-of-two transform sizes from 2x2x2 to 128x128x128 were used. We observe a similar trend of performance increases to the 2D implementation. Once again, the complex-to-complex transform exhibits no performance improvement when using the in-place or

out-of-place algorithms.

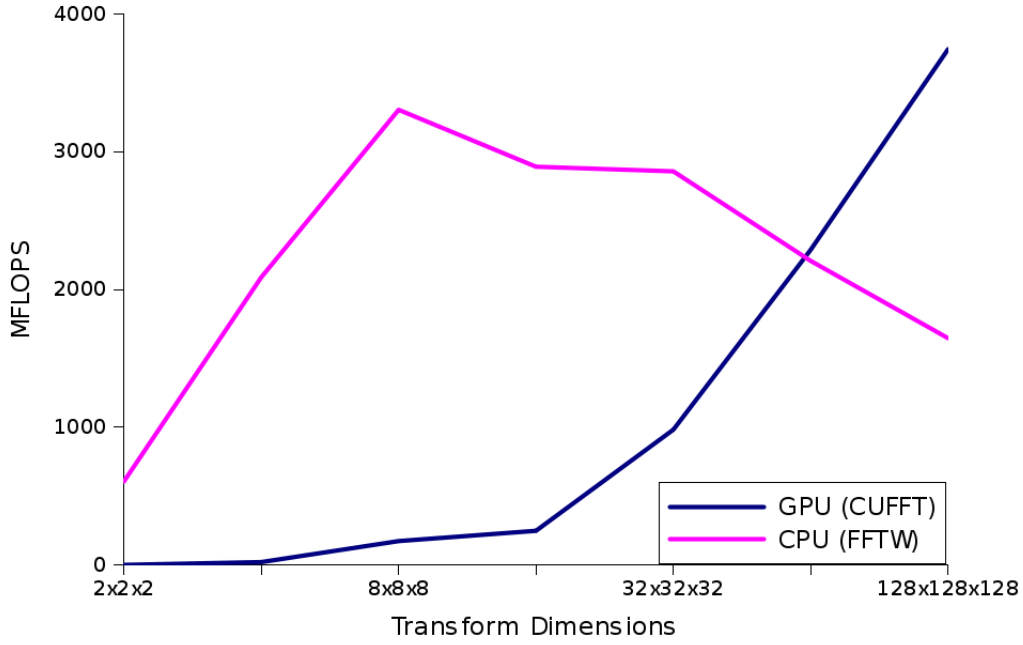

Figure 4.7: Performance vs Transform Size for 2D FFTs

4.5 Summary of Results

Measurement	1D FFTs (64 batched)	2D FFTs	3D FFTs
Minimum Performance (MFLOPS)	2821.41	36.92	1.76
Maximum Performance (MFLOPS)	4935.52	6081.74	3743.42
Average Performance (MFLOPS)	3899.54	2478.63	1066.11
Performance per Watt[1] (MFLOPS/W)	69.51	85.66	52.72
Performance per Rand (MFLOPS/Rand)	2.66	3.27	2.01
Sum of the Errors	-1.387x10^{-6}	1.022x10^{-6}	1.519x10^{-5}

Table 4.3: GPU FFT Performance and Accuracy Summary

Measurement	1D FFTs	2D FFTs	3D FFTs
Minimum Performance (MFLOPS)	1507.39	1608.78	607.78
Maximum Performance (MFLOPS)	2767.66	3466.74	3437.39
Average Performance in MFLOPS	2226.51	2586.40	2222.25
Performance per Watt[2] (MFLOPS/W)	38.98	48.83	48.41
Performance per Rand (MFLOPS/Rand)	1.80	1.87	1.85

Table 4.4: CPU FFT Performance and Accuracy Summary

Chapter 5

Digital Filters on the GPU

In this chapter, I will provide a brief overview of the digital filtering process and key concepts. Following this, I will consider the application of GPUs to simple digital filters. FIR and IIR filters will be implemented on the CPU and GPU and the performance will be measured. In addition, the filter output will be analysed for errors caused by the limited precision on the GPU.

5.1 Brief Overview of Digital Filters

I will now provide a short introduction to the topic of digital filtering with a special focus on what will be tested. For a more in-depth treatment of the topic , refer to [34] and [35].

5.1.1 Digital Filters

Filters are signal processing elements that remove unwanted parts of a signal or extract useful parts, such as those values lying within a certain range of frequencies [36]. Digital filters are a class of filters that perform the task of filtering by performing numerical calculations on sampled values of a signal.

The input analogue signal is converted into a digital one by an analogue-to-digital converter (ADC). The numerical operations normally involve multiplying the signals with some constant and adding the results. The filtered signal is then converted back to an analogue signal by a digital-to-analogue converter (DAC). This process is shown in Figure 5.1.

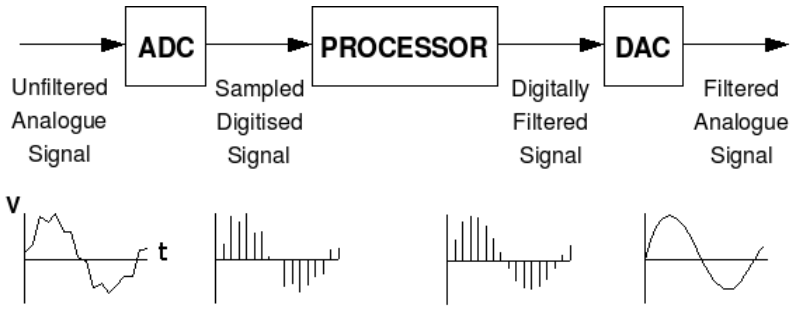

Figure 5.1: The Digital Filtering Process [36]

5.1.2 Recursive and Non-recursive Filters

Filters that are calculated solely from current and previous inputs are called non-recursive filters while those that accept previous outputs as inputs to the filter are recursive. The filter shown in 5.1 is an example of a non-recursive filter. Equation 5.2 is a simple recursive filter.

$$y_n = x_n + x_{n-1} \tag{5.1}$$

$$y_n = x_n + y_{n-1} \tag{5.2}$$

Recursive filters are also called infinite impulse response filters because the output to a unit impulse theoretically continues infinitely whereas non-recursive filters are called finite impulse response because their output to a unit impulse is of finite duration.

An advantage of using an IIR filter is that it can achieve a given frequency response characteristic using much less memory and involving less calculations. On the other hand, IIR filters tend to be more unstable than FIR filters due to their recursive nature and errors in calculation are carried through to all future calculations.

5.1.3 Filter Properties

Order of the Filter

The order of a digital filter is the largest number of previous inputs or outputs that are used to calculate the current output. This may be any positive integer. The FIR filter represented by equation 5.3 is a first order filter as the output y depends on the input at n as well as the input at $n-1$. Similarly equation 5.4 represents a first order IIR filter since the output depends on the input at n and the output at $n-1$. In general, the larger the filter order, the better the frequency magnitude response performance of the filter.

Filter Coefficients

The first-order IIR and FIR filters can be written in the general form shown in equations 5.3 and 5.4.

$$y_n = a_0 x_n + a_1 x_{n-1} \tag{5.3}$$

$$b_0 y_n + b_1 y_{n-1} = a_0 x_n + a_1 x_{n-1} \tag{5.4}$$

In filter 5.3, the constants a_0 and a_1 are called the filter coefficients and these determine the characteristics of a particular filter. For example, if a_0 and a_1 are both $1/2$, it is a two-term average filter. For FIR filters, the filter coefficients are the impulse response of the filter.

The general form of the IIR filter 5.4 has a symmetrical form and the coefficients are denoted by b_0 and b_1.

Transfer Function

The transfer function of a digital filter allows us to describe the filter completely with a compact expression. We apply the delay operator denoted by z^{-1} to the symmetrical form of the filter to get the transfer function as shown in equation 5.5.

$$\begin{aligned} (b_0 + b_1 z^{-1}) y_n &= (a_0 + a_1 z^{-1}) x_n \\ \therefore \frac{y_n}{x_n} &= \frac{a_0 + a_1 z^{-1}}{b_0 + b_1 z^{-1}} \end{aligned} \tag{5.5}$$

The transfer function for non-recursive filters simply makes the coefficient $b_0 = 1$ and all other b coefficients are zero. Also, for higher-order filters, there will be terms for $a_n z^{-n}$ and $b_n z^{-n}$ where n is the order.

5.2 Testing Methodology

I will be testing simple FIR and IIR time-domain filters represented by the Direct Form-I filter structure diagrams in Figures 5.2 and 5.3.

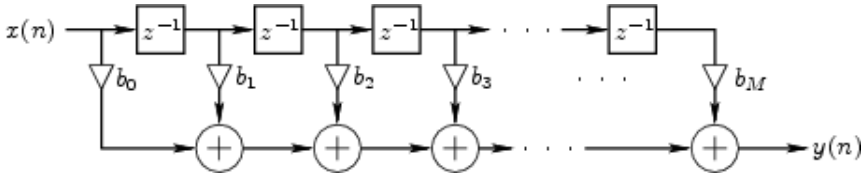

Figure 5.2: Direct Form-I FIR Filter Structure

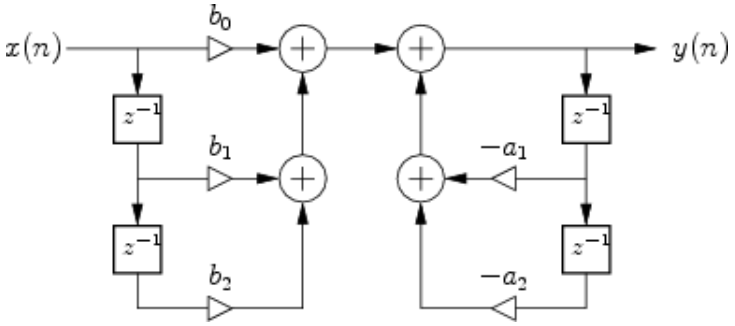

Figure 5.3: Direct Form-I IIR Filter Structure

The output is tested when the input signal is:

- an impulse response
- a unit step function
- randomly generated numbers

For the tests all coefficients are set to 1. The time taken to allocate memory and set up the arrays for processing is not taken into account, only the processing time will be measured. The gettimeofday() function will be used to measure time when calculating on the CPU and cutTimer will be used on the GPU.

To measure precision, the filter output will be compared with the reference CPU implementation. The difference between the two will be calculated. It is expected that the single precision nature of GPUs will have little effect on the output for most FIR filters. This is because errors due to rounding in one stage of the filter are not carried over to the next stage.

IIR filters on the other hand use the output of previous stages as inputs to the next. This has the effect of amplifying the errors and causing the output to be inaccurate at best.

5.3 Finite Impulse Response Test Results

The results of testing the FIR filter are shown in the table below and the discussion follows.

Number of Taps	CPU Processing Time (s)	GPU Processing Time (s)
50	$1x10^{-6}$	$1x10^{-3}$
100	$2x10^{-6}$	$3x10^{-3}$
500	$2x10^{-6}$	0.461
1000	$4x10^{-6}$	1.281
2500	$9x10^{-6}$	7.375
5000	$19x10^{-6}$	32.006
10000	$37x10^{-6}$	128.001

Table 5.1: Results of Performance Test

It must be stated that the implementation was not optimised for the parallel nature of the GPU As can be seen, the graphics card actually performed several orders of magnitude worse than the CPU for the tests run. The CPU implementation offered far better performance even for a large number of taps. However, accuracy was found to be sufficient.

5.4 Infinite Impulse Response Filters

IIR filter implementation and testing could not be conducted due to time constraints. This was due to the delay in hardware delivery, as it was recieved only two weeks prior to the submission deadline.

However the following is expected from an IIR filter implemented on the GPU:

- Increased performance on modern graphics cards due to the presence of the stream output which allows data in the geometry shader of the pipeline to be read out and fed back in to the input assembler to be reprocessed. IIR filters operate on similar feedback loops and their data need not leave the processing pipeline.
- Errors due to rounding (caused by the lack of double precision floating point capability) will be carried forward from previous stages. This will lead to the errors being magnified as the number of filter taps increase.

Chapter 6

Conclusions

The main goals of this thesis as stated in Chapter 1 were to:

- Investigate the existing libraries and tools that may be used to execute signal processing primitives on a graphics card.
- Test an existing GPU-based FFT library and compare the performance and accuracy of that implementation with a reference CPU-based implementation, varying the transform size and number of dimensions.
- Implement simple FIR and IIR filters on the GPU and compare the performance and accuracy of the filter's output for different input signals.

After carrying out an investigation into the nature of the GPU and the various methods of programming it I decided to use NVIDIA's CUDA as the tool for the second half of my thesis. This is the actual implementation of signal processing primitives. The implementation and testing went smoothly for the most part except that I was unable to implement the IIR filter due to time constraints.

6.1 Conclusions from Testing

Based on the test results of the FFT and the FIR filter, the following conclusions can be made.

The GPU-based FFT implementation offered massive performance increases of upto 300% over the CPU for some datasets. The general trend was that the CPU was more efficient at performing single FFTs on small datasets but as the transform size or number of FFTs being processed at a time increased, the GPU offered better performance.

Performance per watt and per Rand was again dependent on the transform sizes. However if the highest performance value obtained from testing was used as a measure of the maximum capability, the GPU far outstrips the CPU. It offers nearly double the performance per watt and 43% increase in performance per Rand.

An interesting point to note is that the performance was generally higher for 2D data. This is because GPUs are optimised to operate on textures, which are basically 2D arrays of pixels. Therefore even when implementing non-graphics related algorithms on the GPU, it is best to convert the input data into one or several rectangular subsets.

The implementation of the FIR filter on the GPU was successful but offered unsatisfactory performance. This is due to the efficiency of the algorithm used. It was implemented using a nested `for` loop over 1D data, which is computationally more efficient on the CPU than the GPU.

GPUs are optimised to perform a small kernel function on a large dataset in parallel and not sequentially as in a `for` loop. Additionally, the algorithm did not implement the efficient caching strategy discussed in Section 2.4.2 to minimise memory accesses. If the algorithm were to be optimised properly, it would be possible to speed up the implementation and make it faster than the CPU for filters with a large number of taps.

6.2 Recommendations for Further Research

In light of the findings presented above, there are several possible avenues for further research that could provide interesting and useful information or tools.

Firstly, the FIR filter that was inefficient in my implementation could be sped up with a more cache and memory aware algorithm. Another recommendation would be to apply the primitives investigated here into a larger system such as a software radio implemented on the GPU using CUDA.

One could also investigate an alternative toolkit such as RapidMind to test the FFT and filters on and compare the performance of the two implementations.

To gain a better understanding of what it is that affects the real-world performance of these signal processing algorithms, tests similar to the ones used in this thesis should be run on other graphics cards. In particular AMD GPUs with their associated CTM toolkit which is analogous to the CUDA toolkit for NVIDIA GPUs would be interesting to investigate. These GPUs have 320 stream processing units which would improve performance.

Appendix A

A Short History of Graphics and GPUs

The history of computer graphics can be traced back to a paper presented by Ivan Sutherland in 1963 entitled 'Sketchpad' at the Summer Joint Computer Conference [37]. It allowed for interactive design on a vector graphics display with a light pen. Since this landmark paper, graphics have come a long way through the efforts of hundreds of different people.

A.1 The 1960s

During the middle to late 1960's, several important advances took place in the fledgling field of graphics. A method for drawing lines and circles on a raster device was discovered and the first computer aided design (CAD) concepts were developed. Meanwhile on the hardware side, flight simulators with real-time raster graphics were created and the first microprocessor was built at Intel.

A.2 The 1970s & 1980s

The early 1970's saw the discovery the z-buffer algorithm and texture mapping. During the latter part of that decade, early games such as Pong and Pac Man became popular and can be viewed as the starting point for the current success of graphics cards due to the gaming industry.

The 1980's signaled the advent of the personal computer (PC) with competing models from both IBM and Apple. The first Video Graphics Array (VGA) card was also invented and research interest in special purpose graphics processors and parallel processors was on the rise.

A.3 The 1990s

The 1990's was arguably the most important decade in the brief history of computer graphics. Both the software and hardware really started to mature and became viable for mass consumption. CPUs alone could no longer handle the massive amounts of data that needed to be processed at the necessary speeds for real-time usage, thus the standalone graphics card was born.

Some of the highlights of the first part of the decade were the introduction of OpenGL as a standard for graphics APIs, the Reality Engine from Silicon Graphics that allowed hardware-based real-time texture mapping and the release of the Nintendo 64 game console that provided real-time 3D graphics to the masses.

Towards the latter part of the 1990's the personal computer market exploded. Companies like 3dfx, ATI, Matrox and NVIDIA all jumped into the ring to claim their slice of the ever-expanding 3D graphics card market in both the personal and business sectors. Microsoft's Direct3D API rose as a challenger to OpenGL on the Windows operating system. Movies began to use 3D technology extensively for special effects and several wholly 3D rendered movies were made.

On the hardware side, ever increasing clock speeds, bit widths and memory capacities pushed the available technology to its limits. It was at this time that consumer graphics cards began using multiple pipelines to speed up processing. Graphics cards also switched from using the PCI bus to the specialized AGP bus.

A.4 The 2000s

That brings us to the current decade, the 2000's. Performance has continued to increase, doubling every year, while computer graphics have become ubiquitous. Almost all modern computers have some rudimentary form of graphics processor. This has led to better tools for creating graphics as well as giving programmers more control over the graphics pipeline. Programmable vertex, pixel and geometry shaders have opened up the GPU's pipelined architecture, giving programmers more control and the new PCI Express bus has allowed for increased parallelism when accessing the system bus.

Appendix B

The Future of GPGPU

We are currently at a critical juncture in the development of both CPUs and GPUs. With the slow convergence of these two kinds of processor, the question remains open as to the future of both. Is the processor of the future a massively parallel CPU a la the Cell stream processor from IBM that can handle GPU tasks? Or is it a GPU that can provide better cache performance and support for branching? When we get right down to it, is there really any difference between these two at all?

These questions will be answered in time. One thing is for sure, the era of general-purpose computing on GPUs is just beginning. In this chapter I will provide an overview of the currently announced GPGPU initiatives from the major players in both the CPU and GPU arenas.

B.1 NVIDIA Tesla GPU Computing

NVIDIA has just launched a new product line in addition to their existing Quadro enterprise line and GeForce consumer brand. They will enter the supercomputer market and offer high performance computing (HPC) machines under the Tesla moniker [38].

These solutions will consist of graphics cards using a modified version of the G80 core that powers the GeForce 8800 Ultra. Each of these has 128 processors in it's computing core, 1.5 GB of GDDR3 video memory using a 384-bit wide memory interface and is capable of more than 500 GFLOPS of floating point performance. Initially there will be three products in this line:

- the Tesla C870 Computing Processor is the 'entry level' model. It costs $1,499 and consists of one 8-series GPU. This graphics card uses a maximum of 170W of power can be connected to any PC to create a personal supercomputer.
- the Tesla D870 Deskside Supercomputer consists of two of the 8-series GPUs and attaches to a workstation via a PCI Express adapter. It uses 520W of power at maximum load and will retail for $7,500.

- the Tesla S870 Computing Server is designed to fit into enterprise server clusters and consists of four of the 8-series GPUs. It scales in performance by simply adding more of these units. It requires a maximum of 800W and costs $12,000.

This product line is aimed at more technical users such as scientists and engineers. Additionally it has applications in industrial IT installations, weather prediction and graphical visualisation for oil and gas exploration.

A big advantage of this product line is the ability to use of the CUDA framework to easily program these massively multi-threaded processors. It would be extremely difficult to effectively hand code an optimum algorithm for more than 128 processors.

If there is a flaw in this product, however, it must be the lack of full IEEE 754 double-precision. This will limit its application to some scientific applications. However since the next line of consumer GPUs from NVIDIA is expected to support double-precision, it seems likely that the next range of Tesla products will include this as well.

B.2 AMD Fusion

When AMD bought ATI, the possibility of a unified CPU and GPU suddenly became very real. This was confirmed by AMD when they announced the AMD Fusion line of heterogeneous multicore microprocessors. This architecture will combine GPU and the CPU into one, where those tasks that involve parallel computing will be automatically assigned to the GPU part of the processor and other general processing will take place on the CPU part [39]. This will greatly simplify the programming model that application developers have to deal with and allow them to create applications faster and easier.

A much touted aspect of this architecture is its very low thermal display power (TDP), which can be as low as 10W. This will allow these processors to be incorporated into handheld devices as well. For GPGPU and graphics processing in general, a major advantage of the Fusion architecture is faster transfer of data to and from the CPU as well as faster memory accesses.

B.3 Intel's "Larrabee" Project

"Larrabee" is the code name for a new many-core architecture being developed at Intel [40, 41].The term 'many-core' implies that the processor will have more than eight cores on one die. The existence of this architecture was just a rumour until very recently, but it has since been confirmed by several sources within Intel. There is still very little information about it, however.

That said, I will briefly state what is known about this project. The Larrabee architecture is designed to be the basis of a new generation of CPUs, GPUs and HPC systems from

Intel. The way it manages to be both these at once is by including a 'fixed-function unit', which could be graphics related or not depending on the specific product being designed.

Larrabee will have 16 to 24 cores that will each have their own L1 cache and support four execution threads at once. These cores will implement a subset of the current x86 instruction set and will include some GPU-specific extensions. There will also be a 512-bit wide fragment processing unit capable of processing vectors of floating points as well as loops and branches. Finally there will also be a memory controller and a shared L2 cache.

These processors will not be available till 2009 at the earliest, however, so a lot could change about them before they are fully realised.

Appendix C

Hardware Specifications

C.1 Graphics Processing Unit

Table C.1 shows the the general technical specifications of the GeForce 8600 GTS being tested. Some points to note from these specifications are:

- The 32 Stream Processors (SPs) on the GeForce 8600 GTS is one-third the number present on NVIDIA's flagship GeForce 8800 model GPU which has 96 SPUs. This effectively means that the number of simultaneous operations possible are far less and execution performance will increase by 2-3 times when run on the GeForce 8800.
- The DDR memory bus is only 128-bits wide. This is a serious bottleneck especially given the 2000MHz DDR3 memory it is connected to. Higher-end models have a 256-bit or 512-bit wide bus and thus memory accesses would be less of an issue.

Functional Parameter	Value
Core Code	G84
Transistor Count	289 million
Manufacturing Process	0.08 microns
Core Clock	675MHz
Vertex Shaders	32 Stream Processors (@ 1450MHz)
Rendering (Pixel) Pipelines	32 Stream Processors (@ 1450MHz)
Pixel Shader Processors	32 Stream Processors (@ 1450MHz)
Texture Filtering (TF) units	16
Raster Operator units (ROP)	8
Memory Clock	2000MHz DDR3
DDR Memory Bus	128-bit
Memory Bandwidth	32.0GB/s
PCI Express Interface	x16
Rated Thermal Design Power	71W
Price	ZAR 1857.90

Table C.1: GeForce 8600GTS Technical Specifications[42]

Table C.2 shows detailed specifications of the NVIDIA GeForce 8600GTS. There are 4 multiprocessors in this GPU and each multiprocessor is composed of eight processors, so that a multiprocessor is able to process the 32 threads of a warp in four clock cycles.

Functional Parameter	Value
Hardware Revision Number	1.01
Total amount of global memory	256MB
Number of Multiprocessors	4
Total amount of constant memory	64KB
Total amount of shared memory per block	16KB
Total number of registers available per block	8192
Warp size	32 threads
Maximum number of threads per block	512 threads
Maximum sizes of each dimension of a block	512 x 512 x 64
Maximum sizes of each dimension of a grid	65535 x 65535 x 1
Maximum memory pitch	256KB
Texture alignment	256 bytes
Clock rate	1458000 KHz

Table C.2: GeForce 8600GTS Detailed Specifications

The graphics card is connected to the motherboard via a PCI Express (PCIe) bus which offers improved transfer rates than was achieved using the older Accelerated Graphics Port (AGP) bus. The bandwidths achieved for transferring data to and from the GeForce 8600GTS are shown in Table C.3 along with transfer speeds within the device. A 32MB file was used in the tests.

Test	Pageable Memory (MB/s)	Pinned Memory (MB/s)
Host to Device	2098.3	2552.9
Device to Host	1582.7	1936.8
Device to Device	19374.6	19385.1

Table C.3: Memory Bandwidth Test

C.2 Central Processing Unit

Table C.4 shows the specifications of the Intel Core 2 Duo processor that the GeForce 8600GTS will be tested against.

Functional Parameter	Value
CPU core	Conroe
CPU socket	Socket 775
Clock frequency	2.33 GHz
Frontside Bus Speed	1333 MHz
L2 cache	4096 KB
Cores	2
MMX/SSE/SSE2/SSE3	Yes
Fabrication process	65 nm
TDP (Max. power)	65 W
Core Voltage	0.85V – 1.5V
Price	ZAR 1537.80

Table C.4: Intel Core 2 Duo E6550 Specifications[43]

Appendix D

FFT Test Results

Here are the detailed results of the tests run for computing the FFT. All tests were done using pinned memory as that was faster than pageable memory (see Table C.3). The results are all in MFLOPS (millions of FLOPS).

D.1 1D Transforms

D.1.1 Complex-to-Complex

Transform Size	Single	Batch (4)	Batch (16)	Batch (64)	CPU (FFTW)
1024	752.90	1988.30	2795.88	3144.70	2587.45
2048	1161.21	2187.13	2627.10	2821.41	2387.64
4096	1861.63	2553.20	2857.45	3263.20	2401.05
8192	2324.81	2824.81	3074.54	3567.08	2250.38
16384	2697.46	2874.08	3095.43	3282.37	2149.21
32768	3047.42	3133.44	3482.42	3597.93	2101.52
65536	3486.52	3780.12	4021.29	4105.18	2121.19
131072	3643.15	4088.79	4222.48	4278.19	2002.24
262144	3208.64	3373.79	3420.98	3444.57	1774.86

Table D.1: Performance of 1D Complex-to-Complex In-Place FFTs in MFLOPS

Transform Size	Single	Batch (4)	Batch (16)	Batch (64)	CPU (FFTW)
1024	752.90	1988.30	2805.45	3126.68	2767.66
2048	1161.21	2197.83	2627.10	2823.66	2478.29
4096	1861.63	2553.20	2861.63	3267.87	2415.24
8192	2334.92	2847.17	3079.86	3566.55	2403.66
16384	2704.34	2874.08	3094.28	3291.55	2366.59
32768	3047.42	3140.81	3497.16	3610.21	2398.25
65536	3491.76	3785.36	4042.26	4120.90	2302.23
131072	3654.29	4099.93	4233.63	4278.19	1740.91
262144	3208.64	3350.20	3420.98	4482.66	1507.39

Table D.2: Performance of 1D Complex-to-Complex Out-of-Place FFTs in MFLOPS

D.1.2 Real-to-Complex

Transform Size	Single	Batch (4)	Batch (16)	Batch (64)	CPU (FFTW)
1024	839.32	2438.09	3828.02	4615.17	2571.93
2048	1325.10	2833.68	3611.69	3954.34	2570.90
4096	2275.49	3377.97	3908.57	4122.62	2317.15
8192	2974.43	3789.66	4219.37	4534.07	2225.43
16384	3203.24	3283.52	3303.01	3494.54	2113.77
32768	3814.20	3740.47	3836.31	3973.94	2040.42
65536	4435.48	4199.55	4445.96	4540.33	2000.32
131072	4645.85	4534.44	4679.27	4723.83	1912.91
262144	3656.91	3680.50	3704.09	3704.09	1806.77

Table D.3: Performance of 1D Real-to-Complex In-Place FFTs in MFLOPS

Transform Size	Single	Batch (4)	Batch (16)	Batch (64)	CPU (FFTW)
1024	839.32	2528.36	4055.40	4804.66	2676.54
2048	1294.68	2851.59	3611.69	3967.41	2631.41
4096	2296.63	3366.42	3908.57	4105.42	2442.11
8192	2990.94	3782.74	4179.97	4491.47	2238.46
16384	3212.41	3382.15	3418.85	3640.20	2264.47
32768	3826.48	3865.80	3986.23	4136.14	2235.13
65536	4430.23	4351.59	4645.19	4744.81	2233.41
131072	4656.99	4701.55	4879.81	4935.52	2139.42
262144	3656.91	3774.87	3822.06	3822.06	1578.18

Table D.4: Performance of 1D Real-to-Complex Out-of-Place FFTs in MFLOPS

D.1.3 Transform Times

Transform Size	Complex-to-Complex	Real-to-Complex	CPU
1024	0.03	0.03	0.019
2048	0.05	0.05	0.048
4096	0.06	0.06	0.101
8192	0.11	0.11	0.237
16384	0.22	0.24	0.537
32768	0.42	0.44	1.17
65536	0.76	0.83	2.46
131072	1.59	1.71	5.54
262144	4.92	5.15	13.17

Table D.5: Time in Milliseconds to Compute a 2D FFT (Excluding Memory Access)

D.2 2D Transforms

D.2.1 Complex-to-Complex

Transform Size	GPU (In-Place)	CPU (In-Place)	GPU (Out-of-Place)	CPU (Out-of-Place)
8x8	36.92	3466.74	36.92	3415.6
16x16	134.73	2919.36	136.53	3279.5
32x32	562.64	3391.63	562.64	3289.43
64x64	1526.42	2633.85	1526.42	2547.59
128x128	2729.57	2102.11	2729.57	2255.15
256x256	3822.06	2223.84	3837.79	2336.25
512x512	3987.21	2086.84	3987.21	1707.13
1024x1024	5138.02	1608.78	5138.02	1546.75

Table D.6: Performance of 2D Complex-to-Complex In-Place and Out-of-Place FFTs in MFLOPS

D.2.2 Complex-to-Real

Transform Size	GPU (In-Place)	CPU (In-Place)	GPU (Out-of-Place)	CPU (Out-of-Place)
8x8	36.92	2587.51	36.92	2517.04
16x16	179.64	3086.38	179.64	3257.38
32x32	752.90	3340.21	764.16	3259
64x64	1683.21	2943.23	1730.64	2939.33
128x128	3229.61	2570.5	3304.16	2515.51
256x256	4445.96	2412.28	4503.63	2342.78
512x512	5025.30	2113.66	5119.67	1994.53
1024x1024	5662.31	1895.41	6081.74	1795.26

Table D.7: Performance of 2D Complex-to-Real In-Place and Out-of-Place FFTs in MFLOPS

D.2.3 Transform Times

Transform Size	Complex-to-Complex	Real-to-Complex	CPU
8x8	0.03	0.03	$863x10^{-6}$
16x16	0.05	0.03	0.003
32x32	0.05	0.03	0.015
64x64	0.09	0.07	0.102
128x128	0.21	0.14	0.530
256x256	0.62	0.43	2.34
512x512	3.32	2.10	11.07
1024x1024	11.41	9.32	63.52

Table D.8: Time in Milliseconds to Compute a 2D FFT (Excluding Memory Access)

D.3 3D Transforms

D.3.1 Complex-to-Complex

Transform Size	GPU (In-Place)	CPU (In-Place)	GPU (Out-of-Place)	CPU (Out-of-Place)
2x2x2	1.76	607.78	1.76	653.08
4x4x4	22.07	2088.88	21.82	2153.23
8x8x8	174.53	3303.85	173.21	3437.39
16x16x16	249.20	2890.8	248.71	2824.08
32x32x32	983.04	2857.31	985.50	2834.68
64x64x64	2288.52	2207.45	2288.52	1950.73
128x128x128	3743.42	1646.59	3743.42	1655.61
256x256x256	-	1741.01	-	1797.88

Table D.9: Performance of 3D Complex-to-Complex In-Place and Out-of-Place FFTs in MFLOPS

D.3.2 Transform Times

Transform Size	In-Place (C2C)	Out-of-Place (C2C)	CPU
2x2x2	0.04	0.04	$187x10^{-6}$
4x4x4	0.06	0.06	$813x10^{-6}$
8x8x8	0.10	0.10	0.007
16x16x16	0.93	0.93	0.094
32x32x32	2.11	2.11	0.868
64x64x64	7.71	7.73	10.53
128x128x128	41.17	41.19	140.93

Table D.10: Time in Milliseconds to Compute a 3D FFT (Excluding Memory Access)

Bibliography

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *The Eurographics Association*, vol. 26, no. 1, pp. 80–113, 2007.

[2] K. Moreland and E. Angel, "The FFT on a GPU," in *SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware*, M. Doggett, W. Heidrich, W. Mark, and A. Schilling, Eds., Eurographics Association. San Diego, California,: Eurographics Association, 2003, pp. 112–119.

[3] NVIDIA, *NVIDIA CUDA Programming Guide Version 1.0*, June 2007, chapter 1, Figure 1.2 and Figure 1.3.

[4] American Micro Devices, Inc. (2007) ATI Radeon HD 2900 - GPU Specifications. [Online]. Available: http://ati.amd.com/products/Radeonhd2900/specs.html

[5] NVIDIA. (2007) GeForce 8800. [Online]. Available: http://www.nvidia.com/page/geforce_8800.html

[6] B. Solca. (2007) NVIDIA's G92 Joins the 1 Teraflops Club - Three times faster than the current G80. [Online]. Available: http://news.softpedia.com/news/NVIDIA-039-s-G92-Joins-the-1-Teraflops-Club-55548.shtml

[7] Wikipedia, the free encyclopedia. (2007) OpenGL. [Online]. Available: http://en.wikipedia.org/wiki/OpenGL

[8] ——. (2007) Direct3D. [Online]. Available: http://en.wikipedia.org/wiki/Direct3D

[9] ——. (2007) Comparison of Direct3D and OpenGL. [Online]. Available: http://en.wikipedia.org/wiki/Comparison_of_Direct3D_and_OpenGL

[10] D. Salvator. (2001) ExtremeTech 3D Pipeline Tutorial. [Online]. Available: http://www.extremetech.com/article2/0,1697,9722,00.asp

[11] J. Guinot. (2006) 3D Pipeline Of SM3/DX9 GPUs. [Online]. Available: http://www.ozone3d.net/tutorials/gpu_sm3_dx9_3d_pipeline.php

[12] K. Akeley and P. Hanrahan. (2007) Lecture 2, The Graphics Pipeline. [Online]. Available: http://graphics.stanford.edu/cs448-07-spring/PipelineLecture

[13] J. Hoxley. (2005) An Overview of Microsoft's Direct3D 10 API. [Online]. Available: http://www.gamedev.net/reference/programming/features/d3d10overview/

[14] Microsoft Corporation. (2007) MSDN : Pipeline Stages (Direct3D 10). [Online]. Available: http://msdn2.microsoft.com/en-us/library/bb205123.aspx

[15] E. Angel and K. Moreland, *Fourier Processing in the Graphics Pipeline*. Boston, MA,: Kluwer Academic Publishers, 2004, pp. 95–110.

[16] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A Memory Model for Scientific Algorithms on Graphics Processors," in *Proc. of ACM SuperComputing*. ACM, Nov 2006.

[17] N. K. Govindaraju and D. Manocha, "Cache-Efficient Numerical Algorithms using Graphics Hardware," 2007.

[18] Wikipedia, the free encyclopedia. (2007) Graphics Processing Unit. [Online]. Available: http://en.wikipedia.org/wiki/Graphics_processing_unit

[19] NVIDIA. (2007) CUDA for GPU Computing. [Online]. Available: http://news.developer.nvidia.com/2007/02/cuda_for_gpu_co.html

[20] American Micro Devices, Inc. (2006) AMD "Close to Metal" Technology Unleashes the Power of Stream Computing. [Online]. Available: http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543 114147,00.html

[21] S. D. Toit. (2007) Rapidmind Reference and User Guide. [Online]. Available: https://developer.rapidmind.net/documentation/guide/

[22] I. Buck, T. Foley, D. Horn, J. Sugerman, P. Hanrahan, M. Houston, and K. Fatahalian. (2007) BrookGPU. [Online]. Available: http://graphics.stanford.edu/projects/brookgpu/

[23] Wikipedia, the free encyclopedia. (2007) Cg (programming language). [Online]. Available: http://en.wikipedia.org/wiki/Cg_%28programming_language%29

[24] ——. (2007) High Level Shader Language. [Online]. Available: http://en.wikipedia.org/wiki/High_Level_Shader_Language

[25] ——. (2007) GLSL. [Online]. Available: http://en.wikipedia.org/wiki/GLSL

[26] M. Frigo and S. G. Johnson. (2005) The benchFFT Home Page. [Online]. Available: http://www.fftw.org/benchfft/

[27] M. Frigo and S. G. Johnson. (2006) FFTW 3.1.2 Documentation - Introduction. [Online]. Available: http://www.fftw.org/fftw3_doc/Introduction.html#Introduction

[28] Wikipedia, the free encyclopedia. GPGPU. [Online]. Available: http://en.wikipedia.org/wiki/GPGPU

[29] N. K. Govindaraju and D. Manocha. (2007) GPUFFTW - Home. [Online]. Available: http://gamma.cs.unc.edu/GPUFFTW/

[30] ——. (2007) GPUFFTW - Benchmarks. [Online]. Available: http://gamma.cs.unc.edu/GPUFFTW/results.html

[31] NVIDIA, *NVIDIA CUFFT Library 1.0*, June 2007.

[32] J. D. Owens, S. Sengupta, and D. Horn. (2005) Assessment of Graphic Processing Units (GPUs) for Department of Defense (DoD) Digital Signal Processing (DSP) Applications. [Online]. Available: http://www.ece.ucdavis.edu/cerl/techreports/2005-3/

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Polynomials and the FFT*. Cambridge, Massachusetts,: The MIT Press, 2001, pp. 822–848.

[34] L. R. Rabiner and B. Gold, *Theory and application of digital signal processing*. Englewood Cliffs, N.J., Prentice-Hall, Inc.,, 1975.

[35] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[36] dsptutor. (2007) Digital Filters: An Introduction. [Online]. Available: http://www.dsptutor.freeuk.com/dfilt1.htm

[37] W. Shoaff. (2000) A Short History of Computer Graphics. [Online]. Available: http://cs.fit.edu/ wds/classes/graphics/History/history/history.html

[38] NVIDIA. (2007) GPU Computing Solutions for HPC. [Online]. Available: http://www.nvidia.com/object/tesla_computing_solutions.html

[39] Wikipedia. (2007) AMD Fusion. [Online]. Available: http://en.wikipedia.org/wiki/AMD_Fusion

[40] Jon Stokes. (2007) Clearing up the confusion over Intel's Larrabee. [Online]. Available: http://arstechnica.com/articles/paedia/hardware/clearing-up-the-confusion-over-intels-larrabee.ars

[41] ——. (2007) Clearing up the confusion over Intel's Larrabee, part II. [Online]. Available: http://arstechnica.com/news.ars/post/20070604-clearing-up-the-confusion-over-intels-larrabee-part-ii.html

[42] Vincent Chang. (2007) NVIDIA GeForce 8600 GTS: The Full Review! [Online]. Available: http://www.hardwarezone.com/articles/view.php?cid=3&id=2237

[43] hardware.info. (2007) Intel Core 2 Duo E6550. [Online]. Available: http://www.hardware.info/en-US/productdb/bGRkZZiWmJTK/viewproduct/Intel_Core_2_Duo_E6550/

[44] D. Manocha and A. Sud. (2007) COMP 790-058: GPGP: General Purpose Computation using Graphics Processors. [Online]. Available: http://gamma.cs.unc.edu/GPGP/

[45] T. Monk. 7 Years of Graphics. [Online]. Available: http://accelenation.com/?ac.id.123.1

[46] J. Owens, "GPU Architecture Overview," in *Proc. of SIGGRAPH 2007*. SIGGRAPH 2007, 2007.

[47] Wikipedia, the free encyclopedia. (2007) Graphics Pipeline. [Online]. Available: http://en.wikipedia.org/wiki/Graphics_pipeline

[48] ——. (2007) Stream Processing. [Online]. Available: http://en.wikipedia.org/wiki/Stream_processing

[49] cclark and mfatica and cmorrison. (2007) CUFFT BENCHMARKING TOOL v1.0. [Online]. Available: http://forums.nvidia.com/index.php?showtopic=42482