University of Nevada, Reno

A Comparison of Massively Parallel Programming Models Through Applications in Sound Propagation and Jitter Measurement

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering

by

Torbjørn Panton Løken

Dr. Frederick C. Harris, Jr., Thesis Advisor

May, 2014

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

TORBJORN LOKEN

entitled

A Comparison Of Massively Parallel Programming Models Through Applications In Sound Propagation And Jitter Measurement

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Dr. Frederick C. Harris, Jr., Advisor

Dr. Sergiu M. Dascalu, Committee Member

Dr. Yantao Shen, Graduate School Representative

Marsha H. Read, Ph. D., Dean, Graduate School

May, 2014

Abstract

As the era of Moore's Law and increasing CPU clock rates nears its stopping point the focus of chip and hardware design has shifted to increasing the number of computation cores present on the chip. This increase can be most clearly seen in the rise of Graphic Processing Units (GPU) where hundreds or thousands of slower cores work in parallel to accomplish tasks. Programming for these chips represents a new set of challenges and concerns. Two of the APIs which have emerged present two different approaches to harnessing the computational power of the GPU. The first explored in this thesis, CUDA, applies a design philosophy similar to the C programming language, low level and high speed. The second discussed in this thesis, Thrust, is modeled after the design philosophy behind the C++ Standard Template Library, low cost abstractions and high flexibility. This thesis explores applications which use these programming models in the quest to understand the reasons behind their use.

Dedication

For Mama, Pappa and Kamille.

Acknowledgments

I would like to thank the following people for their invaluable help and patience as I completed my work:

Roger Hoang, thank you for the advice, teaching and friendship. We've sundered this cabal.

Tantin, thanks for the many hours of laughter and confused reactions to computer related things.

Thank you to my committee, Dr. Frederick C. Harris, Jr., Dr. Sergiu M. Dascalu and Dr. Yantao Shen

Thank you to Dr. Lee Barford and Agilent for providing not only nearly limitless expertise in signal processing, but also access to advanced measurement equipment.

Thank you to John Sien, Devyani Tanna, Nathan Jordan, Mike Cox and Nolan Warner for the support.

Thank you to my family and close friends for dealing with me all this time.

Contents

Abstract **i**

Dedication **ii**

Acknowledgments **iii**

List of Tables **vi**

List of Figures **vii**

1 Introduction **1**

2 Background and Related Work **4**
2.1 Coprocessors 4
2.1.1 Floating Point Units 4
2.1.2 Graphic Processing Units 5
2.1.3 General Purpose Graphic Processing Units 8
2.2 Related Work 11

3 Sound Simulation and Visualization **12**
3.1 Background 12
3.1.1 Computational Acoustics 13
3.1.2 CUDA Streams 14
3.1.3 OpenGL 14
3.2 Overview 15
3.3 Implementation 17
3.3.1 Renderer/Main Thread 18
3.3.2 Simulation Manager 19
3.3.3 Memory Manager 20
3.3.4 Communication 21
3.4 Results 22
3.4.1 Future Work 22

4 Jitter Analysis **26**
4.1 Background 26

4.1.1 Jitter . . . 27
4.1.2 Technologies Used . . . 27
4.2 Overview . . . 29
4.3 Implementation . . . 30
4.4 Results . . . 32
4.4.1 Summary of Contributions and Future Work . . . 38

5 Conclusions **40**
5.1 Comparison of programming models . . . 40
5.2 Summary . . . 42

Bibliography **43**

List of Tables

4.1 Execution time in milliseconds for each implementation for each of the different input signal lengths. DTT = data transfer time. 36
4.2 Throughput for each implementation for each of the different input signal lengths. DTT = data transfer time, input size in samples . . . 36

List of Figures

2.1 An example of the difference between a Scalar addition of a vector and SIMD addition of a vector 6
2.2 A simplified illustration of the graphics pipeline[17] 7
2.3 A block diagram of the G80 architecture[25] 9
2.4 A block diagram of the Kepler architecture SMX[35] 10

3.1 Texture Mapped Volume Rendering: a technique which uses alpha blending to quickly render volumetric data.[11] 16
3.2 An overview of the structure of the application 17
3.3 An overview of the work done by each of the three threads in the application. 18
3.4 An example encoding using 4 bits, the 2 most significant bits represent whether a neighbor exists on the y axis and the 2 least significant bits represent whether a neighbor exists on the x axis. 19
3.5 The communication between the three components (shown in yellow) using the various communication queues (shown in blue). 21
3.6 The initial state before simulation initialization. 23
3.7 A rendering from the simulation while running. 24

4.1 An example of a typical eye diagram with inserted jitter[3] 28
4.2 The estimated time interval error before correction 32
4.3 An example of the final time interval error produced 33
4.4 Measurement apparatus used to verify the proposed method 34
4.5 The TIE probability density for a sample waveform 35
4.6 The TIE histogram for a sample waveform 35
4.7 The execution time of the purposed method in seconds. 37
4.8 The computational throughput of the purposed method measured in samples per second. 37

Chapter 1

Introduction

Throughout the history of computing a primary concern for practitioners has been the speed at which results can be computed. Due to this, an active area of research has been the acceleration of computing. The strongest driver for the increase in the performance of computers has historically been the approximate doubling of the number of transistors in any given integrated circuit every two years [33]. This doubling along with Dennard Scaling powered the steady increase in single core processor performance [10]. However, heat and power problems have forced chip manufacturers to develop processors with a larger number of slower cores to use the extra transistors each year [12]. While both Moore's law and Dennard Scaling have improved the computational power of chips, another path to improving the performance of programs has been the addition of specialized hardware to perform computationally expensive tasks. This specialized hardware often takes the form of a coprocessor.

An early example of coprocessors is the Data-Synchronizer Unit, also known as IO channels, first found in the IBM 709 and 7090 [5]. Overtime other coprocessors were developed for a variety of problems, most notably floating point coprocessors (FPUs). Due to the extremely slow speed of software implementations for floating point math, FPUs became a critical component for scientific and computer graphics programs. For computer graphics programs in particular, FPUs were unable to handle the large amount of floating point operations done for each frame of animation.This gave birth to a new breed of coprocessors called graphics processing units (GPUs). The manufacturers of GPUs used the increase of transistors year-over-year to focus

on parallel computing rather than increasing the performance of a single computing core as CPU manufacturers did[34]. This focus on parallel execution of floating point math over general computation has allowed GPUs to outpace CPUs in computational throughput. While CPU manufacturers allocated swathes of silicon to things like branch prediction, GPU manufactures concentrated on both things like higher core count and memory bandwidth [42]. These two features originally added to assist with graphics also give GPUs extremely high throughput for floating point operations. However this throughput comes at a cost, the highly parallel nature of the hardware requires a different approach to programming to deal with the trade-offs inherent in the hardware.

When designing software to work around these trade-offs, two approaches can be considered. The first approach is to program for the hardware itself. This approach requires the most knowledge about the underlying hardware and must be specialized to the specific GPU used[38]. The underlying hardware can vary greatly in capability. For instance, the presence of certain atomic operations or advanced thread management[37]. The other approach is to use a higher-level framework to abstract from the hardware [39]. The benefit to this approach is that the framework can represent the common idioms and patterns that exist in the programming language used. Both approaches offer a set of trade-offs which will be explored later in this thesis.

This thesis will present two applications which will explore the use of these two approaches for using the GPU as an accelerator. For the first approach, the application models the movement of sound through a room. This problem is an example of a class of problems in physics which map well to the programming model used on the GPU. The second approach is applied to a signal processing problem. The application shows how the use of a framework can speed the development of a solution while providing good levels of performance.

The structure of this document from here on out is as follows. Chapter 2 will examine how the history of the GPU and the underlying hardware of the GPU has shaped the nature of their use. Chapter 3 will discuss the use of the GPU as an

accelerator for simulating and visualizing the movement of sound through rooms using the CUDA programming model. Chapter 4 will explore how a framework called thrust built upon the CUDA can use an existing programming model to deliver better performance with minimal changes to the underlying code. Chapter 5 will finish with closing thoughts and ideas for future work.

Chapter 2

Background and Related Work

2.1 Coprocessors

For some problems, a generalized processor, like the commonly found x86 processor, is unable to deliver either adequate results or adequately fast results [8]. This can be due to a variety of concerns. Commonly for applications dealing with graphics or simulations, it is a problem of throughput. For other fields like computer security, coprocessors enable higher and more reliable levels of security [44, 53]. To remedy these concerns, specialized hardware called coprocessors are often used. These coprocessors allow the either the CPU or the programmer to off-load work from the CPU to a chip more suited to the problem domain. We are most interested in coprocessors which facilitate floating point mathematics.

2.1.1 Floating Point Units

Floating point arithmetic is at the heart of modern graphical and scientific applications. Nearly all modern implementations of floating point math follow the IEEE Standard[1]. Before the IEEE standard became widespread, there are many complications with using one of the various floating point arithmetic libraries. These complications were not limited to a lack of portability or strange rounding rules. Word lengths, exponent biases and precision varied greatly between implementations [22, 41]. Due to the complex nature of the IEEE standard, it is almost necessary to implement the standard in hardware for it to be usable in any application re-

quiring it. This meant that until CPU manufacturers had enough room on the chip to implement the standard, they shipped a separate Floating Point Unit (FPU) to help accelerate any applications using floating point math. Any modern CPU will now ship with FPU, and most ship with some form of a Streaming SIMD Extensions (SSE) instruction set.

Starting with the Pentium III, CPUs began to include extra registers for use with SSE instructions. This instruction set allows the CPU to execute many floating point operations in a single instruction. For example, an application is working with vectors where each vector has three values representing X, Y and Z. Normally to add two vectors together, each value in each of the vectors would need to be read from memory one by one, then added and then saved one by one. Using a SIMD instruction each vector is read at once, then add at once and then saved back to memory at once as shown in Figure 2.1. This can be done while the CPU is working on something else leading to massive speed increases when done properly. This concept of a Single Instruction Multiple Data (SIMD) coprocessor is the driving idea for the architecture of the Graphics Processing Unit.

2.1.2 Graphic Processing Units

Initially rendering tasks were handled in software using the CPU. Commonly, an FPU was used to help provide newer and better graphical techniques at an acceptable level of performance. However as the focus of the graphical techniques moved from 2D to 3D, specialized coprocessors were designed to keep up with increasingly high standards of performance. The design settled on a massively parallel SIMD architecture in which many pipelines are used to take geometry data, in the form of vertices, and transform it into color values to be displayed on the screen. Figure 2.2 shows a simplified version of this pipeline. This SIMD architecture enables many simple processors to work together to produce output simultaneously.

Initially the design of GPUs was that vertex processors would handle transforming vertices into the required perspective. From there vertices are rasterized into the

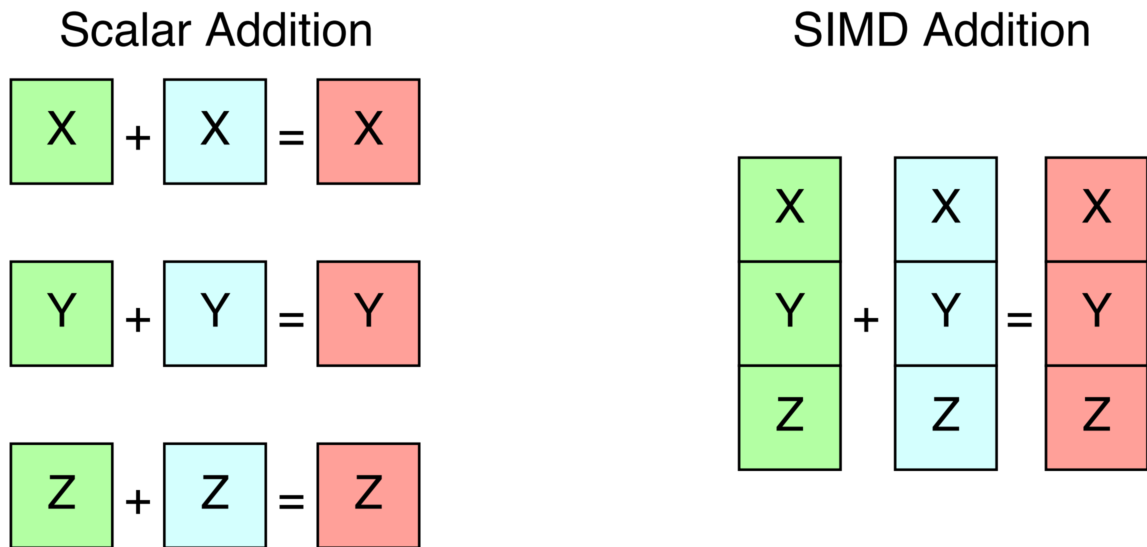

Figure 2.1: An example of the difference between a Scalar addition of a vector and SIMD addition of a vector

view port, and a fragment processor would calculate the color for each pixel of the screen. Initially this functionality was mostly fixed with only some small changes able to made, using an Application Programming Interface (API) like OpenGL [46] or Direct3D [14]. As the demands of these API became wider and wider in scope, the fixed functionality of the pipeline was opened up. Programs called shaders could be written for the vertex and fragment processors. Shader programs initially were limited but overtime have become more expressive and are the backbone of nearly all modern graphical techniques. Each API provides a language to write the shader programs and over time these languages [15, 23, 32]. As time progressed, addition stages were added to pipeline. Each new shader processor enabling application programmers more functionality. For example, the geometry shader enabled new geometry to be procedurally generated on the GPU critical for effects like Marching Cubes[9] to be efficiently done on the GPU. In addition to improving the programmablity of the GPU, other techniques like framebuffer objects[16] and transform feedback [7] allow application programmers to save results from the GPU back to the CPU for use in

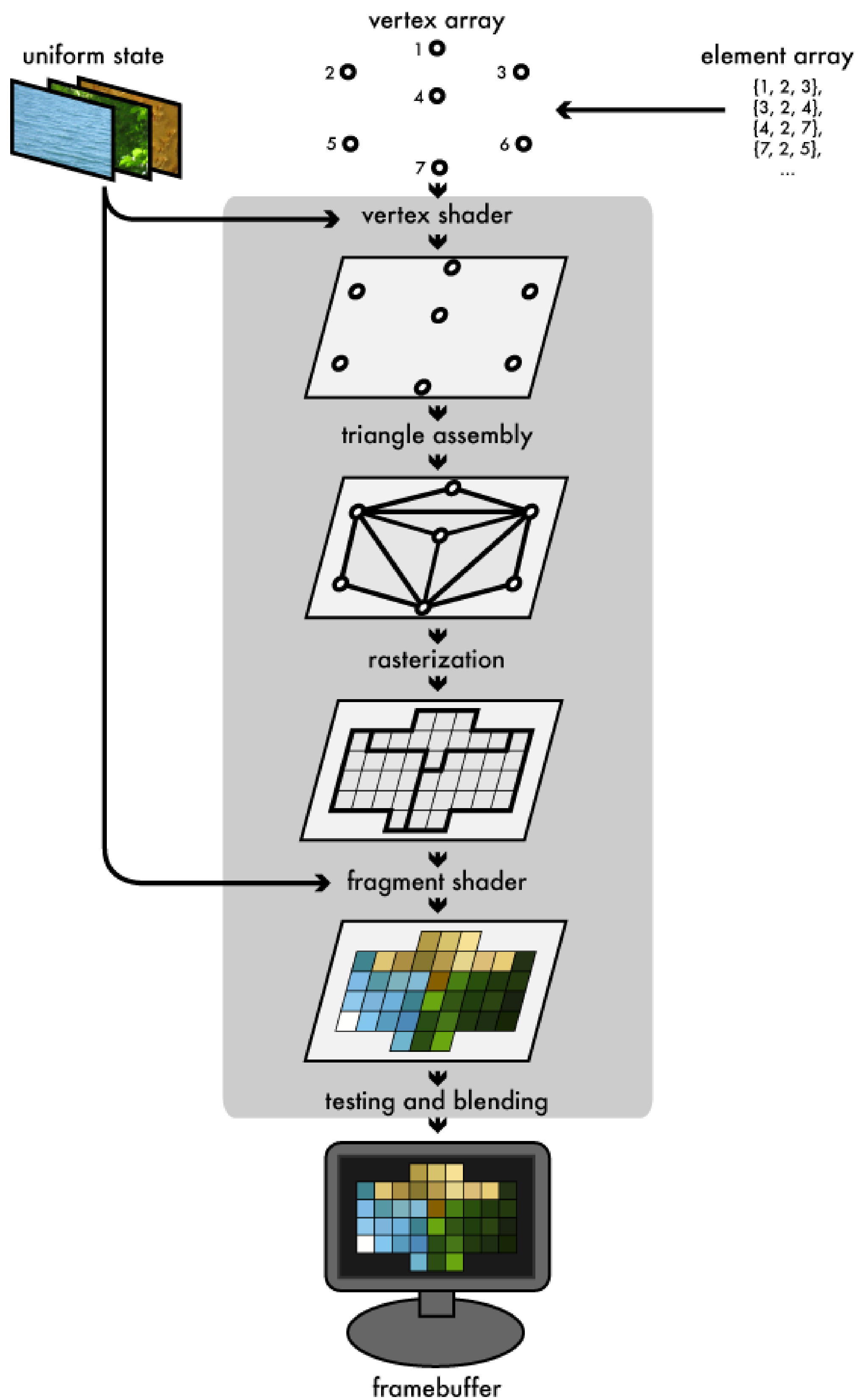

Figure 2.2: A simplified illustration of the graphics pipeline[17]

further rendering techniques.

2.1.3 General Purpose Graphic Processing Units

As GPU manufacturers packed more and more vertex and fragment processors into their GPU chips, the appeal of using the GPU for things other than graphics grew. By using the fragment shader in conjunction with framebuffer objects, it was possible to compute many things in parallel [19, 20, 30]. This practice called General Purpose GPU (GPGPU) programming allowed many scientific application programmers to accelerate all kinds of calculations. However it required not only knowledge of the problem domain, but also knowledge of the underlying graphic constructs and the API to control it. Despite this limitation, GPGPU enabled some applications to achieve performance gains[13].

CUDA

As GPGPU became more widespread GPU, manufacturers began to take note. Starting with the G80 series of graphics cards, NVidia unveiled a new underlying architecture called Compute Unified Device Architecture (CUDA) to ease the struggles of programmers attempting to harness the GPU's computing power [25]. While the new architect did not change the pipeline for graphics programmers, it did unify the processing architecture underlying the whole pipeline. Vertex and fragment processors were replaced with groups of Thread Processors (CUDA Cores) called Streaming Multiprocessors(SM). Initially with the G80 architecture, there were 128 cores grouped into 16 SMs shown in Figure 2.3. The most recent architecture has 2880 cores grouped into 15 SMs[35]. Figure 2.4 shows the evolution of the Streaming Multiprocessor architecture. In addition to the new chip architecture, CUDA also included a new programming model which allowed application programmers to harness the data parallelism present on the GPU. The primary abstractions of the programming model are kernels and threads. Each kernel is executed by many threads in parallel; CUDA threads are very lightweight and allow many thousands of threads to be executing on

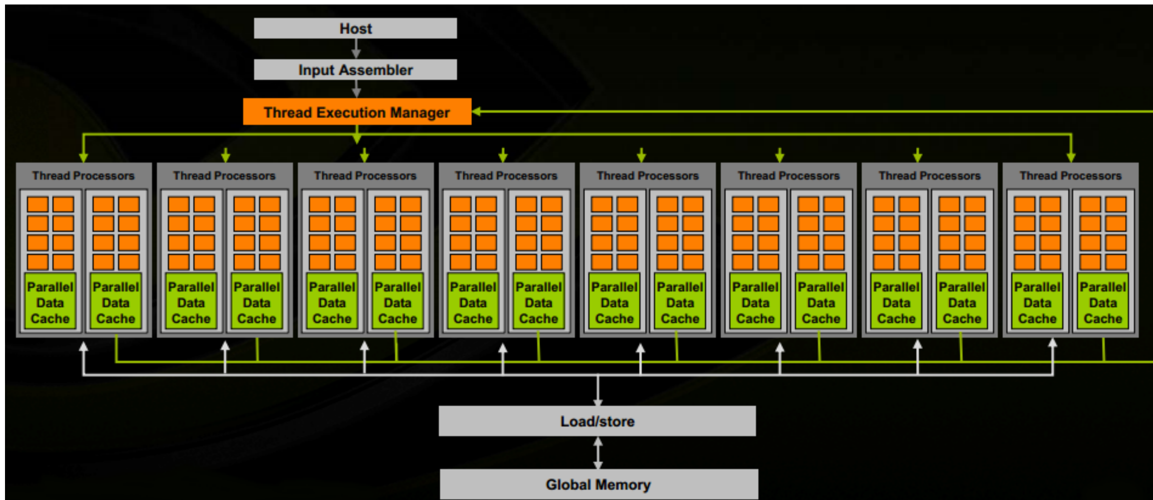

Figure 2.3: A block diagram of the G80 architecture[25]

a device at any given time.

The programming model is exposed to the application programmer by an extension of the C programming language commonly referred to as CUDA[37]. CUDA kernels are executed by threads, each is given an ID which is used to differentiate each thread's behavior. Unlike the previous GPGPU methods, threads are able to access and read/write to any memory location as needed. Threads can be additionally be grouped into blocks by the programmer giving multiple benefits: shared memory and shared cache within the block reducing memory overheads and thread synchronization enabling the programmer to more easily model some algorithms. Overall, CUDA gave programmers more freedom to design algorithms and data structures specifically for the hardware of the GPU.

However CUDA comes with some trade-offs. Due to the design of the underlying hardware, there are concerns the programmer must keep in mind. Care must be taken to ensure that groups of executing threads (warps) access contiguous chunks of memory as often as possible in order to make the most memory access coalescing provided by the hardware. Additionally since threads in a warp execute instructions in lock step heavy use of control flow primitives (if, for, while) can cause threads in warp to be inactive as sections of the warp take divergent paths [43]. Additionally

Figure 2.4: A block diagram of the Kepler architecture SMX[35]

due to its heritage from C, the low level nature of CUDA means that the programmer must be familiar with the hardware to get an acceptable level of performance.

Thrust

In many ways, the programming model presented by CUDA is a match to the programming model of C. For some programmers the programming model of C++ is more desirable; fortunately, the Thrust framework exists. Following the motivations of the Standard Template Library (STL) of an efficient, comprehensive and extensible approach to designing both algorithms and data structures [49], Thrust provides a set of common STL operations and data structures which run on the GPU. This allows an application programmer using C++ to accelerate common operations like sorting, transforming and reducing[39]. Thrust provides these features while maintaining interoperablity with CUDA C; allowing programmers easier access to hard to implement parallel primitives that have been highly optimized like a radix sort which can provide a speed up (ranging from 2-5 times faster than a comparable sorting routine on an 8 core CPU) [45]. These features make Thrust an appealing tool to provide easier access to GPU acceleration for many applications.

2.2 Related Work

At the time of the writing of this thesis the use of GPUs as a massively parallel math coprocessor is increasing. This is due not only to their increase in power over the years but also to their increasing ubiquity. Because of this their use is becoming more common in fields which require a large amount of throughput for any application. For instance, CUDA has been used to accelerate N-body simulations, providing extremely large gains against tuned serial implementations and modest gains over tuned implementations for the CPU[6, 40].

Chapter 3

Sound Simulation and Visualization

The goal of this chapter is to explore the techniques used when accelerating an application of the GPU with CUDA. The goal of this application was to serve as a proof of concept for a larger application. For this prototype, a visualization of the of the sound waves in the room was desired so that phenomena like standing waves could be quickly identified. In order to produce the visualization quickly, some form of acceleration would be required. The GPU was chosen as the accelerator using CUDA to produce the data for the simulation.

3.1 Background

As discussed in Chapter 2, a factor in the rise of the use in GPGPU has been the adoption of GPGPU as an accelerator for many different simulations. Of particular interest to this application are simulations which model the movement of waves through a medium. CUDA has been shown to greatly accelerate the computation of solutions for Finite-Difference Time-Domain (FDTD) methods [28, 54]. FDTD methods work by breaking the computational domain, commonly a physical region being modeled, into a grid. Then the solutions for the chosen equations are calculated for each point in the grid for each time step. The benefits of the method are: values are calculated for each point in the grid which makes visualizations easier to create, a large range of frequencies can be simulated with proper input allowing phenom-

ena to present themselves and the data parallelism inherent in the methods aids in accelerating the computation of results.

3.1.1 Computational Acoustics

Computational Acoustics is largely based on the physical modeling of the propagation of sound waves. This propagation is governed by the linearized Euler equations[50]. There have been many methods developed to model the propagation of sound waves; these methods fall into one of two categories: geometric and numeric.

The geometric methods tend to be based on a technique called ray tracing, more commonly used to create computer graphics. These ray tracing methods are able to quickly model the acoustic properties of a room by assuming that high frequency sound waves behave similarly to a ray of light. However, these methods have a critical flaw. Because the wavelengths of audible sounds are on the order of commonly found objects, sound waves exhibit diffraction (bending) where light would not. Because of this recently these geometric methods have fallen out of favor as more accurate models have become accessible.

The numeric methods attempt to directly solve the Wave Equation. The benefit of this approach is that phenomena such as diffraction are accounted for. However, numerically approximating the wave equation can become expensive, especially for FDTD methods. This is due to the requirements for memory and computation scaling up as the sampling rate desired increases. With the the time step T defined as $T = 1/f_s$ and the grid spacing g defined as $g = \frac{344*\sqrt{3}}{f_s}$ it can be seen that as the sampling rate increases not only does the time step of the simulation get smaller, but the size of the grid also increase. These factors combined have meant that until recently FDTD solutions for the acoustic wave equations have been prohibitively expensive.

Recently solutions using CUDA have shown promising results in accelerating 3D FDTD solutions to the wave equation[47, 51].

3.1.2 CUDA Streams

As discussed in Chapter 2, CUDA expose the computational power of the GPU through a C programming model. It additionally provides an API for scheduling multiple streams of execution on the GPU. This allows the hardware scheduler present on CUDA enabled GPUs to more efficiently use all of the compute resources available. When using streams, the scheduler is able to concurrently execute independent tasks. In order to fully use streams for all including memory transfers, the CUDA driver must allocate all memory on the host that will be used for CUDA API calls. This ensures that the memory is pinned (page-locked) so that Direct Memory Access (DMA) can be enabled.

However, using streams is the only way to get the full performance of the GPU using CUDA, and it comes with some concerns. The first concern is that pinned memory can not be paged out and can therefore impact the amount of paging for any other memory that is virtually allocated. This means that if too much pinned memory is allocated, other components of an application may see a performance loss. In addition care must be taken to order CUDA memory transfers and kernel launches in such a way that the scheduler is able to properly schedule each of the actions concurrently[36]. Additionally some advanced memory features like shared memory or texture memory can become restricted when using streams.

3.1.3 OpenGL

OpenGL is an API for rendering 2D and 3D computer graphics. It was originally developed by Silicon Graphics Inc. It provides a portable API for creating graphics which is independent of the underlying hardware. It is widely used for a variety of applications ranging from Computer Aided Design (CAD) to scientific visualizations.

At its core, the OpenGL API controls a state machine. This state machine maintains the hardware and is in charge of ensuring the proper resources are loaded at the proper time. It is important to note that for performance reasons this state machine was not designed to be used with multiple threads. The OpenGL driver can

be made to run in a multithreaded way, but the driver does nothing to protect the programmer from race conditions. The API has calls for uploading geometry data, texture data and more to the GPU. In addition, it also exposes a compiler for the GLSL shading language.

As discussed in Chapter 2 the GLSL shading language (shaders) gives the application programmer more control over the functions of OpenGL. The programmer can use these shaders to accomplish advanced rendering techniques such as lighting effects, producing rolling ocean waves or programmatically generating textures or geometry.

One technique of interest to the application discussed in this chapter is the rendering of volumetric data. There are many techniques for accomplishing this task[21]. A popular method for volume rendering uses the texture mapping hardware of the GPU and alpha blending to render volume metric data. The technique involves rendering many semi-transparent slices of the volumetric data as shown in Figure 3.1.

3.2 Overview

The application presented in this chapter uses a 3D FDTD method for simulating sound in a room. A couple of assumptions are made. A simplified model of the sound absorbence is used in lieu of a more computationally expensive one and that all sources of sound are omnidirectional.

The application is structured as a chain of producers and consumers. Once the simulation has begun, the simulation manager begins to produce simulation data. This raw form of the data is both unsuitable for visualization and is also located in a section of GPU memory which the OpenGL driver is unable to access directly. The simulation manager therefore publishes this raw data to a queue for processing. The memory manager takes this raw data, copies it to the CPU and puts it into a form suitable for visualization using OpenGL. These processed frames are published to a queue until they can be uploaded to the OpenGL driver. The renderer is responsible for all the calls to the OpenGL API. After uploading the current frame as a texture,

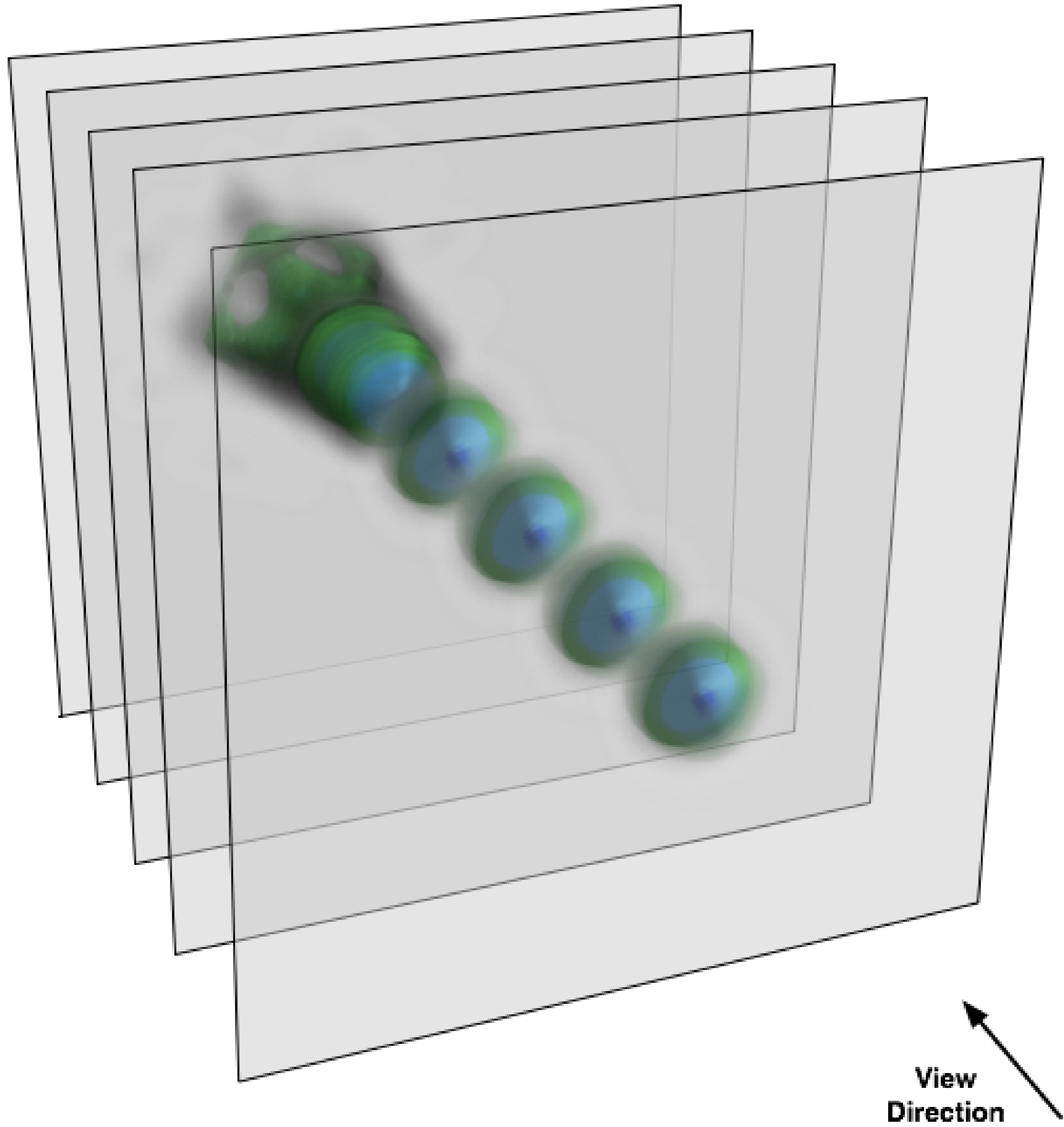

Figure 3.1: Texture Mapped Volume Rendering: a technique which uses alpha blending to quickly render volumetric data.[11]

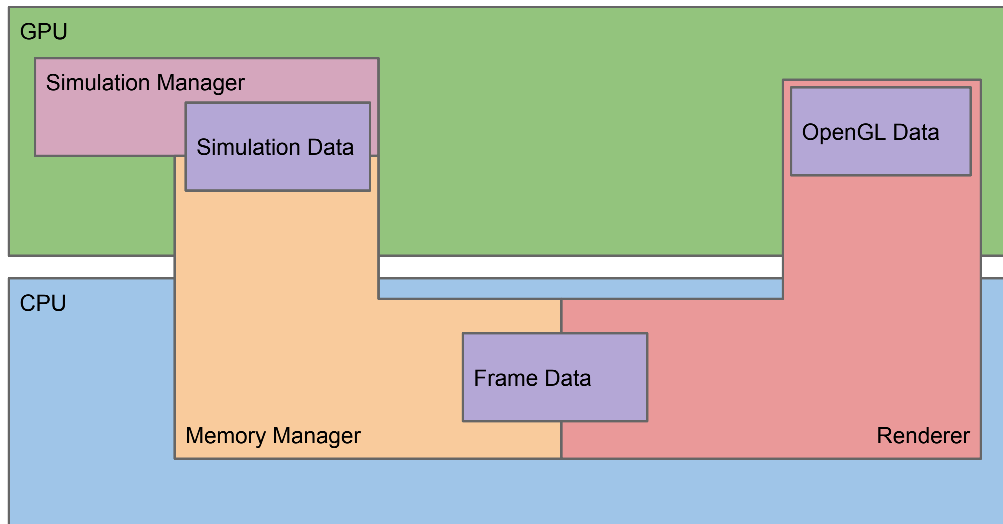

Figure 3.2: An overview of the structure of the application

the frame is drawn using texture mapped volume rendering. An overview of this structure can be seen in Figure 3.2

3.3 Implementation

The application is structure into three major components: the renderer, the simulation manager and the memory manager. Each of these components is run on its own thread to allow as many concurrent actions to occur as possible. An overview of each component's execution is shown in Figure 3.3. Care was taken to design each component in such a way that a component could be redesigned and replaced easily. For instance if the data set would not fit on a single GPU then the simulation manager could be rewritten to accommodate this and the rest of the application could remain unchanged.

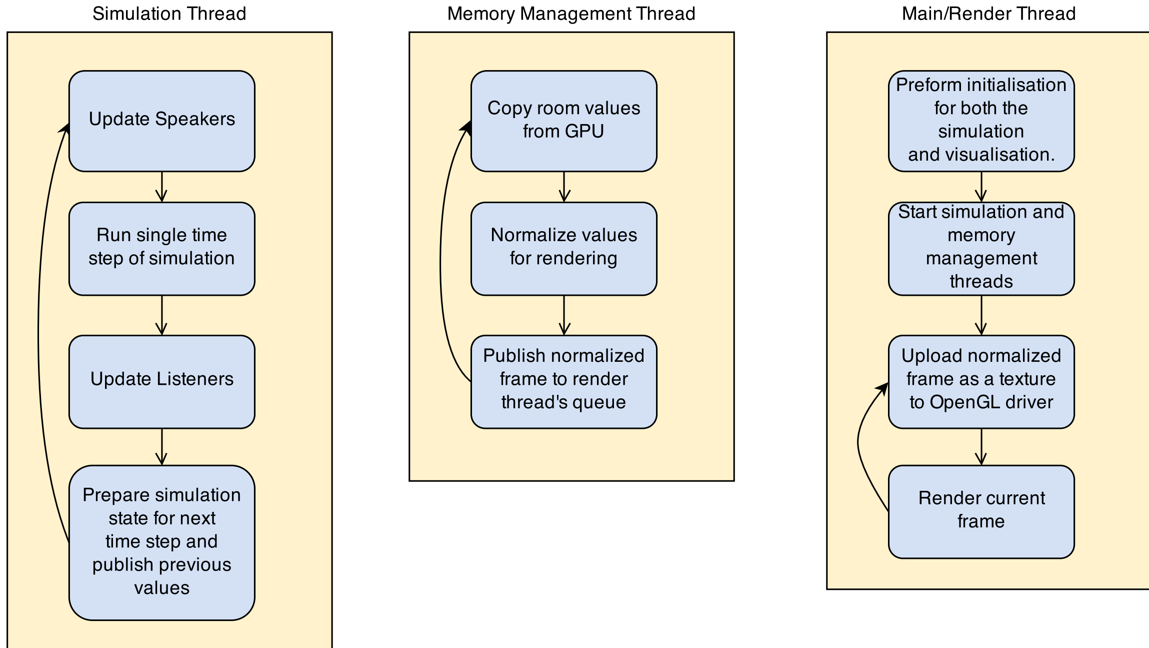

Figure 3.3: An overview of the work done by each of the three threads in the application.

3.3.1 Renderer/Main Thread

The first action taken by the main thread is to initialize the state of the OpenGL driver, create a window and initialize a OpenGL context within that window. These actions are all required to begin making any other calls using the OpenGL API. Assuming that there are no errors, the next thing done is the loading of the model information and test sound sample. For this prototype, a test room was made. For the simulation it is important that the model contains both geometry and absorbence information for each point in the grid. The geometry data consists of a 3D array of values encoding whether or that point in the grid is connected its neighbors. An example of a possible encoding for the 2D case is shown in Figure 3.4. Once both the room and sound sample are loaded from disk, the data is used to construct the initial state of the simulation. After the simulation is initialized, both the simulation manager and memory manager are started and the rendering tasks begin.

The first of the rendering tasks is to initialize the state of OpenGL, this includes loading the shaders required to perform texture mapped volume rendering as well as

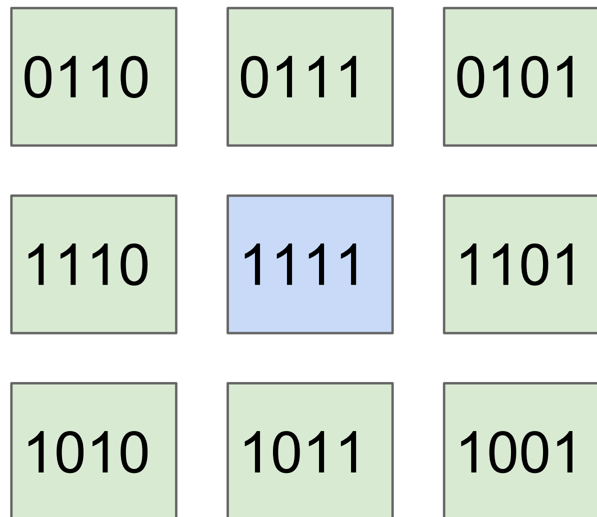

Figure 3.4: An example encoding using 4 bits, the 2 most significant bits represent whether a neighbor exists on the y axis and the 2 least significant bits represent whether a neighbor exists on the x axis.

preparing OpenGL geometry data required for the technique. Once this initialization has been done, the render thread falls into the render loop.

The render loop primarily consists of two actions: uploading frames to OpenGL memory and performing the volume rendering. Uploading frames to OpenGL memory is the first task done. The renderer checks to see if any processed frames have arrived in its queue, shown in Figure 3.5. If a processed frame has been published then it is uploaded to OpenGL memory and the now blank frame is published back to the memory manager. The renderer then volume renders the current frame that is in OpenGL. This loop continues until a user inputs the exit command or the simulation reaches its completion.

3.3.2 Simulation Manager

The simulation manager's first action is to allocate all the memory required for the simulation (room information, blank simulation state arrays and input/output ar-

rays). It then transfers the encoded grid representing the room, the absorbence values for each point in the grid and the input for the simulation. Once all of the CUDA memory has been allocated and all of the initial simulation state has been transferred onto the GPU, the simulation manager waits to be told to start.

Once the signal to start is received, the simulation acquires three simulation blanks, representing the t_{n-1}, t_n and t_{n+1} simulation states, from the simulation blanks queue. With that last bit of preparation done, the simulation manager drops into the main simulation loop.

The first step of the simulation is to update all of the input into the simulation. Each input source has a location associated with it that is updated. After that the simulation kernel is run. The kernel uses an equation in the form of Equation 3.1, where p_n is the acoustic pressure for t_n, to numerically approximate the propagation of the wave.

$$p_{n+1}(x,y,z) = (1/3[p_n(x+1,y,z) + p_n(x-1,y,z) + p_n(x,y+1,z) + p_n(x,y-1,z) + p_n(x,y,z+1) + p_n(x,y,z-1)]) - p_{n-1}(x,y,z) \quad (3.1)$$

The form of the equation depends on the value of the encoded geomotry for that point on the grid. Care is taken to ensure that memory accesses are sequential for the warps assigned to the the kernel. The kernel is run in its own stream, which allows the hardware scheduler to schedule both the kernel and any copies that the memory manager is scheduling. If the kernel was not run in its own stream, then the hardware scheduler would not have enough information to schedule the two tasks concurrently. Once the kernel has finished, any listeners present in the model room are updated. The simulation manager then decides whether it's time to publish a raw frame to the unprocessed frame queue as shown in Figure 3.5.

3.3.3 Memory Manager

The memory manager is the simplest of the three components of the application. As seen in Figure 3.2 the memory manager acts a a bridge between the simulation

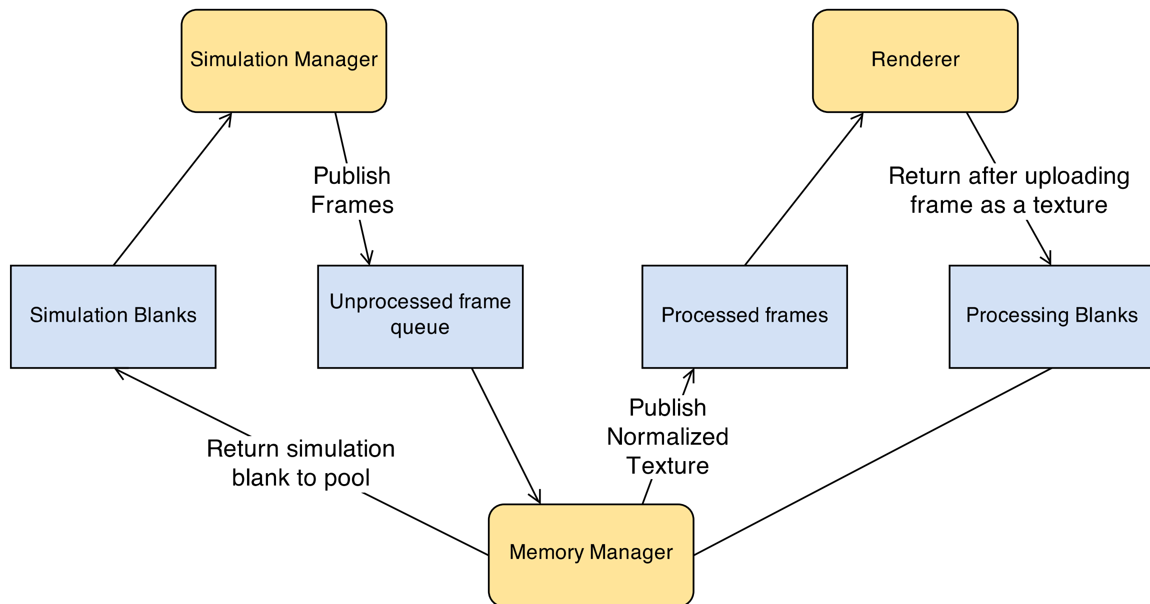

Figure 3.5: The communication between the three components (shown in yellow) using the various communication queues (shown in blue).

and the visualization. The memory manage takes the raw frames published by the simulation and normalizes them before publishing them to the renderer. The only major concern for the memory manager is that memory transfers off of the GPU must be run in a separate stream of execution than the simulation kernel. If this is not done, then any memory transfers will block the execution of the simulation kernel.

3.3.4 Communication

The application uses three threads to run the three components concurrently. These threads communicate using thread safe queues. The reason that queues were chosen as the data structure for the message passing between the thread is that they both naturally preserve ordering and allow the simulation and visualization to not outpace

each other. The application is essentially a producer-consumer problem where both the producer and consumer are fighting for the same resource. If there was not some way to limit which component controls most of the GPU time, then the contention for the GPU would cause problems in either the simulation or the visualization. Additionally, the use of queue as the inter-thread communication medium makes any future attempts to use multiple machines or devices easier by allowing the queue to hide the origin of any information.

3.4 Results

The prototype was tested on a computer with an Intel Core2 Quad core CPU Q9450 and a NVidia GeForce GTX 480 GPU which contains 15 groupings of 32 compute cores for a total of 480 compute cores and has 1.5 GB of on board memory. Two different test signals were used, one a music sample and the other a 8kHz sin wave. The test room modeled was 12m x 12m x 3m with a pillar in the center of the room. When using the sin wave the simulation modeled standing waves where standing waves would be expected to form. Figure 3.6 and Figure 3.7 show the visualization during a test run using the music sample.

3.4.1 Future Work

This chapter presented an application that benefited greatly from the use of GPU acceleration. The problem presented exhibited data parallelism which assisted in the implementation of the kernel. This acceleration allowed the simulation to run at a quick enough pace to facilitate the creation of a visualization alongside the simulation.

Despite this, there are issues that should be addressed in future work. Currently, the simulation is limited in size and frequency range due to memory concerns. This could be remedied by replacing the current single GPU simulator with a simulator that uses multiple GPUs if present on the computer or even a clustered simulator. Additionally, the visualization requires that the user has some understanding of the mechanics of wave propagation to be useful. If a user wanted to use the simulator

Figure 3.6: The initial state before simulation initialization.

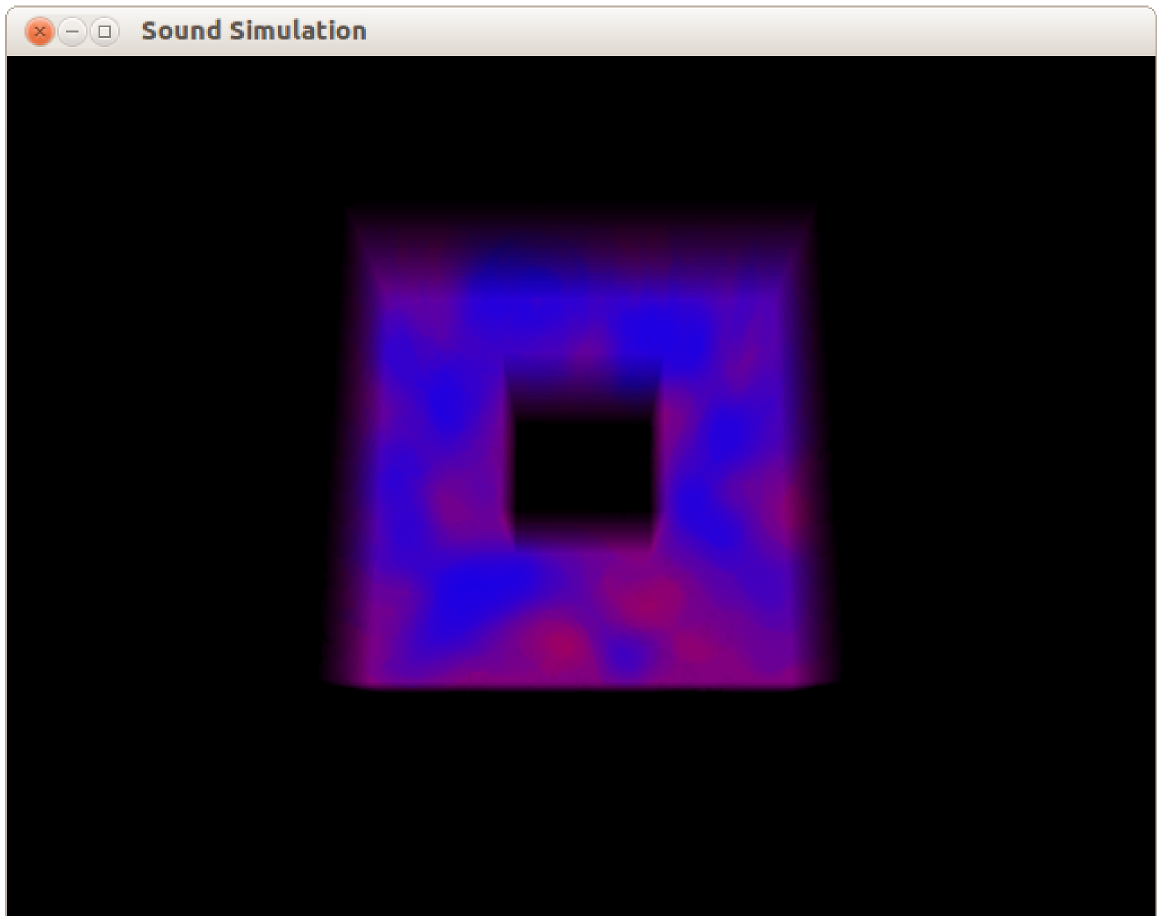

Figure 3.7: A rendering from the simulation while running.

and visualizer to place speakers in a room, it might be more helpful for the visualizer to analyze the frames coming from the simulation to programmatically find good speaker placements.

Chapter 4

Jitter Analysis

The basis of this chapter was previously published in the 2013 IEEE International Instrumentation and Measurement Technology Conference in a paper entitled Massively Parallel Jitter Measurement from Deep Memory Digital Waveforms[29].

The goal of this chapter is to explore the benefits of using the Thrust framework provided by Nvidia. The application presented compared the performance of an implementation of the purposed method for both CPU and GPU. The CPU implementation used the Standard Template Library(STL), this allowed the GPU implementation to be very similar to the CPU implementation due to the parity of the STL and Thrust.

This chapter first will delve into the background required for Digital Signal Processing (DSP) and will more deeply explore Thrust. From there, a brief overview of the purposed method will be discussed. After that the implementation of the method and then the results will be discussed.

4.1 Background

As previously discussed, GPUs are highly capable at accelerating applications which deal with problems that inherently data parallel. This is due to the extremely specialized nature of the hardware. Another example of a problem domain that is largely

data parallel is Digital Signal Processing. Due to this, DSP has been an attractive target for acceleration on the GPU [4, 24]. The specific DSP problem explored in this chapter is the measurement of jitter present in a given digital signal.

4.1.1 Jitter

The jitter present in a signal can be major factor considered when designing a serial digital communications channel. As the bit error rate (BER) drops, the allowable amount of jitter also drops. Because of the importance of some of these channels, there is a need to accurately measure jitter. This relationship between the BER and jitter means that it becomes more valuable to gather a longer section of the waveform, a "Deep Waveform". This larger sample of the waveform increases the probability that some rare event affecting the signal is detected, as well as increasing the statistical significance of the tails of the measured jitter histogram or fitted probability distribution used to extrapolate jitter performance and BER[31]. Jitter histograms and eye diagrams like the one seen in Figure 4.1 are used to not only predict the BER, but can also provide insight into the source of the jitter in the signal. The desire to obtain a deep waveform of a given signal is met by modern oscilloscopes. However the computing a jitter histogram, preparing graphical representations of signal properties and clock recovery become computationally expensive as the size of the waveform increases. This makes both the clock recovery and the measure of jitter for a signal an attractive target for acceleration.

4.1.2 Technologies Used

Standard Template Library

The Standard Template Library (STL) is a C++ library since its release has heavily influenced the design of the C++ Standard Library. Any reasonably current compiler will ship with a version of the STL. The STL is broken into 4 major categories: containers, algorithms, iterators and function objects (functors)[18]. Each part of the STL is designed to generically work with not only the rest of the STL but also the

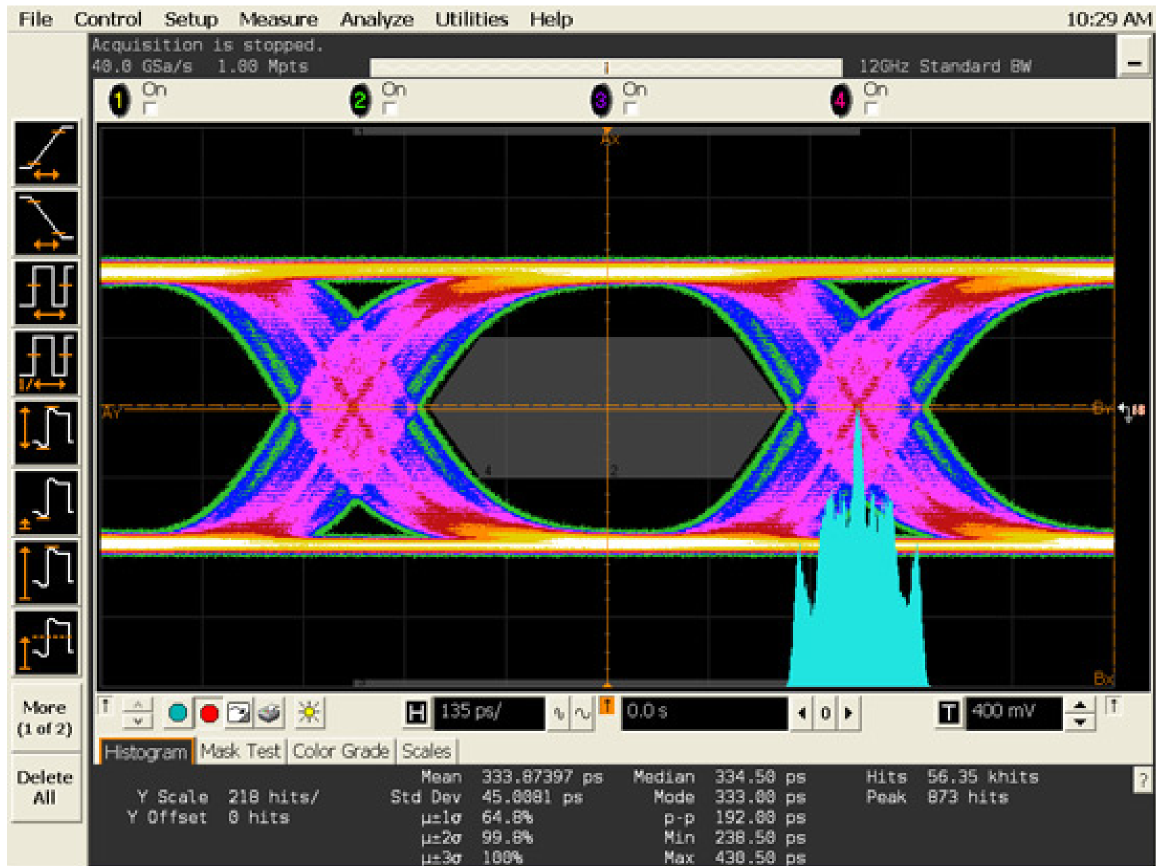

Figure 4.1: An example of a typical eye diagram with inserted jitter[3]

primitive types and constructs provided by C++. Due to this flexibility, the use of the STL is widespread and many compilers have specially optimized the code generation to maximize the effectiveness of the STL.

The most notable parts of the STL are the algorithms and containers. The STL provides many common algorithms, such as a fast sort, reverse and unique. In addition, it provides generic containers using templates for these algorithms to work on.

Thrust

As previous stated in Chapter 2, Thrust is a framework built upon CUDA modeled heavily after the STL[39]. Thrust provides a much higher level of expressiveness than CUDA by providing more abstractions from the underlying hardware. One of the most immediately notable things abstracted away from the application programmer's control is memory management. Thrust provides two templated containers: host_vector and device_vector. These two containers are both arrays which automatically grow when their capacity runs out. Data can be easily copied from a host_vector to a device_vector using the C++ assignment operator =. Additionally, the vectors can be used interchangeably with the rest of the the libraries provided by Thrust and the freedom from manual memory manage allows for a more rapid pace of development when combined with the provided algorithms.

4.2 Overview

The goal of this application was to present a method for fixed frequency clock recovery and jitter measurement of a two-state digital signal using massive parallel processors and large data sets. To accomplish, this a CPU implementation was first done using C++ to establish that the method would produce sufficiently accurate results. Once the correctness of the algorithms used was established, a version which ran on the GPU was written using Thrust. The next section discusses in depth the method used.

4.3 Implementation

The purposed method makes some assumptions about the input signal. Other than the assumption that the signal is digital, the most important of which is that transitions will occur in the signal. Due to this, transmission protocols like 8b/10b[52] should be used so that transitions can be guaranteed to happen.

The purposed method is a series of transformations and reductions of the input signal. The first and most important transformation preformed finding each of the transitions in the input signal with sub-sample accuracy. This initial transformation forms the basis of the method. This is accomplished by first determining relative values for the high, low and middle voltages of the input signal. Using these values, the transition points in the signal can be found. Once the transition points are found, a rough estimate of the number of samples per unit interval is made. Once the rough estimate of the duration of a unit interval is made it is further refined. This refinement is done by comparing the number of samples to the estimated total number of unit intervals in the input signal. With this refined estimate, a phase offset estimate can be calculated for each of the transition points in the input signal. Using linear regression, the phase offset and a correction for our final estimate can be calculated. From this further analysis tools like jitter histograms or eye diagrams can be constructed.

While the proposed method is straight forward in its design, there are considerations that were made to translate the algorithm to work efficiently on the GPU. The most major change is that while the STL provides a primitive for selecting the nth element of a container, Thrust provides no matching call. For the CPU implementation the nth_element was used to obtain percentile data without the overhead of constructing sorted data or histograms. However since this functionality is missing from Thrust, it was necessary to sort the data each time a percentile was needed. This is not an issue since Thrust provides a very fast sort [27]. However, the sort destroys its input data, so it was important to maintain proper copies of data which was to be sorted. Another concern when working on the GPU is minimizing the total

time spent transferring data over the PCI bus to the card. Care was also taken to reduce all data transfers to the bare minimum, especially in the case of the large data sets used.

A more in-depth explanation of the process follows.

Step One: Determine relative high, low and middle voltage levels in the signal. The first step is determining what the values for high and low will be based on the input signal. This done by finding the 1st percentile of the voltages in order to find the value for low and finding the 99th percentile for the high value. From these values, a middle value can be calculated by finding the midpoint of the two values.

Step Two: Find transition points in the signal with in sub-sample accuracy. The second step is finding all the points of voltage transition in the signal. When a transition is found, its position is noted. For each of these transition points, a linear interpolation step is done to find the sub-sample transition time. These values are then summed to find each transition point in the signal with sub-sample accuracy.

Step Three: Make a rough estimate of the number of samples in the signal per unit interval. From these transition times, a histogram of the inter-transition intervals is made. To make this histogram ,the inter-transition values are found by finding the adjacent difference of the sub-sample transition times. Using these inter-transition values, a histogram is made. The rough estimate of the samples per unit interval is obtained by finding the 25th percentile of these inter-transition times. Using the 25th percentile of these inter-transition times is only valid due to the nature of the signal.

Step Four: Refine the rough estimate of samples per unit interval. The rough estimate from the last step will not be accurate enough for further use. Due to this, the rough estimate of samples per unit interval is refined. To do this the first thing needed is the number of estimated unit intervals in the signal. Following that, the total number of samples which occur from the first transition to the last transition in the signal is calculated. That number of samples is then divided by the

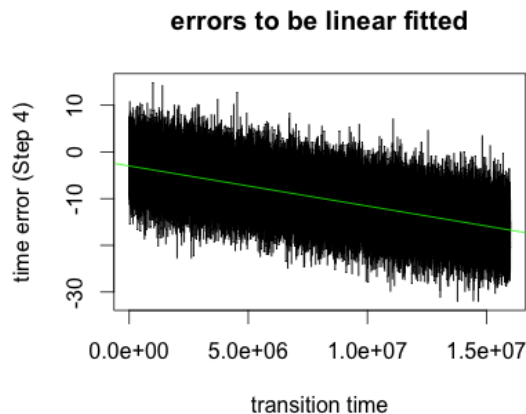

Figure 4.2: The estimated time interval error before correction

estimated number of unit intervals in the signal.

Step Five: Eliminate remaining linear trend in time interval error. In this step, the estimate will be corrected a final time using linear regression. Figure 4.2 shows an example of the linear trend being corrected. In addition to giving the final estimate of the samples per clock, this linear regression step also gives the phase correction needed for clock recovery. To correct the refined estimate, a linear regression of estimated time interval error, given the period estimate obtained in step 4 and the sub-sample transition times from step two is performed. From the linear least squares fitting, a coefficient and an offset are obtained. The coefficient is used to correct our refined period estimate from the last step, and the offset is the phase offset. Figure 4.3 shows the result of this step.

Step Six: Construct appropriate plots Once the phase offset is calculated and the estimate of the unit interval has been corrected, the appropriate plots are constructed.

4.4 Results

The proposed algorithm was implemented two times. One implementation used only the CPU and was written in C++ using the primitives provided by the STL to verify

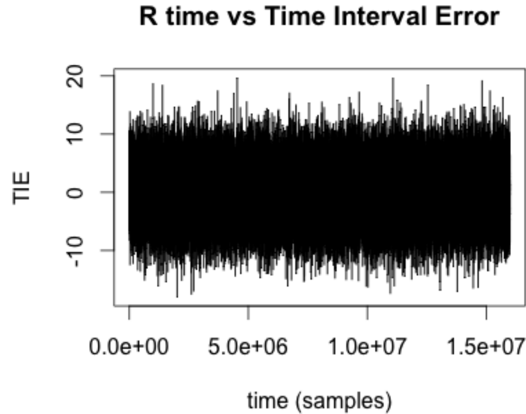

Figure 4.3: An example of the final time interval error produced

the results of the purposed method. The other implementation written in C++ using Thrust to run code on an NVIDIA GPU was used to measure the total speed up of the method with increased core counts. Both implementations were tested with an actual measured waveform obtained using the apparatus shown in Figure 4.4, in which waveforms were produced by an Agilent 8133A Pulse Generator and captured by a Agilent DSA91304A Digital Signal Analyzer.

The experimental platform is an Intel Core2 Quad core CPU Q9450 at 2.66GHz with 8 GB of memory. The GPU used was an NVidia GeForce GTX 480 which contains 15 groupings of 32 compute cores for a total of 480 compute cores and has 1.5 GB of on board memory. The CPU implementation was compiled with g++ version 4.6.3, but due to the restrictions of Thrust the GPU implementation was compiled with the CUDA toolkit version 5.0 and g++ version 4.4.7. All implementations were compiled on 64 bit Ubuntu 12.04 with optimization level -O3.

The measurement results obtained by the serial and GPU versions were identical. For example, when all 16 million samples are used, both versions yield the standard deviation of TIE of 4.630199 samples. The resulting plots of the time interval errors (TIEs) for each transition and the jitter (probability density of the TIE) are shown in Figures 4.3, 4.5 and 4.6. (The probability density is a kernel density estimate [48] computed from the TIEs.)

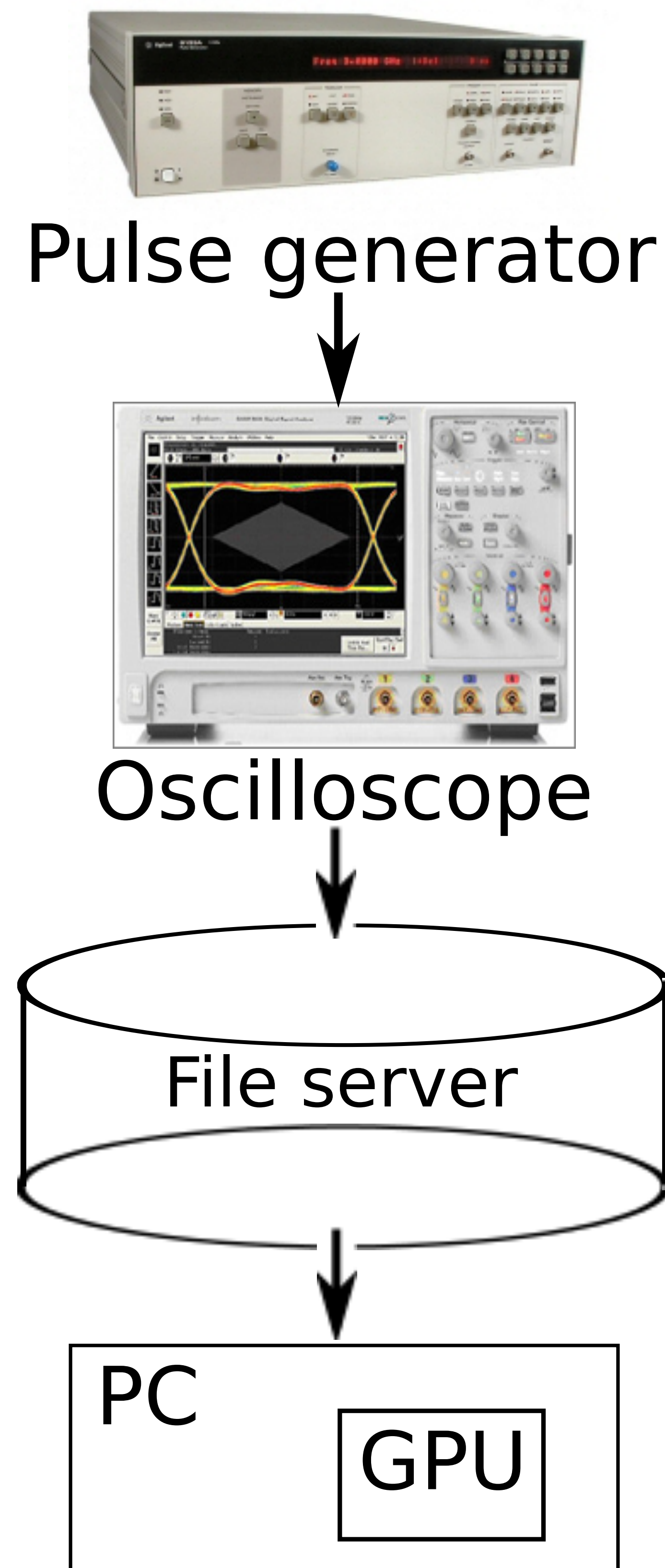

Figure 4.4: Measurement apparatus used to verify the proposed method

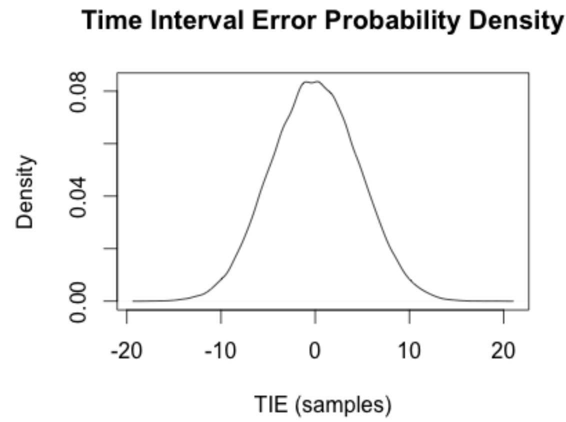

Figure 4.5: The TIE probability density for a sample waveform

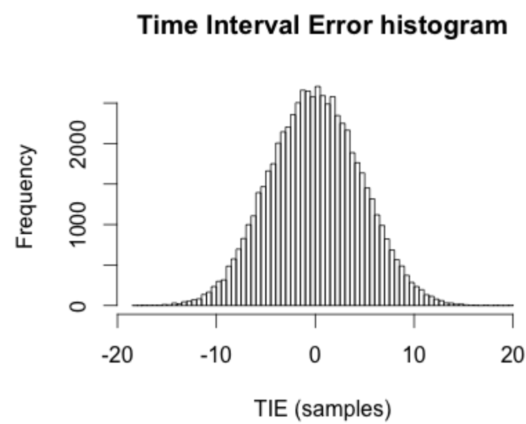

Figure 4.6: The TIE histogram for a sample waveform

Table 4.1: Execution time in milliseconds for each implementation for each of the different input signal lengths. DTT = data transfer time.

Input Size (samples)	CPU	GPU (no DTT)	GPU (with DTT)
1,000,000	15.19	7.17	9.89
1,414,213	22.38	7.43	10.96
2,000,000	28.59	8.61	13.21
2,828,427	43.44	10.11	16.36
4,000,000	62.24	12.33	20.88
5,656,854	89.56	16.26	28.15
8,000,000	144.44	20.84	37.38
11,313,708	242.54	28.37	51.66
16,000,000	309.37	38.23	70.88

Table 4.2: Throughput for each implementation for each of the different input signal lengths. DTT = data transfer time, input size in samples

Input Size	CPU (MSa/s)	GPU (MSa/s, no DTT)	GPU (MSa/s, with DTT)
1,000,000	65.835	139.460	101.122
1,414,213	63.180	190.330	129.046
2,000,000	69.946	232.297	151.401
2,828,427	65.108	279.701	172.917
4,000,000	64.269	324.492	191.561
5,656,854	63.163	347.852	200.963
8,000,000	55.385	383.880	214.013
11,313,708	46.646	398.743	219.002
16,000,000	51.717	418.563	225.740

There were three timings measured: one timing for the CPU, one timing for the GPU which did not include the data transfer times (DTT) and a one timing for the GPU which included the transfer times as well. Table 4.1 and figure 4.4 show the results for these timings as the size of the input signal was increased, while Table 4.2 and figure 4.4 shows the throughput of these results. For signals less than 16,000,000, samples in length sections of the input waveform were used. The waveforms used in these timing experiments were pseudo-random binary signals (PRBSs).

Table 4.1 and Figure 4.4 show that when considering only the computations, the difference between the serial CPU implementation and the parallel implementation on the GPU as the input size grows is sizable. While the CPU implementation was

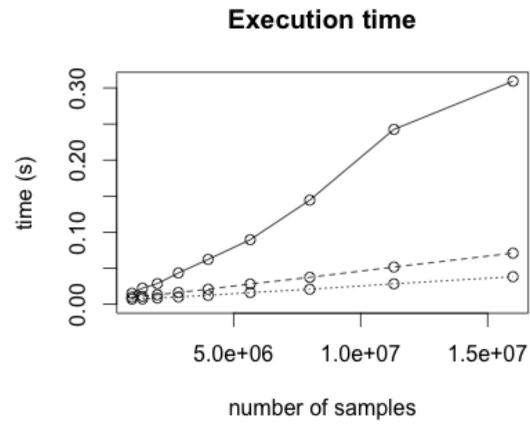

Figure 4.7: The execution time of the purposed method in seconds.

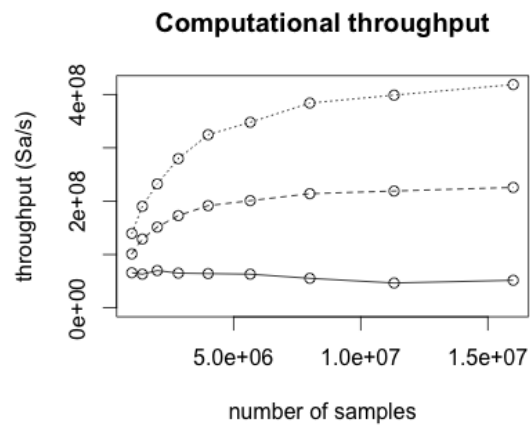

Figure 4.8: The computational throughput of the purposed method measured in samples per second.

not expected to keep pace with the GPU implementation, it is interesting to note the significant speed up that is present. Another interesting thing to note is that for even with signals as small as 1,000,000 elements, when the data transfer time is considered the GPU implementation is able to handily beat the CPU. This result means that it is quite practical to use this method over a pure CPU implementation. Table 4.2 and Figure 4.4 show the massive difference in throughput that the GPU is able to provide over the CPU. For the entire 16 million sample signal, the GPU implementation was able to handle 8 times as many samples as the CPU in the same amount of time when not considering the total transfer time of data on and off of the GPU. This is a reasonable expectation to make as the method presented is likely to be an intermediate step in a larger process.

An interesting thing to note is that despite the relatively high throughput shown by the GPU in Table 4.2 and Figure 4.4, the application is still computationally bound. While the method as implemented is capable of consuming 418 million samples a second the PCI express bus is capable of transmitting more samples[26].

4.4.1 Summary of Contributions and Future Work

There are two novel contributions of the method purposed in this chapter: (1) the application of massively parallel processors to do clock recovery and to quantify jitter and (2) the use of sorting to replace building histograms when analyzing waveforms in order to more efficiently use such processors. By using the efficient sorting, transformation and reduction routines provided by Thrust, the benefits of constructing a solution using the primitives offered by Thrust were shown. Identical measurement results were obtained, so using parallel processing did not affect the measurement uncertainty. A feature very important when considering changing the method of a measurement.

The application discussed in this chapter, however, is not without flaws. In order to further increase the robustness of this solution a more complete handling of identifying voltage reference levels (high, medium and low) and location of state

transition times is needed. Significant progress toward achieving the second of these on GPUs has been reported elsewhere [4, 24], and it should be possible to combine those methods with the one presented in this chapter. The handling of these two tasks in the current application is possibly sensitive to noise, glitches and runt pulses. Any sensitivity to this phenomena, however, did not present itself during testing.

In addition to become more compliant with the IEEE Standard 181-2011 [2], we plan on experimenting with ways to build a proper histogram on the GPU more efficiently. The current implementation does not need histograms but to become IEEE Standard 181 compliant. A future implementation will require an efficient method for building histograms on the GPU.

Finally, the presented method is also unable to deal with non-constant clocks. In order to add functionality to recover these non-constant clocks a phase locked loop would be required.

Chapter 5

Conclusions

5.1 Comparison of programming models

In the previous two chapters, two applications were discussed. The common thread between the two applications was the use of the GPU as an accelerator. However, they differed in the programming model used, for the sound simulation CUDA was used and for the jitter analysis Thrust was used. While both programming approaches were able to provide the necessary acceleration for each application downsides, drawbacks and trade-offs revealed themselves throughout the process. This raises an important question: which programming approach should be used?

The first concern that presented itself throughout both of the applications is the structure of data which will be used on the GPU. Because of the simple design of the compute cores present on the GPU, hardware functions which a programmer might be accustomed to having on the CPU are not present on the GPU. This means that data structures which rely on use of control flow primitives (primarily if/else) or heavy use of pointer indirection will suffer a performance penalty when used on the GPU. To maximize performance data structures should be designed with the optimal memory access patterns of the GPU in mind. This leads to flat data structures, structs of arrays rather than arrays of structs and lots of "wasted" work. Interestingly, the idea of allowing "wasted" work can be difficult at first for many programmers, but is often key to maximizing the throughput of the hardware.

CUDA itself also presents its own design challenges. Careful ordering of kernel

launches and memory transfers can increase the utilization of the hardware without changing the code of the kernel. Determining the optimal number of threads to launch a kernel can present a problem. However once these challenge and issues are handled, CUDA offers the most raw performance available for GPGPU programming. CUDA makes available all of the features of the underlying hardware, which depending on the problem being solved may be a deciding factor.

Thrust can shield the application programmer from these concerns by providing data structures and algorithms that are abstracted far enough from the underlying hardware. The primary data structure provided by Thrust is a managed array. Memory transfers from the host to the GPU and GPU to the host are hidden from the application programmer using idiomatic C++ code. This design decision causes most Thrust code to fit into the optimal memory access patterns for the hardware. Thrust's algorithms have also been optimized heavily to properly utilize the hardware as completely as possible. Unfortunately, Thrust is unable to use a major performance feature which is available to CUDA, streams of concurrent execution. This can lead to Thrust applications leaving portions of the GPU unused because the scheduler is unable to concurrently execute tasks.

Ultimately, the question that decides whether the trade-offs of Thrust or CUDA makes more sense is; what matters to the application? Is the application mostly done and only a quick addition of some acceleration is needed? Then perhaps Thrust is a better fit as the provided algorithms mirror the STL and are extremely fast. Is throughput the only concern? Then CUDA is the best option, since it is able to squeeze the most performance from the GPU. In the end, it can become the classic trade-off of the cost of the engineer's time versus the cost of the extra time the application takes to run. However for most purposes, Thrust can provide a useful prototype and if extra speed is required then particularly slow parts can be rewritten in CUDA.

5.2 Summary

This thesis presented two applications which used GPU acceleration for two different tasks. The two applications solved problems that were well suited to the computing model of the GPU. In the process of developing each application the characteristics of the programming model used were considered. This process also revealed the short comings of each approach. While CUDA gives the application programmer much more control over the power of the hardware, it also has somewhat rigid requirements for programming style. If these requirements are not followed the resulting code runs much slower than the programmer would expect. Thrust on the other hand, gives the programmer access to algorithms which would otherwise be difficult to implement in exchange for losing access to memory and scheduling features. While losing access to these features does cause an algorithm written in Thrust to be slower than one written in CUDA the ease of implementing the algorithm in Thrust may make Thrust the more appealing option.

The two applications presented also showed promising results and represent fertile ground for future work. The sound simulation application could be expanded to use multiple computer and multiple GPUs or to more accurately model the physical world with enhanced models for absorbence or a higher resolution grid. The jitter analysis could be expanded to more closely follow the IEEE standard or be adapted to run on a system on a chip (SOC) suitable for embedding in a larger device. In the end both applications showed the strengths of not only the GPU as a platform of computing, but also the strength of the SIMD programming model to greatly parallelize some applications.

Bibliography

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[2] IEEE standard for transitions, pulses, and related waveforms. *IEEE Std 181-2011 (Revision of IEEE Std 181-2003)*, pages 1 –71, 6 2011.

[3] Agilent. Agilent technologies - HDMI backgrounder, May 2011. `http://www.agilent.com/about/newsroom/tmnews/background/hdmi/`.

[4] L.A. Barford. Parallelizing small finite state machines, with application to pulsed signal analysis. In *Instrumentation and Measurement Technology Conference (I2MTC), 2012 IEEE International*, pages 1957–1962. IEEE, 2012.

[5] C. J. Bashe, W. Buchholz, G. V. Hawkins, J. J. Ingram, and N. Rochester. The architecture of IBM"'s early computers. *IBM Journal of Research and Development*, 25(5):363–376, 1981.

[6] R. G. Belleman, J. Bédorf, and S. F. Portegies Zwart. High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in CUDA. *New Astronomy*, 13(2):103–112, 2008.

[7] N. Carter, C. Lao, J. Sandmel, C. Woolley, and A. Eddy. EXT transform feedback, September 2013. `https://www.opengl.org/registry/specs/EXT/transform_feedback.txt`.

[8] D. Chirkin. Photon tracking with GPUs in icecube. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 725:141–143, 2013.

[9] C. Crassin. OpenGL geometry shader marching cubes, 2007.

[10] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[11] S. Eilemann. Volume rendering, 2011. `http://www.equalizergraphics.com/documents/design/volumeRendering.html`.

[12] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[13] K. Fok, T. Wong, and M. Wong. Evolutionary computing on consumer-level graphics hardware. *IEEE Intelligent Systems*, 22(2):69–78, 2007.

[14] K. Gee. Introduction to the direct3D 11 graphics pipeline. *Microsoft Corporation*, 2008.

[15] K. Gray. *Microsoft DirectX 9 programmable graphics pipeline*. Microsoft Pr, 2003.

[16] S. Green. The openGL framebuffer object extension. In *Game Developers Conference*, volume 2005, 2005.

[17] J. Groff. An intro to modern opengl. chapter 1: The graphics pipeline, April 2010. `http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html`.

[18] Hewlett-Packard. Standard template library programmer's guide, 1994. `https://www.sgi.com/tech/stl/table_of_contents.html`.

[19] R. Hoang. Wildfire simulation on the GPU. Master's thesis, University of Nevada, Reno, 2008.

[20] R. V. Hoang, M. R. Sgambati, T. J. Brown, D. S. Coming, and F. C. Harris Jr. Vfire: Immersive wildfire simulation and visualization. *Computers & Graphics*, 34(6):655–664, 2010.

[21] M. Ikits, J. Kniss, A. Lefohn, and C. Hansen. Volume rendering techniques. *GPU Gems*, 1, 2004.

[22] W. Kahan. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[23] J. Kessenich, D. Baldwin, and R. Rost. The openGL shading language, January 2014. `http://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf`.

[24] V. Khambadkar, L. Barford, and F.C. Harris. Massively parallel localization of pulsed signal transitions using a GPU. In *Instrumentation and Measurement Technology Conference (I2MTC), 2012 IEEE International*, pages 2173–2177. IEEE, 2012.

[25] D. Kirk. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[26] M. J. Koop, Wei Huang, K. Gopalakrishnan, and D. K. Panda. Performance analysis and evaluation of PCIe 2.0 and quad-data rate infiniband. In *High Performance Interconnects, 2008. HOTI '08. 16th IEEE Symposium on*, pages 85 –92, aug. 2008.

[27] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[28] M. Livesey, J. F. Stack, F. Costen, T. Nanri, N. Nakashima, and S. Fujino. Development of a CUDA implementation of the 3d FDTD method. *IEEE Antennas & Propagation Magazine*, 54(5):186–195, 2012.

[29] T. Loken, L. Barford, and F. C. Harris. Massively parallel jitter measurement from deep memory digital waveforms. In *Instrumentation and Measurement Technology Conference (I2MTC), 2013 IEEE International*, pages 1744–1749. IEEE, 2013.

[30] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck. GPGPU: general-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 208. ACM, 2006.

[31] W. Maichen. *Digital Timing Measurement: From Scopes and Probes to Timing and Jitter*, chapter 9.3. Springer, 2006.

[32] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 896–907. ACM, 2003.

[33] G. E. Moore. Cramming more components onto integrated circuits. 1965.

[34] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE micro*, 30(2):56–69, 2010.

[35] NVIDIA. NVIDIAs next generation CUDA compute architecture: Kepler GK110. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[36] NVIDIA. CUDA concurrency & streams, 2011. `http://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf`.

[37] NVIDIA. CUDA C programming guide, July 2013. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[38] NVIDIA. Tuning CUDA applications for kepler, July 2013. `http://docs.nvidia.com/cuda/pdf/Kepler_Tuning_Guide.pdf`.

[39] NVIDIA. Thrust quick start guide, February 2014. `http://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf`.

[40] L. Nyland, M. Harris, and J. Prins. Fast n-body simulation with CUDA. *GPU gems*, 3(1):677–696, 2007.

[41] M. L. Overton. *Numerical computing with IEEE floating point arithmetic*. Siam, 2001.

[42] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[43] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008.

[44] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, pages 3–3. San Diego, California, 2009.

[45] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[46] D. Schreiner. *Open GL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison Wesley, 2013.

[47] J. Sheaffer and B. M. Fazenda. FDTD/K-DWM simulation of 3D room acoustics on general purpose graphics hardware using compute unified device architecture (CUDA). *Proc. Institute of Acoustics*, 32(5), 2010.

[48] B. W. Silverman. *Density estimation for statistics and data analysis*, volume 26. Chapman & Hall/CRC, 1986.

[49] A. Stepanov and M. Lee. *The standard template library*, volume 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304, 1995.

[50] C. KW. Tam and J. C Webb. Dispersion-relation-preserving finite difference schemes for computational acoustics. *Journal of computational physics*, 107(2):262–281, 1993.

[51] C. J. Webb and S. Bilbao. Computing room acoustics with CUDA-3d FDTD schemes with boundary losses and viscosity. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 317–320. IEEE, 2011.

[52] A. X. Widmer and P. A. Franaszek. A DC-balanced, partitioned-block, 8B/10B transmission code. *IBM Journal of research and development*, 27(5):440–451, 1983.

[53] B. Yee. *Using secure coprocessors*. PhD thesis, Carnegie Mellon University, 1994.

[54] M. R. Zunoubi, J. Payne, and W. P. Roach. CUDA implementation of-FDTD solution of maxwell's equations in dispersive media. *Antennas and Wireless Propagation Letters, IEEE*, 9:756–759, 2010.