AUDIO ON THE GPU: REAL-TIME TIME DOMAIN CONVOLUTION ON GRAPHICS CARDS

A Thesis
by
ANDREW KEITH LACHANCE
May 2011

Submitted to the Graduate School
Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

May 2011
Department of Computer Science

AUDIO ON THE GPU: REAL-TIME TIME DOMAIN CONVOLUTION ON GRAPHICS CARDS

A Thesis
by
ANDREW KEITH LACHANCE
May 2011

APPROVED BY

ABSTRACT

AUDIO ON THE GPU: REAL-TIME TIME DOMAIN
CONVOLUTION ON GRAPHICS CARDS

Andrew Keith LaChance

M.S., Appalachian State University

Thesis Chairperson: Rahman Tashakkori

The architecture of CPUs has shifted in recent years from increased speed to more cores on the chips. With this change, more developers are focusing on parallelism; however, many developers have not taken advantage of a common hardware component that specializes in parallel applications: the Graphics Processing Unit (GPU). By writing code to execute on GPUs, developers have been able to gain increased performance over the traditional CPU in many problem domains, including signal processing.

Time domain convolution is an important component of signal processing. Currently, the fastest process to perform convolution is frequency domain multiplication. In addition to being more complex, inconsistencies such as missing data are difficult to solve in the frequency domain. It has been shown that executing frequency domain multiplication on GPUs improves performance, but there is no research for time domain convolution on GPUs.

This thesis provides two algorithms that implement time domain convolution on GPUs: one algorithm is for computing convolution all at once and another is designed for real time computation and playing the results. The results from this thesis indicate that using the GPU significantly reduces processing time for time domain convolution.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my wife who has stood by me during this very long process and had to deal with the strain of living in two cities while I finish my degree. I could not have done this without her continuous support.

My thanks also extend to my thesis advisor, Dr. Rahman Tashakkori, without his help and guidance this thesis would not have been possible.

I would also like to thank Dr. Barry Kurtz and Dr. James Wilkes, my thesis committee members, for their valuable advice.

Table of Contents

Chapter 1 - Introduction

Traditionally, CPUs have been the only target architecture of software developers. With the advent of three-dimensional graphics and multimedia applications, however, dedicated processors for graphics (GPUs) have become a standard part of most computers. At first, GPUs were only used for graphical rendering; gradually they became powerful enough that developers saw the potential use of GPUs as parallel processors. Software that is highly parallel derives a performance benefit by utilizing the GPU instead of exclusively using the CPU. The GPU designer company NVIDIA took advantage of the fact that developers were using their GPUs for general purpose problems and introduced new hardware to their GPUs to make them easier to program.

In 2007, NVIDIA introduced Compute Unified Device Architecture (CUDA) as a solution for general purpose programming on GPUs [5]. CUDA allows developers to write code in traditional programming languages, such as C and C++, to be compiled and executed on GPUs. Prior to CUDA, problems needed to be cast into constructs that the GPUs could understand, such as a data array being cast to a two-dimensional array (texture) and accessed as such. Because GPUs were already designed to be highly parallel processors, CUDA made it possible for programmers to develop non-graphics applications that exploited data parallelism to execute on GPU hardware.

Data parallelism is a property of some algorithms in which the calculations on each datum in the problem are independent of the results of the other calculations. Therefore, the calculations on different data can be executed simultaneously. Applications that contain inherent data parallelism include matrix multiplication, electrostatic potential map calculations, as well as convolution, which is one of the topics of investigation of this thesis.

According to [6], the convolution in the time domain of two functions, a Dry Signal and an impulse response, is defined as (1).

$$h(n) = \sum_{m=0}^{n} f(m)g(n-m) \tag{1}$$

In this definition, n is the total number of samples in the impulse response, m is the current sample number, $f(n)$ is the Dry Signal (the one to add reverberation to), $g(n)$ is the impulse response (the reverberation of the room), and $h(n)$ is the resulting Wet Signal (the Dry signal with reverberation added). In audio terms, every sample in the impulse response represents the amplitude of a single echo. By performing convolution on these signals, reverberation is applied to the Dry Signal. The resulting Wet Signal has the reverberation characteristics of the room in which the impulse response was acquired. In other words, the Wet Signal would sound like the Dry Signal being played in that room.

Sequencing software to create and manipulate audio is popular for musicians, sound engineers, and those who are interested in creating music digitally. The number of effects that can be added to a signal by the audio creation software within a usable time frame is dependent upon the speed of the CPU. If the effects can be parallelized and executed on a GPU, however, it then becomes possible to apply more effects at once since the computation required by the CPU has been reduced.

Although convolution can be parallelized, this is not true for all audio effects. In some algorithms, such as equalization, the value of the current sample depends upon the value of a previous sample or samples. Fabritius investigated the GPU implementation of four audio effects: equalization, delay, convolution, and compression [3]. He concluded that of the four effects tested, convolution/reverb was the only algorithm that had a clear advantage when using the GPU, compared to that obtained by running exclusively on the CPU.

Fabritius focused on the fastest known way to perform convolution. This approach was to transform the data to the frequency domain using the Fourier Transform, and thus signal data was stored as frequency data instead of time data. After that, frequency domain multiplication, equivalent to time domain convolution, was performed using the frequency data. Finally, the result was transformed back to the time domain using the Inverse Fourier Transform [3]. Unfortunately, Fabritius did not compare time domain convolution with frequency domain multiplication on GPUs.

Time domain convolution has the benefit of being far simpler to analyze and understand. In addition, as a sliding average, it has applications in statistics. In particularly, the Stanford Exploration Project [13] lists five complications that are more difficult to solve in the frequency domain:

- time boundaries between past and future
- delimiting a filter
- erratic locations of missing data
- time-variable weighting
- time-axis stretching.

Furthermore, there may yet be undiscovered advantages to keeping the data in the time domain.

In addition to its focus on time domain convolution, this thesis investigated different optimization strategies. For GPU programming, there are many different options for optimization, but not all of them work well on every type of problem. Presently, there are no tried-and-true methods for determining whether or not a certain optimization will actually provide a benefit other than to implement and test that optimization strategy [5]. This thesis investigated using texture memory, shared memory, prefetching, loop unrolling, and removing some error-checking by moving that responsibility to the CPU. This will benefit developers wanting to optimize applications similar to signal processing by allowing them to focus only on the optimizations that have the best chance of leading to improved performance.

Chapter 2 - Background Information

Although the same programming language (C/C++) is used for programming Central Processing Units (CPUs) and Graphics Processing Units (GPUs), the methods in which these processors are programmed are quite different. CPUs are traditionally sequential architectures in which the instructions are ordered and must be executed in the same order. Current hardware now allows CPUs to execute instructions out of order, and overlap the execution of individual instructions, as long as the results produced are the same as that of sequential execution. This is a form of parallelism known as fine-grain parallelism. CPU design is currently moving towards a more coarse-grain parallel design with up to eight processing cores. Eight cores allow up to eight threads to run simultaneously using a Multiple Instruction Multiple Data (MIMD) strategy.

2.1 – GPU Programming

GPUs utilize parallel architectures in which multiple processors are executing the same instruction on different data. This strategy is known as Single Instruction Multiple Data (SIMD). SIMD processors can be used to exploit the data parallelism of algorithms that perform the same operation on each datum of a typically large set of data. Figure 2.1 describes a SIMD architecture and illustrates how instructions are shared between every thread. An example of an algorithm that has data parallelism is an add operation of two arrays that adds elements from the same index together and stores the result in another array at that index.

Most GPU applications follow a particular code layout: CPU code is used to set up the application and data, memory is allocated and copied using API calls, a GPU kernel (complete code that runs on a GPU) is launched to operate on the data and upon completion, the results are copied back to the main memory. The two copy steps and the kernel launch would replace the calculation step of traditional CPU code, especially if the code is being modified to run on GPUs.

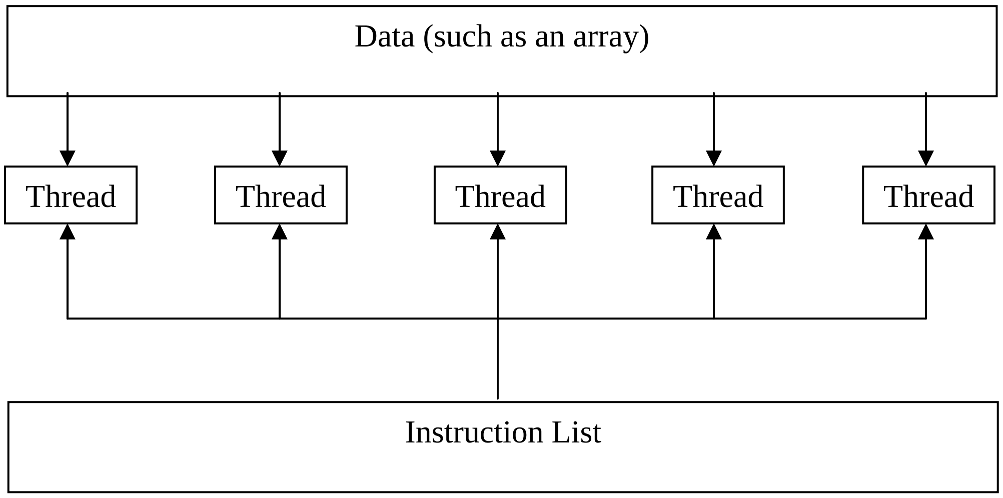

Figure 2.1 – SIMD Architecture

A GPU kernel launch is different from a typical application launch. Along with any parameters passed to a kernel, such as the address of the data in GPU memory, a launch configuration must also be provided. A launch configuration tells the GPU how to organize its runtime resources. At least two parameters must be specified in the launch configuration: the number of threads per block and the number of blocks per grid.

A thread on the GPU is very similar to a thread on a CPU; however, control can be switched among threads much faster on GPUs due to the specialized hardware. Groups of threads organized into blocks can have one, two, or three dimensions of threads, configured at runtime. The number of dimensions used is application-dependent. For example, image or video applications would probably find it beneficial to use two dimensions since images and video frames are usually organized as two-dimensional arrays.

Threads in a block share resources. For example, each thread does not have its own register file. Instead, a large register file exists on a multiprocessor and threads use registers from there. All the blocks reside in a one or two dimensional grid, defined at runtime. A grid is the highest level in the hierarchy, thus when a grid has completed execution, the kernel has also completed. Figure 2.2 illustrates an example of a grid with threads in three dimensions and blocks in two dimensions.

A thread and the data upon which the thread operates are identified via the built-in variables *threadIdx*, *blockIdx*, *blockDim*, and *gridDim*. These variables are assigned at kernel launch based on the launch configuration. The *threadIdx* variable holds the index of the thread. It has three components: *x*, *y*, and *z*, accessed using dot notation (for example *threadIdx.x*), which refer to the three dimensions in which threads can be launched. The *blockIdx* variable holds the block index and has two components, *x* and *y*, to correspond to the two dimensions in which blocks can be launched. The *blockDim* variable holds the number of threads per block and *gridDim* holds the number of blocks. Together, these variables can be used to determine an index for that thread. In the simplest case of blocks and threads in one dimension, an index for data can be determined by *blockDim* * *threadIdx.x*. The next index for that thread is the sum of the current index and *blockDim* * *gridDim*.

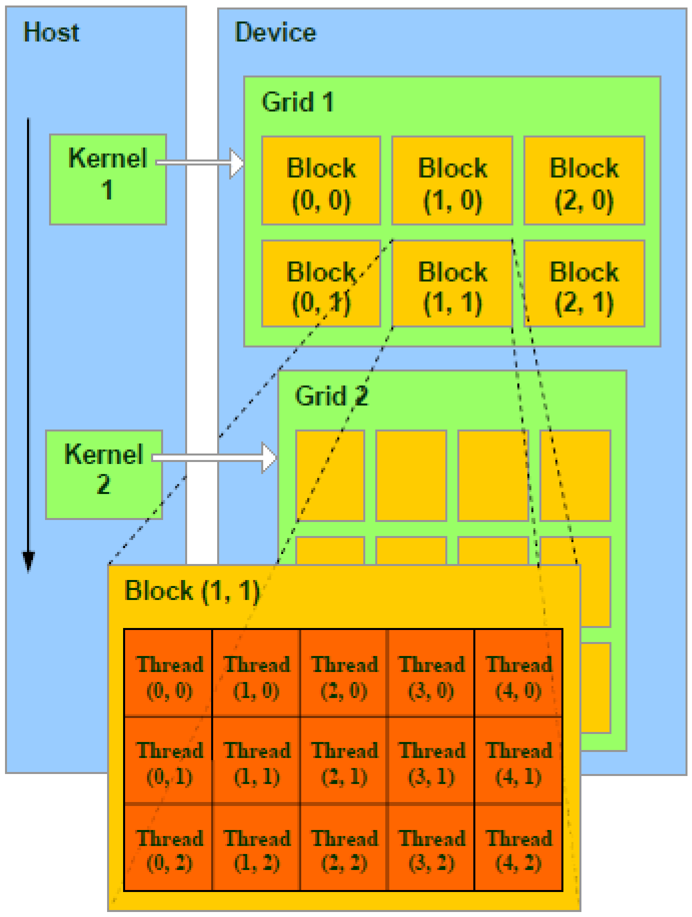

Figure 2.2 – Configuration of two dimensions of both threads and blocks [15]

GPUs do not execute threads one at a time; instead, the hardware divides the threads into warps. A warp is a group of threads that are executed simultaneously. For the most recent hardware revision, compute capability 2.0, the warp size is 32 threads. Because the threads in a warp are executed at the same time and in a SIMD strategy, the instruction that every thread in a warp executes must be the same. When some threads in a warp execute the *then* clause and some execute the *else* clause, thread divergence occurs, in which the *then* clause is executed, followed by the *else* clause. In such a case, only the threads that have evaluated the expression to be true execute the instructions in the *then* clause; the other threads are disabled. The opposite happens for the *else* clause, and finally all threads are enabled and continue to execute. Thread divergence can slow processing and developers should keep this in mind when designing an algorithm. For example it would be disadvantageous to have an algorithm that had a conditional statement depending on the parity of a thread index.

Figure 2.3 illustrates how warps are distributed across thread blocks.

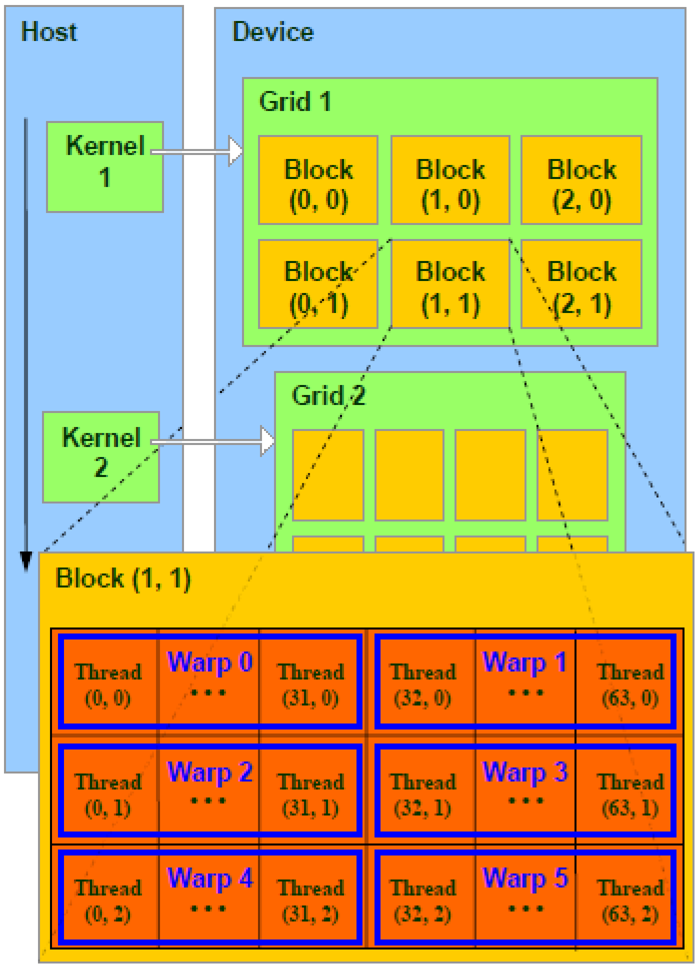

Figure 2.3 – Difference between thread blocks and warps [15]

The difference between the grouping of threads into blocks and threads into warps is that thread blocks are logically executed in parallel while warps are physically executed in parallel. Figure 2.3 also illustrates this difference.

2.2 - Digital Audio

A sound as heard by our ears reflects the changes in pressure caused by vibrations in the air. These pressure changes in time are compression waves and describe the audio in the time domain. The ear turns these pressure differences into frequency, which is heard as pitch. Audio is stored on a computer as digital audio by sampling the sound waves. This means that at a certain time interval, the pressure is measured and stored. The time interval is determined by the sampling rate, which is based upon what frequencies should be stored. The Nyquist Theorem states that a sampling rate of twice the highest distinct frequency to be stored is needed in order to retain all information. An average person can hear between 20 Hz and 20 KHz, which means that the sampling rate must be at least 40 KHz. However, due to imperfect recording equipment, a small threshold is added to the sampling rate to counteract the imperfections of the recording equipment. For example, a sampling rate of 44.1 KHz, or 44,100 samples per second (as was decided upon by Sony and Phillips), has become the de facto standard. This means that approximately every 23 microseconds, or 2.3×10^{-5} seconds, a measurement of the pressure is recorded and stored.

The standard audio format is Pulse Code Modulation (PCM), in which sound is stored as amplitudes (the pressure level) with a sampling rate and bit depth. The bit depth determines the number of different pressure values that can be stored. A common bit depth is 16 bits per sample and measured at a sampling rate of 44.1 KHz.

A real time system is one in which failure occurs when a deadline is missed. Audio applications are known as soft real time systems, in the sense that if a deadline (for example, every 23 microseconds) is missed (sending audio to a speaker), there is a gap in the audio that the user can hear as an interruption, but it does not lead to a catastrophic failure as no actual harm occurs – the audio can still be played. The algorithm devised in the thesis is considered to be running in real time such that immediately upon completion of any calculated Wet data, it is sent to the speakers, and that there are no stops in the audio for any amount of time until all the audio has been played.

Chapter 3 - Related Work

GPUs have been used to solve problems in many domains, including MRI analysis and molecular visualization [5]. While a large chunk of research has been in visualization, there has also been research into signal processing on GPUs.

In 2008, Cardinal et al. used GPUs for computing acoustic likelihoods in a speech recognition system [2]. Cardinal's team implemented their algorithm using dot product operations, which GPUs can process efficiently, and concluded that their GPU implementation was 5 times faster than the CPU version, which led to a speedup of 35% on a large vocabulary task.

Kerr et al. developed an implementation of the Vector Signal Image Processing Library (VSIPL) for GPUs in 2008 [4]. This group implemented two versions: a time-domain finite impulse response filtering runtime and a fast Fourier Transform runtime. They achieved an 82 times speedup for their time-domain runtime and a 14.5 times speedup for their fast Fourier Transform runtime.

Nieuwpoort and Romein developed a GPU version of a software correlator to filter out noise from radio astronomy signals [8]. A correlator is a streaming, real-time application that is more I/O intensive than typical applications developed for multiple-core architectures (such as GPUs). They found that using an NVIDIA GPU gave them a speedup of 6.3 times over a CPU.

In 2010, Nexiwave announced that the next version of their speech indexing application will be accelerated by using NVIDIA GPUs [7]. The application by Nexiwave for speech indexing has the ability to search for and locate spoken words in media, such as videos. They claim to have achieved a speedup of 70 times over the existing solutions.

In 2009, Frederik Fabritius described the use of GPUs for audio processing, including convolution as reverberation [3]. His goal was to determine whether or not GPUs were suitable for convolution and to investigate how they compare with CPUs in performance, specifically using the fastest known approach to convolution. This convolution method involves using the frequency domain in which a signal is described in terms of frequency and power. Convolution is performed by multiplying the spectra of the two signals. Fabritius concluded that in all practical cases, it is either as fast, or faster, to compute reverberation on GPUs.

Chapter 4 – GPU Optimization Strategies

Software developers have been writing code for CPUs for a while now. They have gotten used to the underlying architecture and have developed numerous strategies for speeding up their programs. GPUs have different hardware from CPUs and the optimization strategies are more specialized. While there are some strategies, such as memory coalescing, that will always work, some depend on the specifics of the problem and the algorithm used. Many optimization strategies involve utilizing the GPUs memory or memories in an efficient way. Figure 4.1 shows the layout of memory on an NVIDIA CUDA GPU. Each block has its own register file and shared memory, while every block has access to the same constant and global memory.

4.1 – Memory Coalescing

One tried-and-true method to improve efficiency is to make sure that most, if not all, memory accesses to global memory are coalesced. Global memory loads and stores, under certain requirements, can be reduced to a single instruction [9]. Recall from chapter 2 that the smallest execution unit is a warp (or 32 threads). Depending on the compute capability of the GPU, memory instructions are either executed by a half-warp or a full warp. This means that it is possible for 16 or 32 load or store instructions to be reduced to a single instruction.

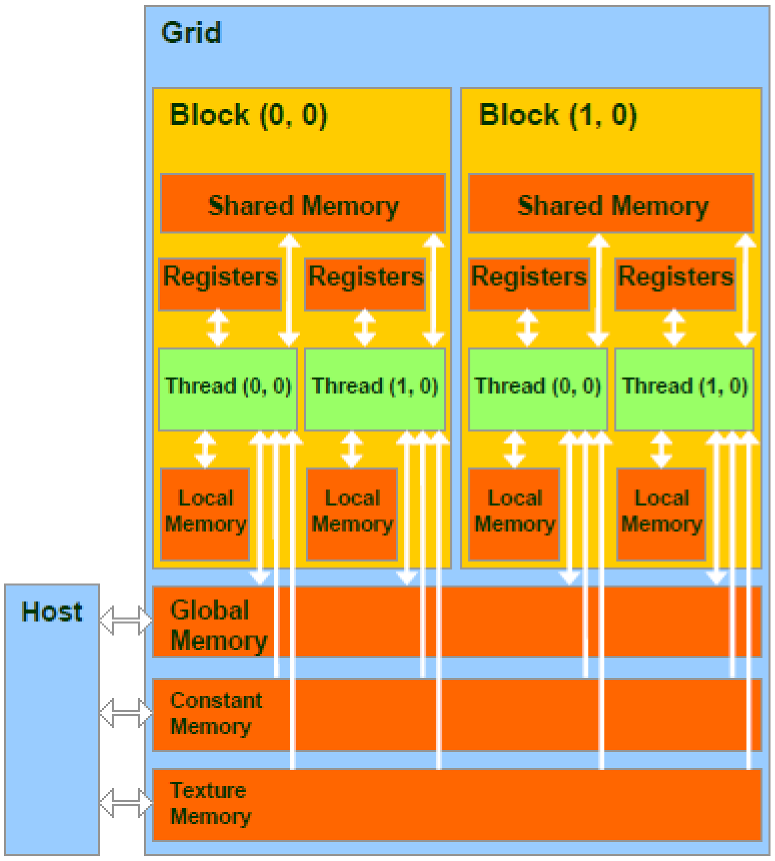

Figure 4.1 – Memory layout on CUDA GPUs [15]

Coalescing is achieved when all threads of a half-warp or full warp (depending on the hardware version or compute capability) access different, but contiguous, areas of memory. The only other option is for all the threads to access the same memory location. For example, the following code segment has coalesced access:

int index = threadIdx.x;

float element = globalArray[index];

This segment has coalesced access because each thread is accessing a different element of *globalArray*, but are contiguous in memory. Thread 0 accesses element 0, thread 1 accesses element 1, and so on. Since this can be reduced to a single instruction, global memory is only accessed one time. This removes up to 31 global memory accesses, and with each global memory access taking hundreds of clock cycles, hence saves a significant amount of time [9].

4.2 – Shared Memory

Unlike CPU architectures, which include a hardware cache for memory, GPU architectures have not had a hardware cache. Only the most recent GPUs have hardware caching (compute capability 2.0). This means that every global access will always go to the global memory. However, GPUs do have a type of cache on-chip which is entirely software managed - shared memory [9].

Shared memory exists for threads in the same block, which allows all threads in a block to share information. The latency in accessing data from shared memory is about 100 times lower than that of accessing data from global memory [9]. If the same data is used more than once, significant gains can be realized by storing the data in shared memory rather than fetching it from global memory each time.

Shared memory is divided into banks so that all threads can access shared memory at the same time. For example, element 0 is located in bank 0, element 1 in bank 1, and so on. The number of banks differs by compute capability (32 for compute capability 2.x and 16 for lower) [9]. This means that in order to get the most out of shared memory, it is best to access data in such a way that every thread accesses a different bank, or all threads access the same address (thus one bank can broadcast its data to every thread).

4.3 – Texture Memory

Textures are images that are used to decorate polygons in graphics rendering, such as a texture of a tabletop looking like wood. Accessing textures is a very common operation, and GPU hardware has texture units to help speed up that process by optimizing reads by 2D spatial locality [9]. In general purpose computing, this provides a type of hardware cache for memory to which the GPU will not write (since textures do not change), especially prior to compute capability 2.0 when there was no hardware for caching. In cases where memory has been “bound” as a texture and its access is generally in a way to take advantage of 2D spatial locality, it is possible to achieve a higher bandwidth.

4.4 – Pinned Memory

Pinned memory is memory that has been specifically allocated so that it cannot be swapped to disk [12]. This makes it safe for the operating system to allow the application to access the physical address space of the memory. In the case of GPU programming, this allows the GPU to use Direct Memory Access (DMA) to copy data to and from the CPU. The CPU is not involved with DMA, and allows a potential speed increase due to not needing to execute memory copying instructions. Even if a developer does not allocate pinned memory and performs a copy, the driver copies the memory to a pinned memory region and then moves it to the GPU. Therefore, a copy from pageable memory occurs twice – first from the pageable memory to temporary pinned memory, and then from pinned memory to the GPU.

4.5 – float2/float4 Data Type

The float2 and float4 data types are built-in vectors, represented as structures that contain two or four floats, respectively. Because they are built-in, it may be possible to slightly optimize memory accesses for accessing two or four floats, depending on the application, access pattern, and usage.

4.6 – Control Flow and Padding

Branching instructions are executed differently on GPUs than CPUs. The reason is because of the minimum number of threads that are executed at the same time (the warp size). The branching condition is checked by all threads. If the results do not agree for all threads in the warp, the execution paths need to be serialized [10]. This means that all statements in the clause, both that are executed when the condition is true and when it is false, are executed serially. This increases the total amount of instructions executed by the warp, and can drastically reduce the execution speed.

A possible way to reduce the number of control flow statements is to use array padding. Padding an array changes the size of the array but does not change the resulting value of calculations on the array. For example, with convolution, an array can be padded with zeroes without changing the result. This is because the multiplication step will produce a value of zero, and then adding zero to the sum will not change the sum. Padding works in this case by completely removing all bounds checking in the convolution calculation. Normally, convolution will go out of bounds on the Dry array at both ends. However, adding padding to both sides allows all accesses to stay in bounds, thus removing the need to check for negative accesses and greater-than-length accesses.

Padding arrays to become a certain size or a certain multiple of a size can also be beneficial. A good example is to allow chunks of data to fit in shared memory with a number of elements equal to the number of threads per block.

4.7 – Streams

Streams can be thought of as self-contained parts of an application. As GPUs' hardware have advanced, it became possible to execute kernels at the same time as copying memory to and from the GPU. In order to accomplish that, however, a new method had to be employed to separate the execution from the memory copying. This method uses streams. A stream can be copying data to the GPU at the same time that another stream is executing its kernel. This allows some of the work to overlap and can save time. The only caveat is that streams must operate only on pinned memory.

Starting with compute capability 2.0, multiple streams can be executed on the GPU simultaneously, provided that resources are available. Therefore, if a kernel does not use a lot of resources, or if the GPU has an abundance of resources, multiple streams can execute at the same time where previously queued streams had to wait. A common situation is when one stream is finishing up and is not using as many resources, in which case the next stream can start using the resources that the previous stream does not need anymore.

4.8 – Launch Configuration

A simple way to optimize a kernel is to write it so that it can run with any configuration of threads per block, number of blocks per grid, and number of grids. This allows the programmer to try different launch configurations to find the most suitable one for the program. A tool to help is known as the CUDA Occupancy Calculator, provided by NVIDIA. It is a spreadsheet in which the programmer fills in information about the kernel and it shows them the limitations of the kernel and potentially new configurations to try. It also shows the occupancy of each multiprocessor, which is defined as the ratio of the number of active warps to the maximum number of active warps [9]. A higher occupancy does not always yield better performance – there is a point at which a higher occupancy does not increase performance.

4.9 – Prefetching

Another use for shared memory is to use the prefetching technique. Prefetching is an algorithm in which the next chunk of data is read from memory before the data is actually needed. The goal is to overlap the memory reading instructions (more specifically, the delay in accessing the data) with instructions performing calculations on the data already retrieved with the hope of reducing the time spent waiting on data from memory. However, prefetching requires twice the amount of shared memory (to store the data to do the calculations on and the data to be prefetched) and may use more registers, which may reduce the number of warps that can be executed simultaneously.

Chapter 5 – Methodology

The algorithm used on this thesis for the implementation of time domain convolution on GPUs can be broken down into three major steps: 1) acquiring the sampled data of the Dry and impulse response Signals, 2) preparing the devices, and 3) performing the convolution. Figure 5.1 illustrates the computation process. The sampled data are broken into chunks guaranteed to be a multiple of the number of threads per block. A stream will process one chunk of data with the number of streams determined at compile time. The GPU will hold a circular buffer into which chunks of sampled Dry data will be written. After streams finish convolution on their chunk, the chunk gets copied back to the CPU where the audio data are played.

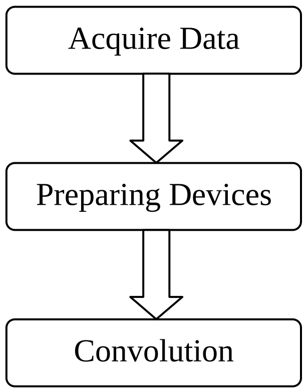

Figure 5.1- High-level algorithm design

5.1 – Acquisition of Sampled Data

The first step in the process is to acquire the samples for both sounds: the Dry Signal and the impulse response. The standard WAVE file is selected for the ease of obtaining the signal data. The raw data read from the WAVE file must be converted to floating point, mapped as a *float2* type, and padded with zeroes before it can be used. Figure 5.2 illustrates this process.

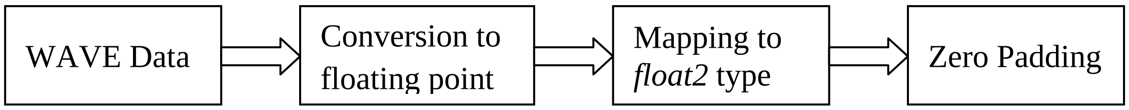

Figure 5.2 – Conversion process from WAVE data to usable data for convolution

5.1.1 – Brief Overview of the WAVE File Format

The data in the WAVE files are split into chunks. There are three main chunks: the RIFF - or Resource Interchange File Format - chunk, the format chunk, and the data chunk. Each chunk has a size field that defines its size. The RIFF chunk defines that the file is a WAVE file. The format chunk describes the properties of the signal, including the number of channels (for example, mono or stereo), the sample rate, the byte rate, the number of bits per sample, and the audio format (usually uncompressed). The data chunk contains the actual sampled data [14]. Figure 5.3 describes the format in detail.

In order to access the sampled data, each chunk must be visited. The RIFF chunk is used to verify that the file is a WAVE file. The fields of the format chunk are read first which define how the audio will be sent to the speakers after convolution. Finally, the sampled data are read in as described by the format chunk, with attention given to the number of channels and bits per sample. If a chunk is read that is not of type format or data, it can be skipped by reading the size field and by skipping the specified number of bytes.

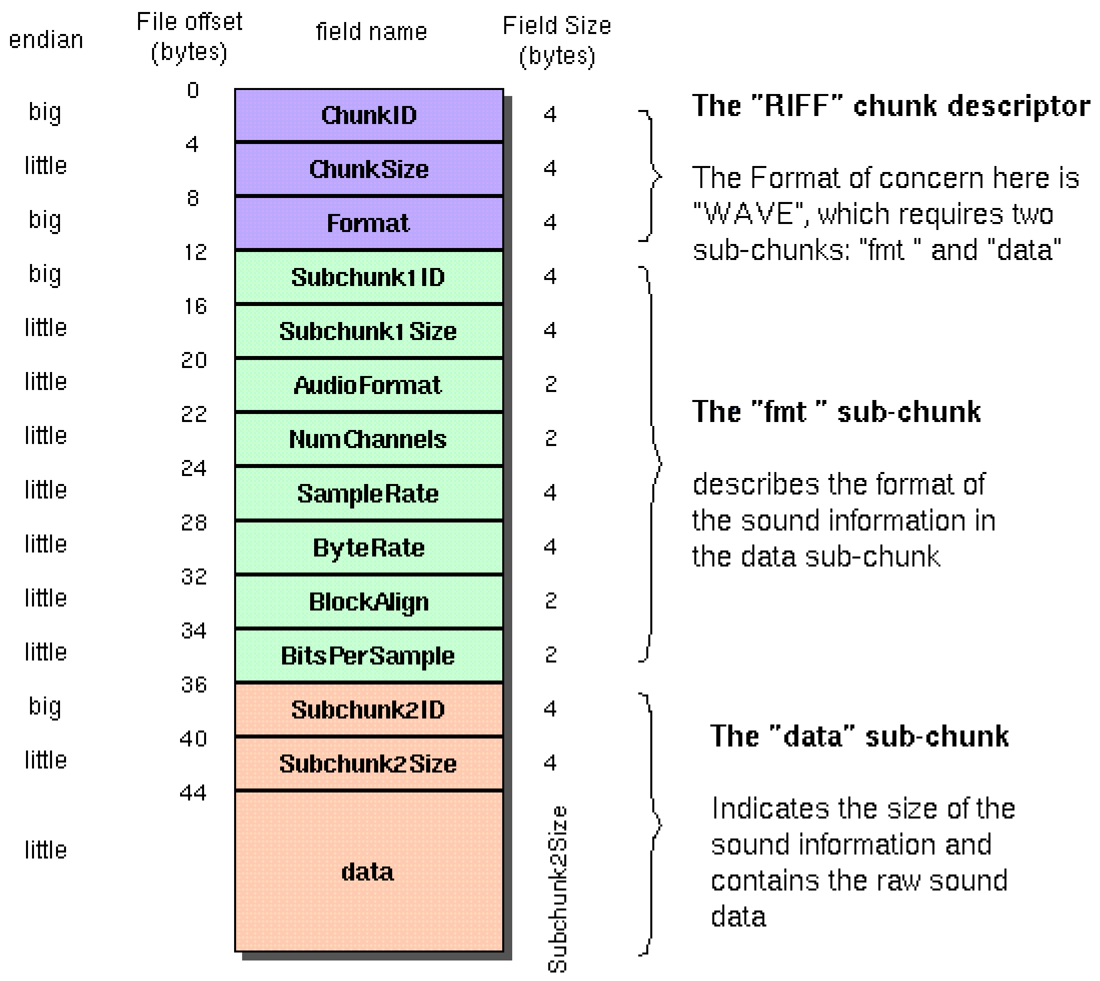

Figure 5.3 – Description of the Wave file format [14].

5.1.2 – Conversion of Sampled Data

In the WAVE format, sampled audio data are stored as integral data types, such as a *char* or *short*. GPUs have excelled in floating point calculations and have dedicated hardware specifically designed for fast floating point computation. Especially useful is a single instruction (known as a fused multiply-add instruction) that multiplies two floating point numbers and adds the result to another. In order to take advantage of this, the data must be converted to a floating point data type before convolution.

5.1.3 – Sample Frame Representation

The audio data are separated into channels. Stereo signals, which are audio signals with a left channel and a right channel, are a common format. A sample frame is a sample from every channel that is played at the same time. Figure 5.4 illustrates an example of a two-channel sample frame. Thus, for stereo audio, a sample frame consists of two samples; one for the left channel, and one for the right. The *float2* data type, a built-in structure that holds two floating point values, was selected to represent one sample frame of data because it maps well to stereo audio. Figure 5.5 illustrates the mapping of the *float2* type to a sample frame.

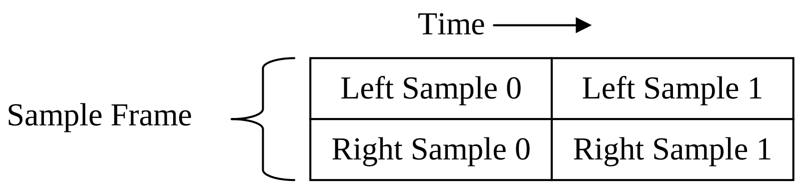

Figure 5.4 – Example of a two-channel sample frame

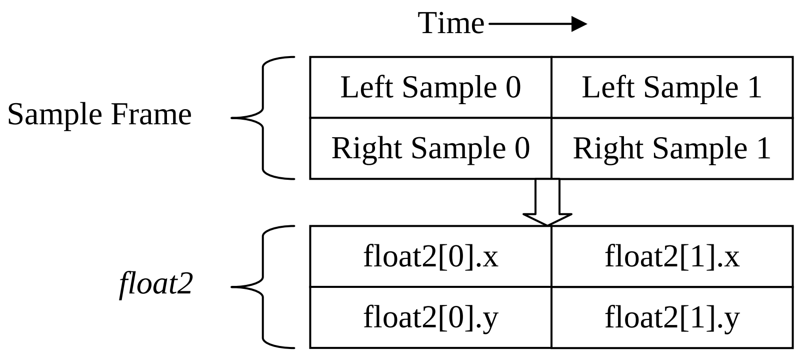

Figure 5.5 – The mapping of a sample frame to the *float2* data type

5.1.4 – Zero Padding

The impulse response and Dry Signals are zero padded to eliminate bounds checking and thus reduce the number of branching instructions in the kernel code. The impulse response data has zeroes added to the tail end to make it a multiple of the number of threads per block. This allows ease of access from shared memory so that each thread of a block can always access and store one element. The Dry Signal has leading and trailing zeroes added such that when performing convolution on the GPU, the indices never become less than zero or greater than the size of the array holding the Dry data. To make sure the indices never go out of bounds, the Dry Signal is padded by one less than the number of sample frames in the impulse response on both ends.

5.2 – Preparing Devices

Before any GPU kernel code can be executed on a GPU, all data on which the kernel operates must to be allocated appropriate space. In addition, the GPU streams need to be defined and the device for playing audio needs to be configured. Figure 5.6 illustrates this process.

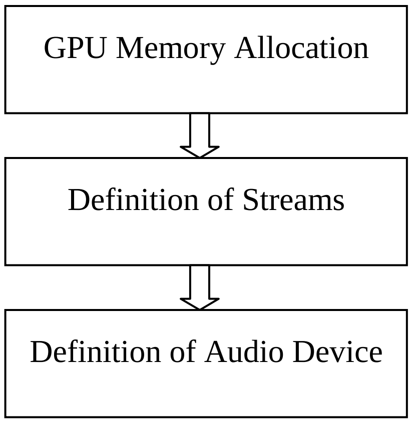

Figure 5.6 – Setup process for convolution

5.2.1 – GPU Memory Allocation

Unlike programs that are developed on the CPU, in the case of programming for GPUs the programmer has explicit access to many types of memory, which vary in size, scope, and purpose. Not including registers, four types of memory, constant, texture, global, and shared, are used in the final algorithm designed for this thesis. Three types are defined in the CPU code.

Constant memory is useful if a kernel needs access to small constant data. For convolution, the length of the impulse response and the length of the buffer for the Dry data do not change. Since the length variables are well within the limit of the constant cache size of 64 KB, it is possible to store them in constant memory. The first read of every element in constant memory will be read from global memory and also stored in the constant memory cache. Each subsequent read will then be routed to the constant memory cache; therefore using constant memory will not increase efficiency in the case of one read [10]. Using constant memory reduces the number of parameters needed for each kernel, making the code simpler.

Texture memory can be used as a type of hardware cache provided that the access pattern that takes advantage of 2D spatial locality. Texture memory is used for the impulse response because of the way the impulse response data is accessed. Every sample of the calculated Wet data needs every sample of the impulse response. This means that every sample of the impulse response is accessed many times, and having some or all of the data in cache is beneficial. After allocation, the impulse response is copied to the GPU.

Global memory is used to store the Dry and Wet sample frames. Since each sample frame of the Wet data is only accessed once to store the calculated value, there is no need to use any special memory. The access pattern of the Dry data does not map well to the properties of any special memory, thus global memory is used. Since compute capability 2.0, a hardware cache exists to handle access patterns that do not map well to the other memories.

The real time convolution algorithm developed for this thesis is designed to use a Dry Signal of arbitrary length; therefore, a method is needed that allows operations on the data either if the entire signal would not fit within global memory or if the entire signal is not known at once, such as if it were streaming across the Internet. The method chosen is to use a circular buffer in global memory such that a signal of any size would fit. The length of the buffer must be chosen carefully, otherwise corruption can occur. The minimum length of the circular buffer must be based upon the number of streams (so that no data currently being used is overwritten) and the number of sample frames per chunk. To calculate one sample frame of the Wet signal, a number of Dry sample frames equal to the length of the impulse response is needed. Because of this, there must always be in memory a contiguous number of sample frames of the Dry signal equal to the length of the impulse response. This means that the total length of the circular buffer also must not be smaller than the length of the impulse response. This problem can be alleviated by allowing each stream a number of buffers of a length equal to the Dry chunk size, as part of the circular buffer.

To calculate the minimum length of the circular buffer, first we determine the total length of the Dry data, l, that must be used in calculations at a single time:

$$l = n * r \tag{2}$$

In this definition, n is the number of streams and r is the number of sample frames per chunk. This represents the number of sample frames that cannot be overwritten because kernels of different streams are currently using this data.

Once the minimum length of the circular buffer has been determined, the number of buffers, *b*, needed for each stream to allow replacement of data is calculated by (3).

$$b = \left\lfloor \frac{(i + l) + (l - 1)}{l} \right\rfloor \tag{3}$$

In this equation, i is the length of the impulse response and l is the total length of the Dry data buffer.

The buffers are arranged in memory sequentially by increasing stream first, then buffer. For example, if there are two streams and two buffers needed, the layout in memory would look like Figure 5.7.

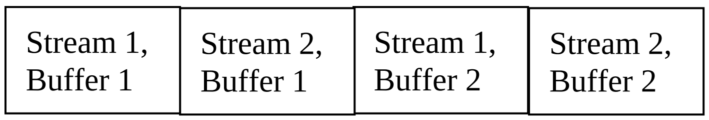

Figure 5.7 – Layout sequence of stream buffers in GPU memory

Finally, the total length, *s*, of the circular buffer is calculated using (4).

$$s = b * l. \tag{4}$$

The buffer is allocated on the GPU of size s and then cleared. Clearing the data achieves the same effect as padding leading zeroes to the signal; when the buffer wraps around, the data being read are zeroes.

5.2.2 –Definition of Streams

The Dry signal is being copied to GPU memory at the same time the kernel is processing this data. Thus, it is beneficial to use streams. Streams are used to compute convolution on a chunk of data independently of other chunks. This allows possible overlaps to be exploited. Before streams can be used, they must be defined. In addition, for the implementation of this thesis, each stream has a non-pageable (pinned) memory buffer that holds the most recent chunk of Dry data to be sent to the GPU for computation.

5.2.3 – Definition of Audio Devices

Before audio can be sent to an audio output device such as speakers, the device must be configured. The configuration uses the sample rate, block align, and byte rate of the input files. The number of channels is always two because this is convolution on stereo data, the format is PCM since the data was read from WAVE files, and the number of bits per sample is 16, which is a common bit depth.

5.3 – Convolution

The main process for this convolution algorithm is to:

- send the current chunk of the Dry Signal to the GPU
- send the previously received Wet data to the speakers
- use the GPU to compute the Wet data on the current chunk
- copy the Wet data back to the CPU.

The CPU is responsible for manipulating data and keeping track of CPU-GPU coordination (via streams) whereas the GPU is focused solely on the computation of the Wet data. Figure 5.8 illustrates this process.

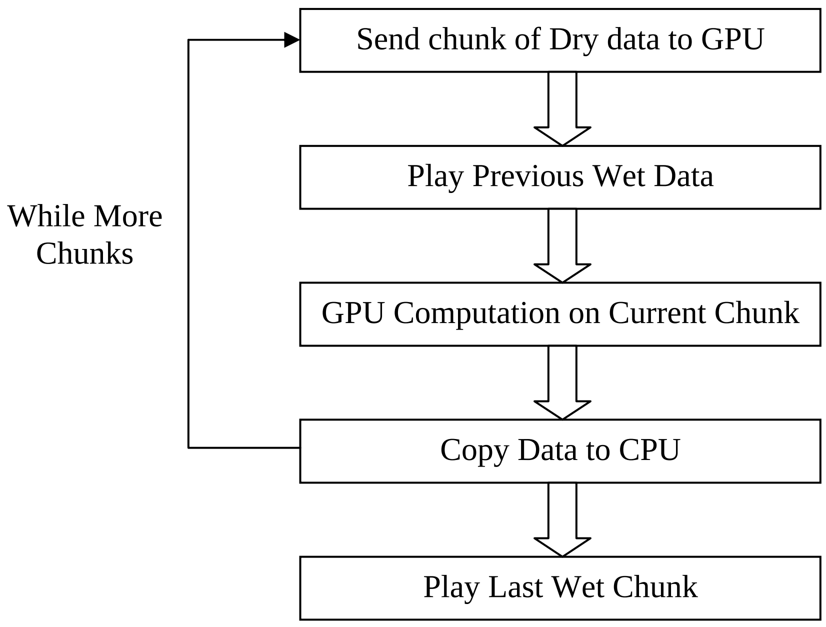

Figure 5.8 – Overall convolution algorithm

5.3.1 – CPU

The CPU is the driving force behind the entire process. It has the responsibility of copying memory to and from the GPU, coordinating between streams, and playing audio.

The convolution step begins with the CPU copying the current chunk of Dry data to the pinned memory buffer of the current stream. If there is insufficient data left to fill the pinned buffer completely, zeroes are appended to the end until the buffer is full. This is equivalent to padding the Dry Signal with trailing zeroes.

Once the GPU has finished executing the previous stream, the next chunk of the Dry data is asynchronously copied to the circular buffer on the GPU. Upon completion of this step, the function returns immediately, allowing the copying of the computed section of the Wet signal of the current stream to the CPU. The copy to the GPU is conducted first because it is possible to asynchronously copy memory to the GPU, but not from the GPU.

Asynchronously copying the data to the GPU allows the copy from the GPU to begin as soon as possible. Since the newer GPUs can support simultaneous memory copies to and from the GPU, ordering the memory copies in this way provides a benefit for the new GPUs without sacrificing efficiency on older GPUs.

Once the GPU has the next section of the Dry data, the kernel is launched. The kernel launch is asynchronous thus control returns to the CPU immediately. While the GPU is computing the next section of the Wet signal, the CPU normalizes the data of the chunk recently copied from the GPU. The CPU scans through the chunk and compares the largest value of the chunk to the largest value of all chunks. Each element is then divided by the largest value, which is guaranteed to keep the data clamped between -1 and 1. This eliminates any distortion and clicking in the audio signal while keeping the volume as consistent as possible with other chunks. This also results in reducing the overall volume if a new maximum value is found.

The audio device expects the audio data to be 16-bit integer samples instead of floating point data. Therefore, the normalized floating point data is converted to the *short* integer data type. After the audio device is finished playing the current stream, the CPU writes the audio data, which is in short integer form, to the audio device.

At least two streams must be present for real-time audio, which is similar to the concept of double buffering. In the case of one stream, there is only one buffer for audio, thus when the stream is finished being played, there is no audio data ready to play. The data received from the GPU must be copied into the buffer, and that can only occur after the stream has finished playing. Waiting for the device to finish playing and then copying the data to the buffer takes a small amount of time, which causes a gap in the audio that is heard as a clicking sound. Following this step, the stream number is incremented and set to zero if it is greater than the number of streams. When the stream is set to zero, the buffer number is incremented. If the buffer number exceeds the number of buffers for each stream, it is also set to zero. The loop continues until all Dry data and its trailing zeroes have been sent to the GPU for computation.

5.3.2 – GPU

The main task of the GPU is to perform the necessary calculations for convolution. Figure 5.9 illustrates the GPU process for the convolution algorithm of this thesis.

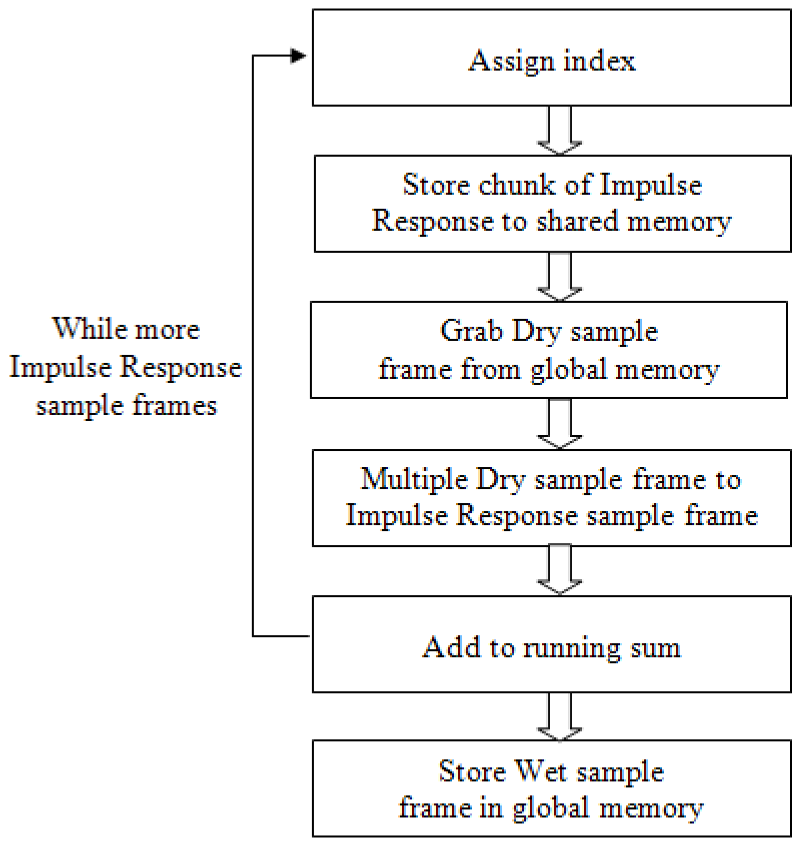
Assign index
Store chunk of Impulse Response to shared memory
Grab Dry sample frame from global memory
Multiple Dry sample frame to Impulse Response sample frame
Add to running sum
Store Wet sample frame in global memory
While more Impulse Response sample frames

Figure 5.9 – GPU convolution steps

The GPU kernel needs four parameters – the address of the Wet data array, the address of the Dry data array, the index from which the kernel is starting to read the Dry data, and the starting index in which the calculated results are to be saved. The addresses of the arrays are assigned to variables on the CPU when the memory for the arrays is allocated on the GPUs. The addresses are pointers to global memory on the GPU.

The first step is to assign each individual thread the data used as input and the location where it stores the result. This is accomplished via the *blockIdx*, *blockDim*, and *threadIdx* variables. The data are one-dimensional and the calculation to guarantee every sample frame is assigned to only one thread is as shown in (5).

$$index += blockIdx.x * blockDim.x + threadIdx.x \tag{5}$$

This assignment also makes efficient use of memory by allowing the GPU hardware to coalesce memory accesses since each thread has an index adjacent to its neighboring threads. The same assignment is used to calculate the index used to store the Wet data.

After completing the above task, a *while* loop is used to guarantee that all sample frames in the chunk are processed by not allowing the kernel to terminate until all sample frames have been computed. This is dependent on the number of threads per block and the number of blocks. If there are not sufficient threads in the kernel to process all of the sample frames, a stride value is needed. This stride value is the total number of threads in the grid and is calculated by (6).

$$stride = blockDim * gridDim \tag{6}$$

During each iteration of the while loop, the stride is added to the index variable of the thread. Note that if the kernel is launched with a sufficient number of threads per grid to process all the sample frames in the chunk, the condition code of the while loop and the stride instructions can be omitted. This can save computation time and reduces the size of the kernel. The minimum number of blocks needed to guarantee all sample frames in the chunk are processed as shown in (7).

$$blocks = \left\lfloor \frac{chunk\ size}{threads\ per\ block} \right\rfloor \tag{7}$$

Every sample frame of the Wet data depends on every sample of the impulse response. Each thread in the block is using the same element of the impulse response at the same time, on different sample frames of the Dry data. Using shared memory can improve the performance of this access pattern over using global memory.

Each thread of a block grabs a sample frame of the impulse response from texture memory and stores the value in a shared memory array. Because all data accesses are neighboring, all the accesses are coalesced. Therefore the number of sample frames in shared memory of the impulse response is equal to the number of threads per block. The block waits for every thread to access its sample frame before continuing execution. Each sample frame in the shared memory is then accessed in a loop. The current Dry sample frame is read from global memory using the current index. One is subtracted from the current index for the next calculation, resetting to the other end of the circular buffer if the value is -1. The Dry sample frame is multiplied with the impulse response sample frame and added to a running sum.

When all the samples from shared memory have been processed, the loop starts again, this time accessing the next group of sample frames from the impulse response. After all values have been calculated and all samples of the impulse response processed, the resulting sum is stored in global memory. Finally, the stride mentioned previously is added to both the Dry index and the Wet index.

The real time kernel is intended for applications where the entire Dry signal is not known in advance, such as streaming across the Internet. In such a case, some optimizations cannot be performed. In order to view many optimizations, in addition to the real time kernel, a separate kernel that computes the entire signal at once was developed as part of the research for this thesis. The algorithm is similar to that of the real time kernel, except that the Dry signal is explicitly padded with zeroes, is bound to texture memory, and a single kernel computes the entire Wet signal.

Figures 5.10 and 5.11 illustrate the main differences between the offline and real time kernels by showing a simplified version of the optimized kernels. The main difference is the circular buffer check for the real time kernel. Because the offline kernel has all the Dry data, there is no need for a check. The other main difference is the use of texture memory. Because the offline kernel has all the Dry data, it can bind the Dry data as a texture. Since the real time kernel does not have the full Dry signal at any one time and because more recent chunks of Dry data overwrite previous chunks, it cannot utilize texture memory.

```
while(THREADS_PER_BLOCK * loopNum < impulseResponseSize) {
  impulseResponseCache[threadIdx.x] =
    tex1Dfetch(impulseResponseTexture,
    (THREADS_PER_BLOCK * loopNum) + threadIdx.x);

  __syncthreads();

  for(j = 0; j < THREADS_PER_BLOCK; j++) {
    drySampleFrame = tex1Dfetch(dryTexture, dryIndex);
    dryIndex--;
    sum.x += (impulseResponseCache[j].x * drySampleFrame.x);
    sum.y += (impulseResponseCache[j].y * drySampleFrame.y);
  }

  loopNum++;
  __syncthreads();
}
```

Figure 5.10 – Simplified offline kernel

```
while(THREADS_PER_BLOCK * loopNum < impulseResponseSize) {
  impulseResponseCache[threadIdx.x] =
    tex1Dfetch(impulseResponseTexture,
    (THREADS_PER_BLOCK * loopNum) + threadIdx.x);

  __syncthreads();

  for(j = 0; j < THREADS_PER_BLOCK; j++) {
    drySampleFrame = dry[dryIndex];
    dryIndex--;

    if(dryIndex == -1)
      dryIndex = bufferSize;

    sum.x += (impulseResponseCache[j].x * drySampleFrame.x);
    sum.y += (impulseResponseCache[j].y * drySampleFrame.y);
  }

  loopNum++;
  __syncthreads();
}
```

Figure 5.11 – Simplified real time kernel

Chapter 6 – Results

For proper comparison of running a convolution algorithm on the CPU against convolution running on a GPU, the best kernel for offline and the best kernel for real time were selected. The selection process involved testing a 16-second Dry signal with a 2.5-second impulse response, with a sampling rate of 44.1 KHz. The time, in milliseconds, of the kernels, was recorded using the CUDA API's *cudaEventRecord()* function. In the case of the real time kernel, the kernel time is the sum of each kernel launch. The GPU used was an EVGA Nvidia GTX260 Superclocked version with 216 cores and a compute capability of 1.3. The CPU was an Intel E5320 1.86 GHz.

6.1 – Improving GPU Kernels with Optimizations

The optimizations selected are to use texture memory, shared memory, loop, prefetching, and removal of the outer loop by making sure the kernel launches with a sufficient number of blocks to complete the computation. The "simple" kernel is the kernel without any optimizations.

Figure 6.1 shows the results for the offline kernel in which it is apparent using shared memory and texture memory achieved better performance. This is because there is no hardware cache on the GTX260 as it is of compute capability 1.3. Prefetching the impulse response did not affect the performance much, but still saw a small speedup. A loop unroll of eight was found empirically to provide the best performance, and yielded an approximate speedup of 1.8 times. Finally, removing the outer loop slightly hindered the performance. This has more to do with launching more blocks rather than reusing already executing ones than removing the loop logic.

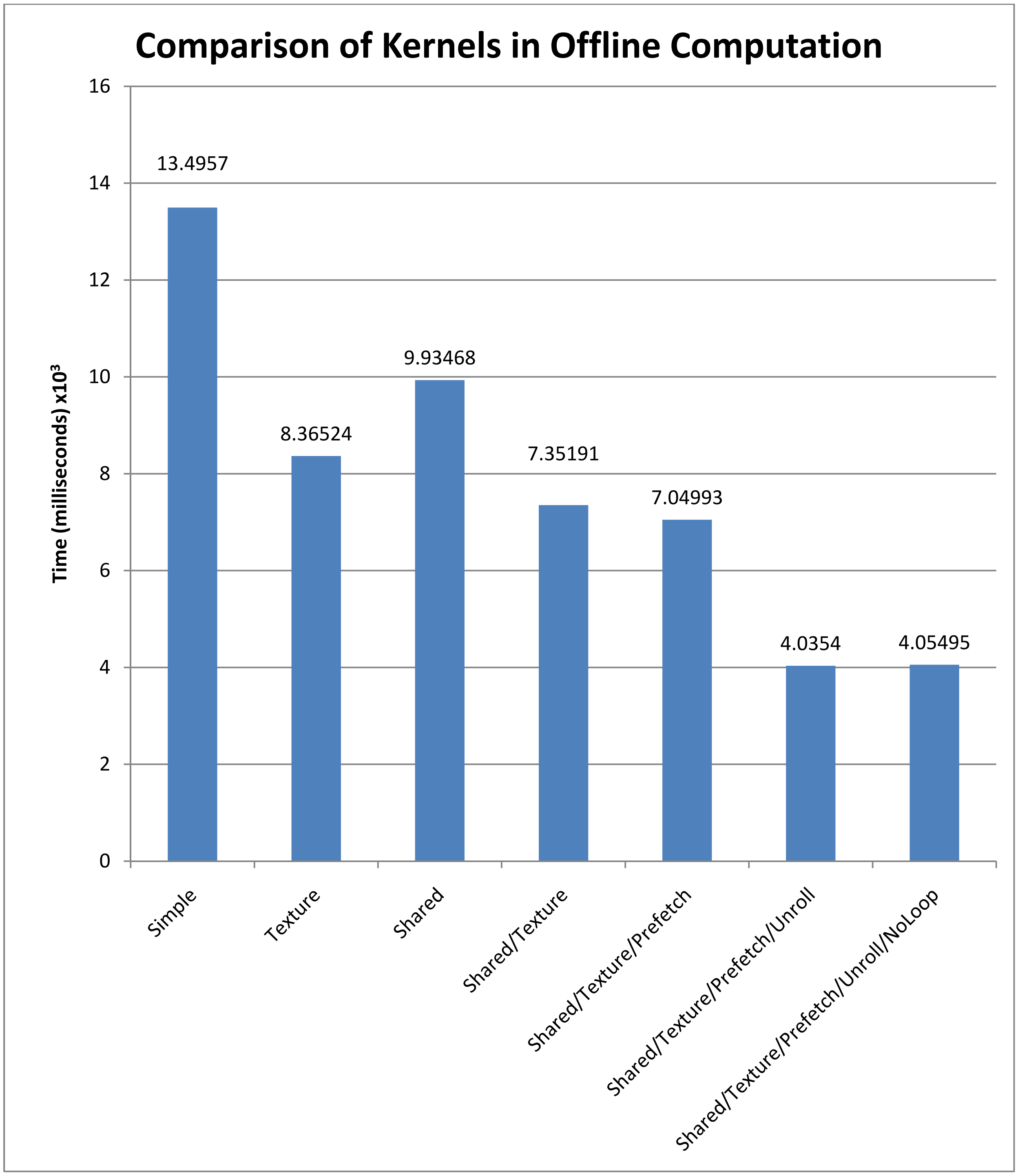

Figure 6.1 – Comparison of execution time of the offline kernel using different optimizations

Figure 6.2 illustrates the results for the real time kernel using a chunk size for the Dry data of 22,528 sample frames. This sample frame size was found experimentally to produce the best performance. Similar to the offline kernel, shared memory and texture memory resulted in improved performance. Shared memory improved the performance approximately the same amount as in the offline kernel. However texture memory did not have as much effect on performance, because in the real time kernel, only the impulse response can be bound as a texture as the CPU is constantly writing data to the Dry data circular buffer. While removing the outer loop only saved 6 milliseconds, it also reduced the amount of registers used by four. This can be important for developers who need to reduce the amount of registers their kernels use to keep the occupancy high.

As contrasted to the offline kernel, prefetching the impulse response data did not improve performance, because each of the real time kernels are running for less time than the offline kernels and the overhead of prefetching overcomes the benefit on the shorter kernels. Similar to the drop in performance with prefetching, unrolling the loop decreased performance. A loop unroll of 4 was found to yield the best loop-unroll performance, but still reduced overall performance while using five more registers. The reduction in performance and use of more registers is caused by the circular buffer checks to make sure wrap around is applied if necessary.

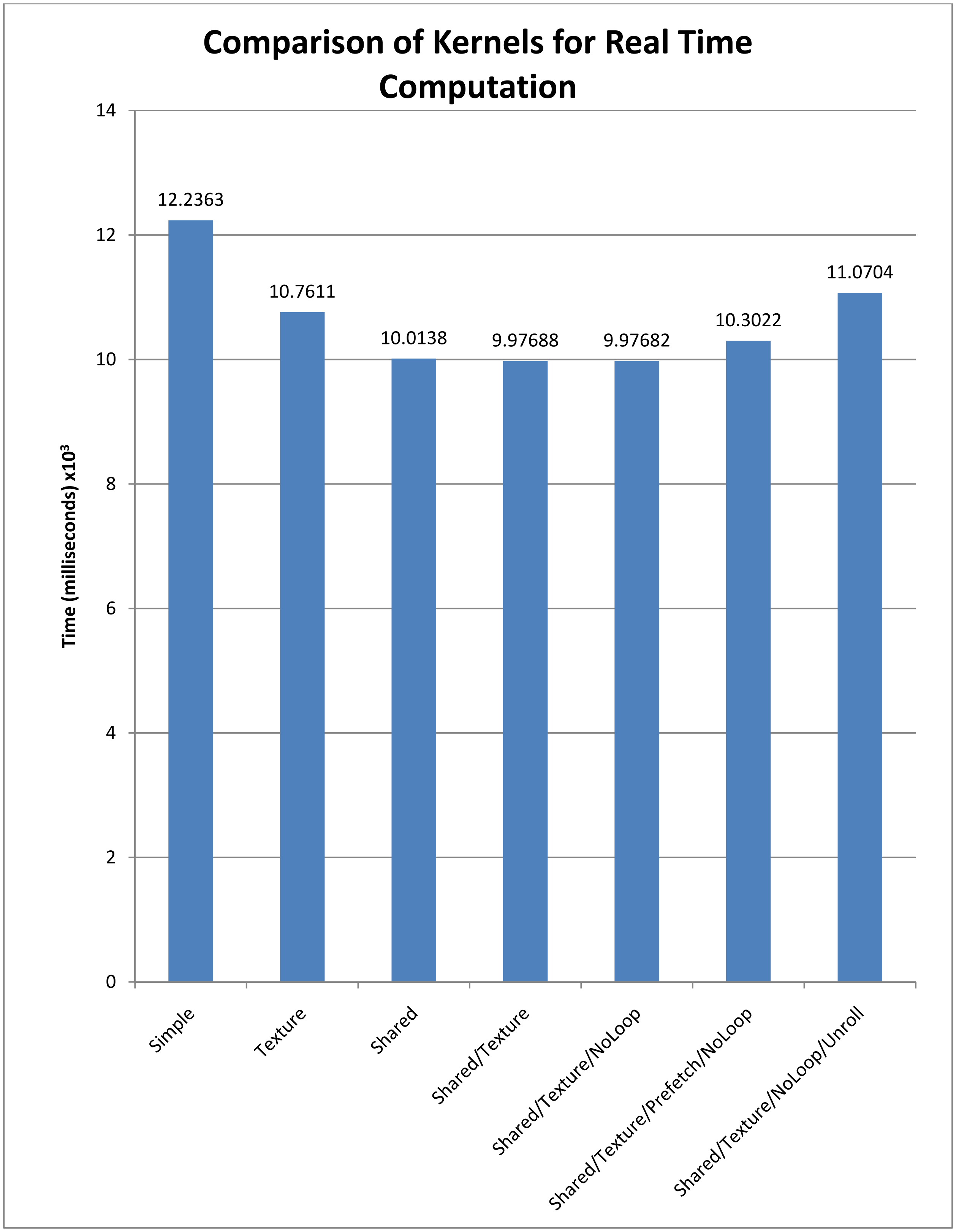

Figure 6.2 – Comparison of execution time of the real time kernel using different optimizations

6.2 – Improving Convolution

Among the offline kernels, the kernel with all optimizations without the removal of the outer loop performed best. Of the real time kernels, the kernel utilizing shared memory, texture memory, and the removal of the outer loop performed best. For this reason, these two kernels will be compared in time domain convolution and frequency domain multiplication on the CPU. The CPU functions are built-in functions for MATLAB. MATLAB's conv() function performs time domain convolution and cconv() performs frequency domain multiplication. The tests were performed on an Intel E5320 1.86 GHz processor. All tests were run on two different impulse responses: one created digitally by generating white noise and the other recorded from an unfinished chapel and obtained from the Real Rooms online repository [11]. Both were edited to be exactly 2.5 seconds long. Figure 6.3 illustrates the waveform for the chapel impulse response, and Figure 6.4 illustrates the waveform for the white noise impulse response.

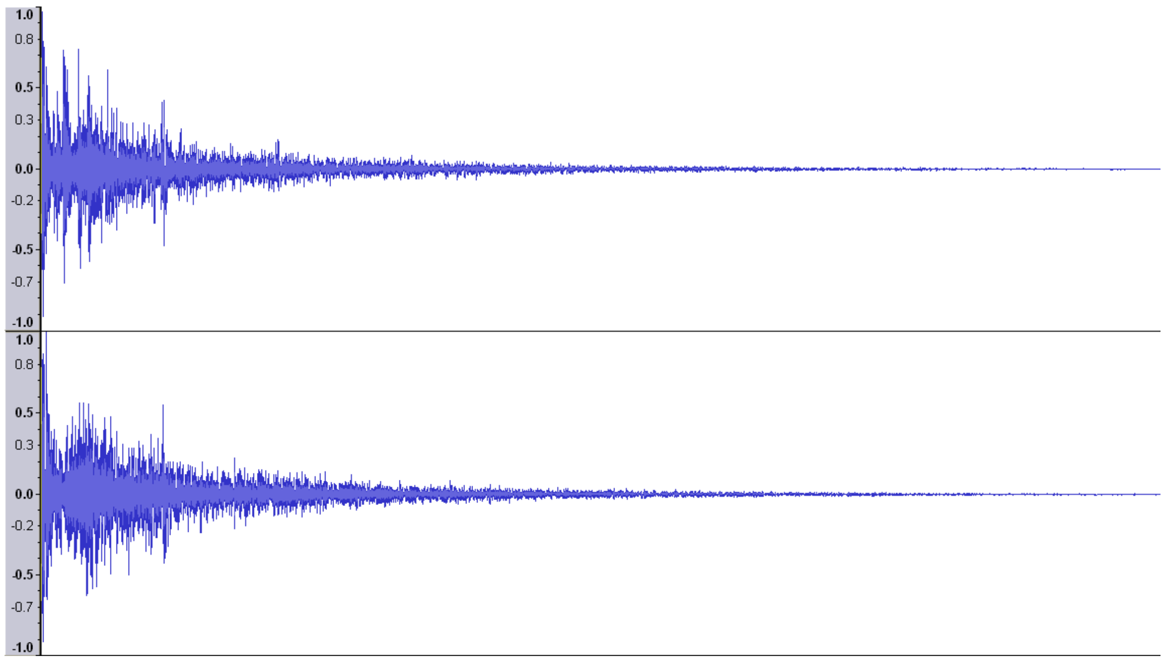

Figure 6.3 – The waveform of the chapel impulse response

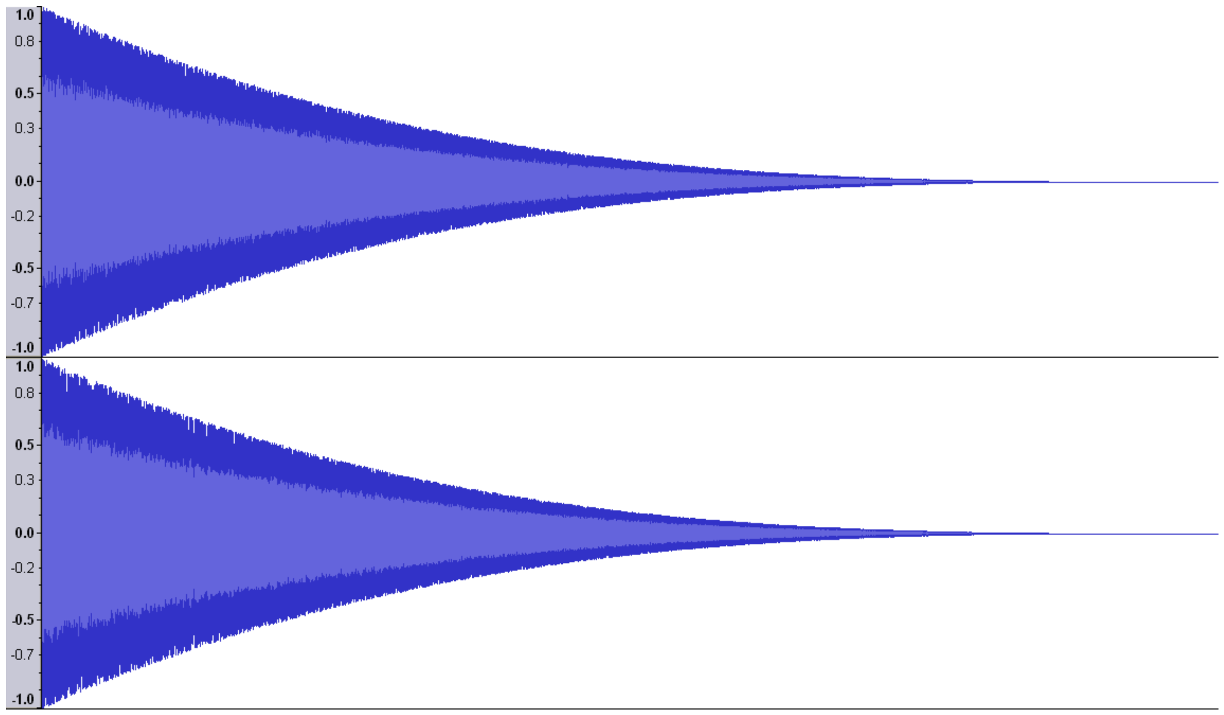

Figure 6.4 – The waveform of the white noise impulse response

Figure 6.5 illustrates using a logarithmic scale the comparison of MATLAB's conv() function to the best-performing GPU kernels for the impulse response recorded in a chapel. Both GPU kernels performed better than the CPU, with speedups ranging from 106 times to 125 times faster. These results show that CPU time domain convolution has a very steep linear nature. The GPU kernels, however, have a slight parabolic curve.

Figure 6.6 shows the comparison of MATLAB's cconv() function to the best-performing GPU kernels. While the cconv() function performs better in most cases, the offline kernel performs close to it or performs better for impulse response lengths less than three seconds. As Figure 6.5 illustrates, the processing time for the cconv() function does not follow a specific curve. This may be due to the frequencies present in the impulse response.

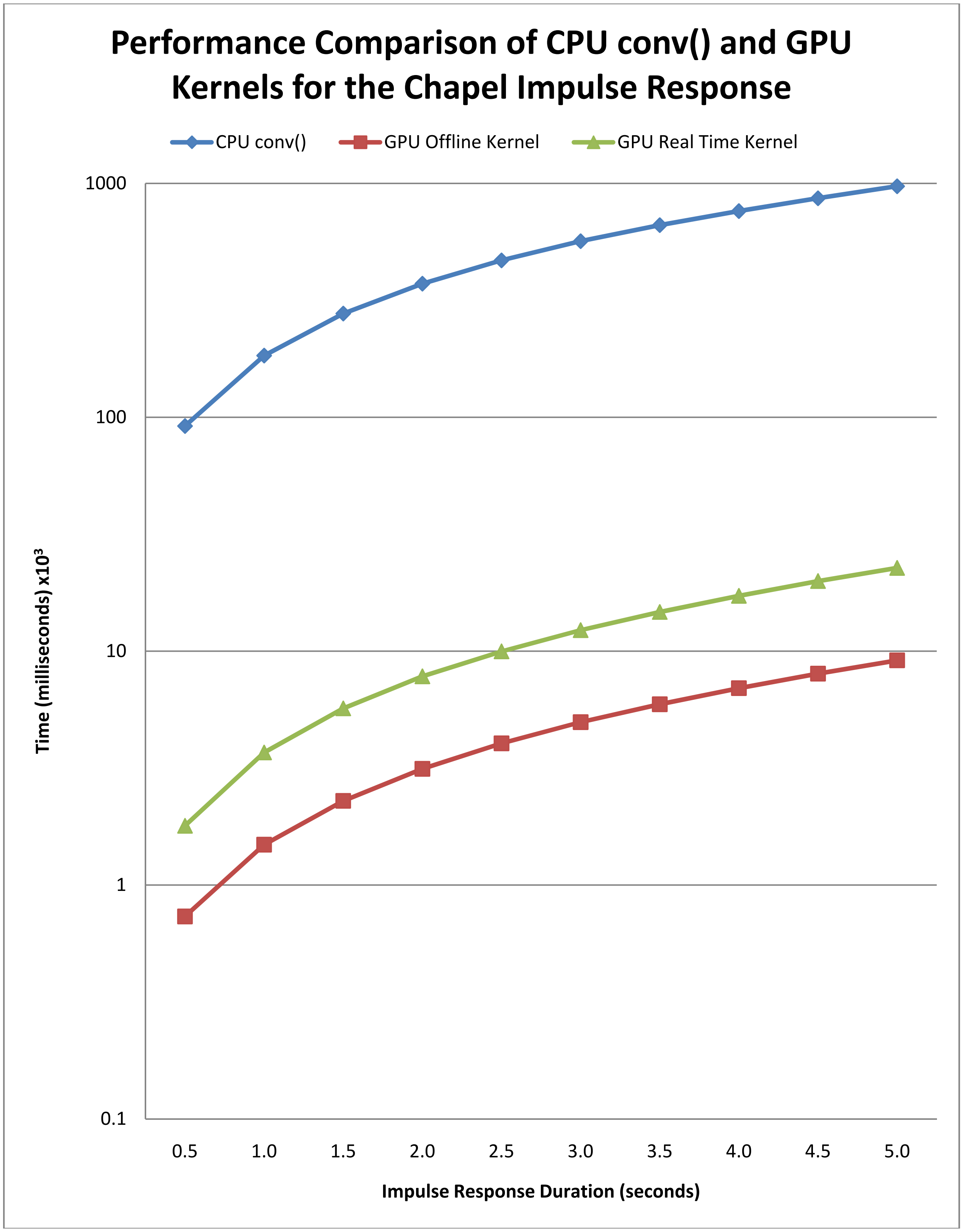

Figure 6.5 – Comparison of the CPU function conv() and the GPU kernels for the chapel impulse response

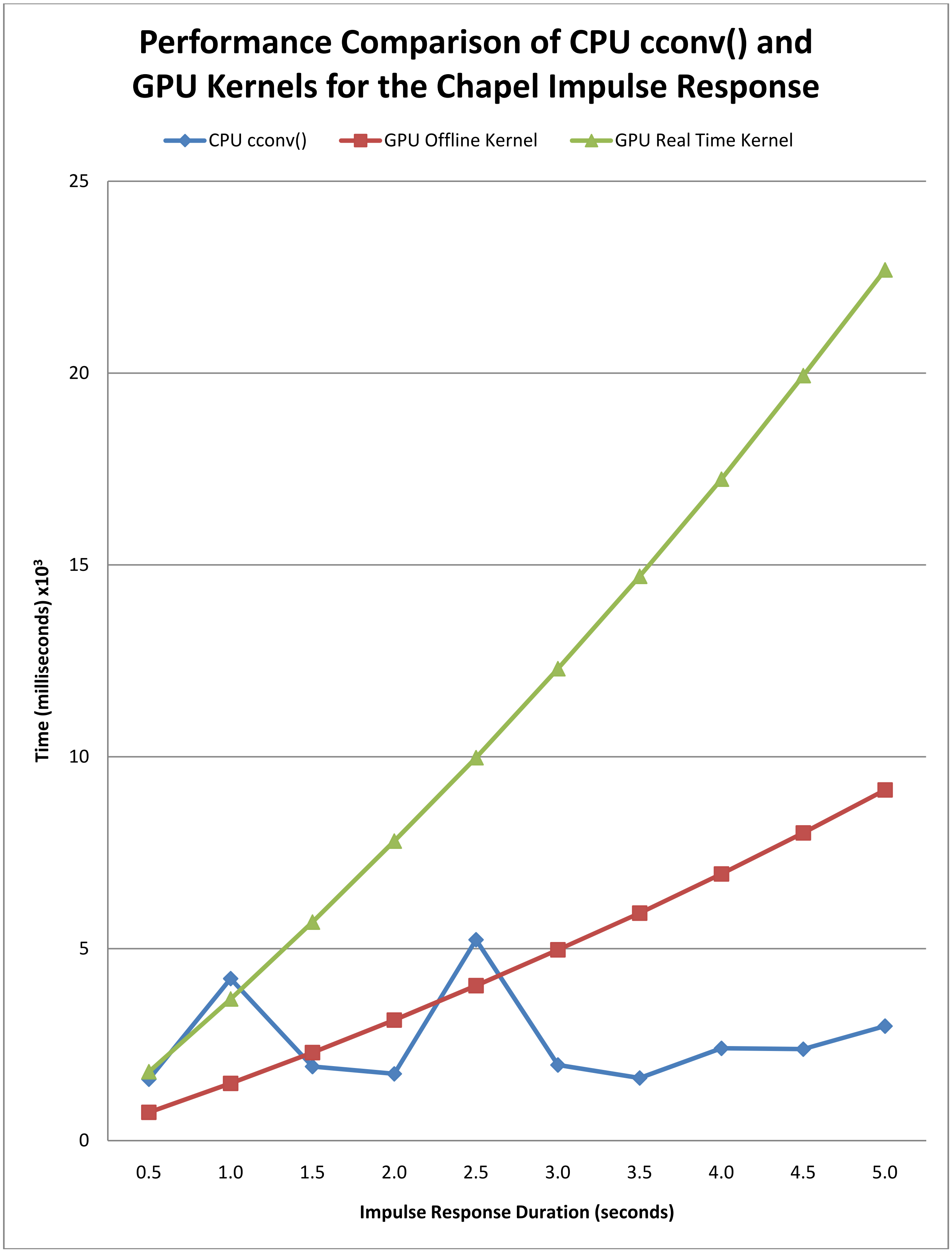

Figure 6.6 – Comparison of the CPU function cconv() and the GPU kernels

Figure 6.7 illustrates using a logarithmic scale the comparison of the conv() function for the white noise impulse response. The execution time for both GPU kernels and the conv() function are almost identical to those for the chapel impulse response. This is opposite of the cconv() function, in which the processing time depends upon the signal. It appears that unlike time domain convolution, where processing time is only dependent on the length, frequency domain multiplication appears to be dependent on the frequencies of the signals.

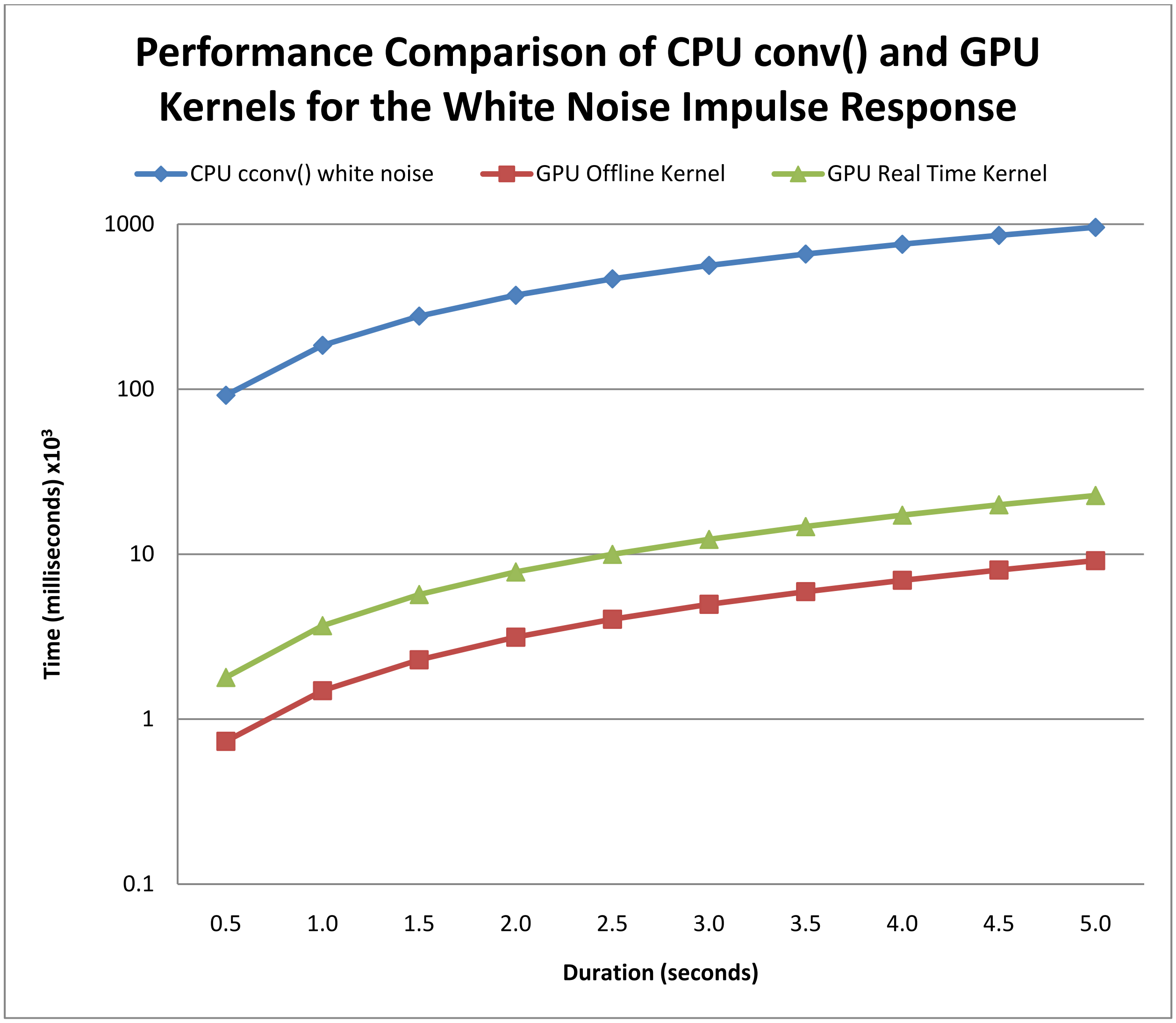

Figure 6.7 - Comparison of the CPU function conv() and the GPU kernels for the white noise impulse response

Figure 6.8 illustrates the frequency spectrum for the chapel impulse response. As the figure illustrates, the spectrum is not flat. For this spectrum, as the frequency goes up, the amplitude of the frequency goes down. This means that there are less high frequency components than low frequency components for the chapel impulse response. This could be the result of the air dissipating the high frequency energy faster than the low frequency energy, the properties of the microphone used to record the impulse response, or the gunshot used to create the sound file [11].

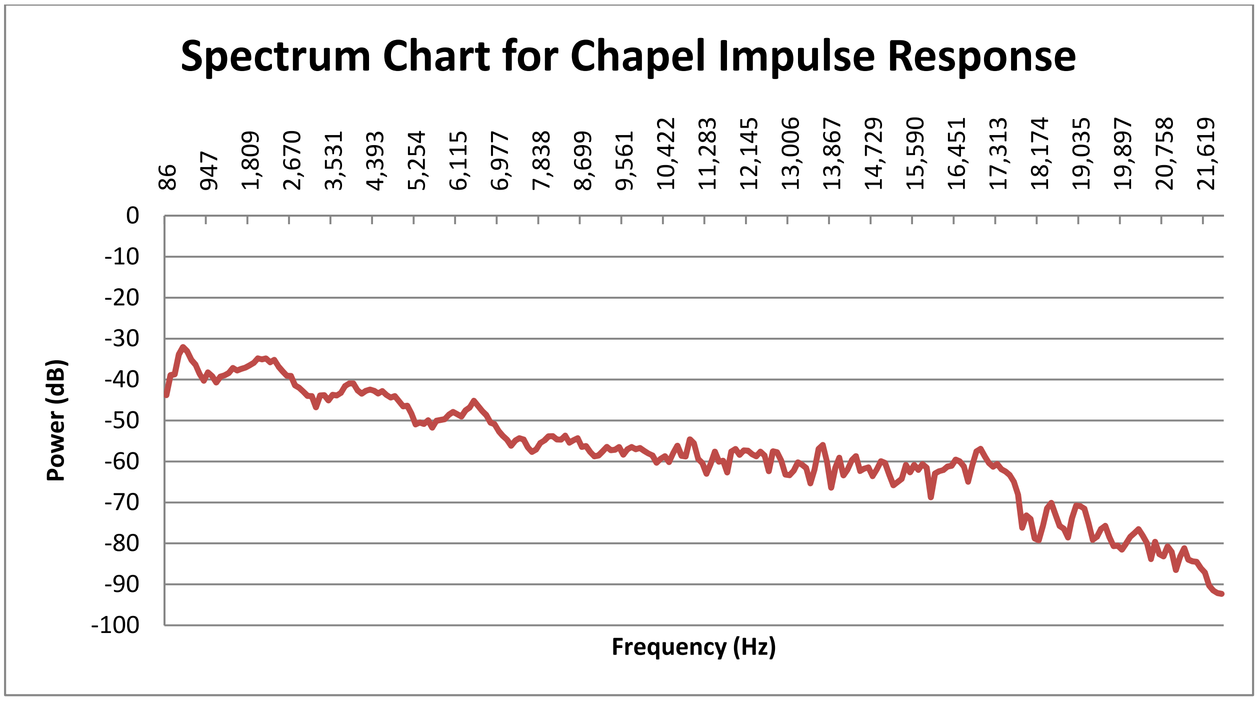

Figure 6.8 – Spectrum for the chapel impulse response

Figure 6.9 illustrates the frequency spectrum for the white noise impulse response. As the figure illustrates, the spectrum is flat, indicating that the amplitude of all frequencies in the impulse response is very similar. White noise was chosen as a baseline for comparing to a recorded impulse response for its properties of having a flat spectrum.

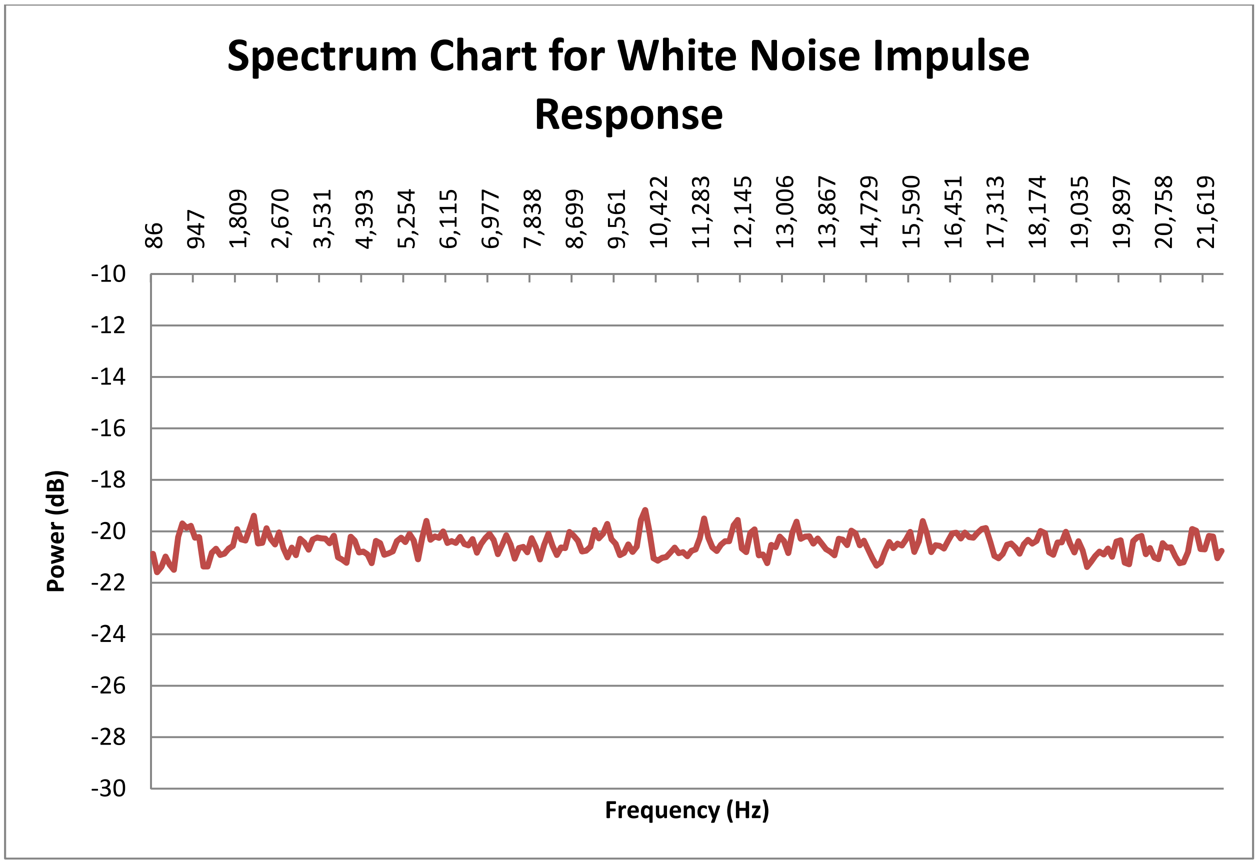

Figure 6.9 – Spectrum for the white noise impulse response

Figure 6.10 illustrates the performance comparison of the CPU cconv() function against the GPU kernels for the white noise impulse response. The results from the cconv() function are much more stable for the white noise impulse response than the chapel impulse response. The GPU kernels perform similarly to the cconv() function for short impulse responses. However, in all cases but the shortest impulse response, the CPU cconv() function performed better.

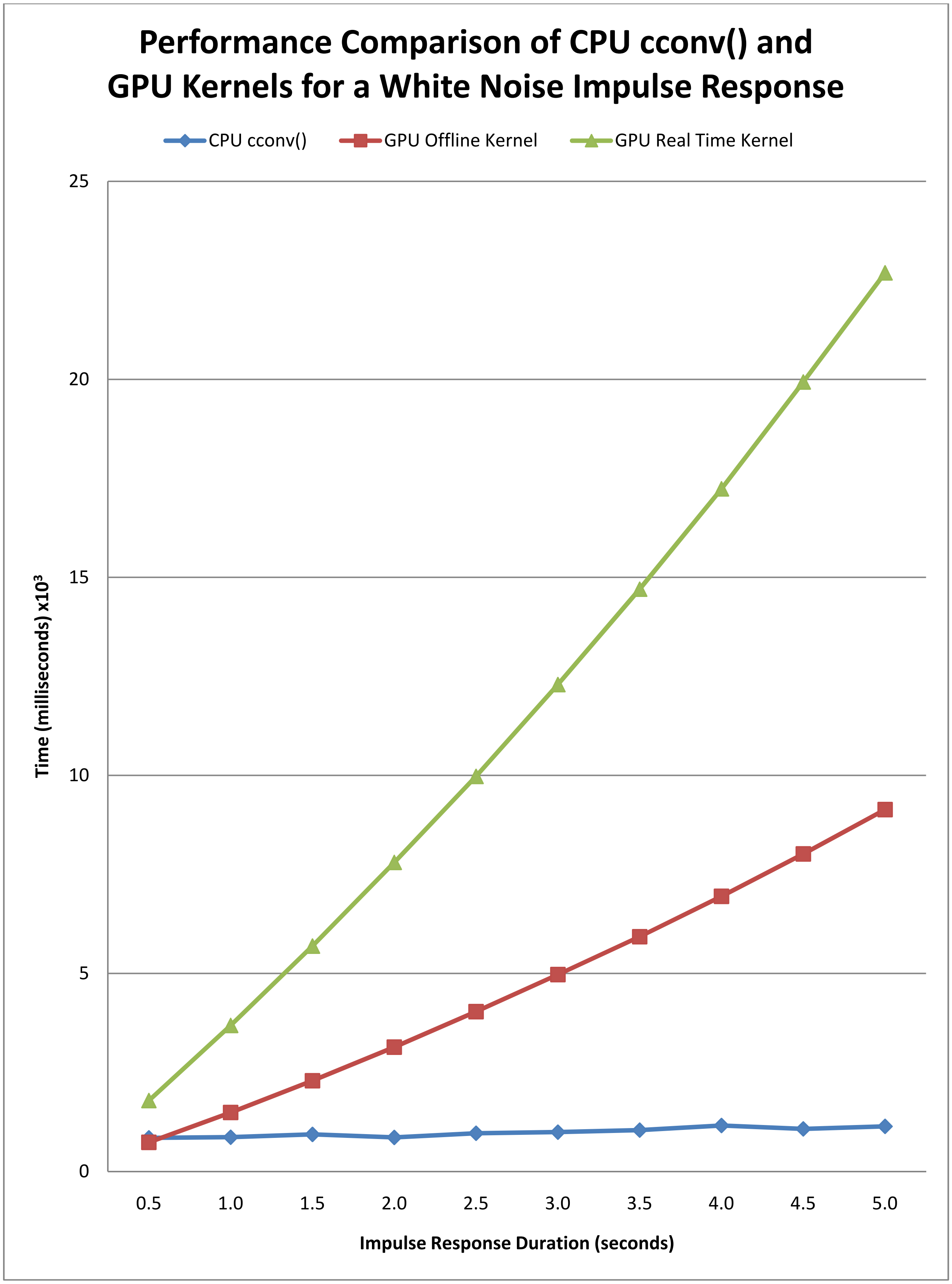

Figure 6.10 – Performance comparison between CPU cconv() and GPU kernels

In addition to testing various lengths of impulse responses, a variety of Dry lengths and a constant length impulse response were tested to see how performance scales with increasing Dry length. Dry lengths tested were 30-second increments from a length of 30 seconds to five minutes. The Dry signals were created via Audacity, a free, open source application for editing sound files [1]. The left channel was a Sine wave sweep from 20 Hz to 20 KHz and the right channel was white noise. This arrangement was chosen to give the broadest test possible by including all frequencies and by having the samples both close to each other (the sine wave sweep) and far apart (white noise). The impulse response used was the 2.5-second chapel impulse response.

Figure 6.11 illustrates using a logarithmic scale the performance comparison between different Dry lengths on the two GPU kernels and the CPU conv() function. The conv() function takes significantly longer to execute in all cases. The growth is similar to the growth for increasing impulse response lengths and is steep. Again, the offline kernel performs better than the real time kernel in all cases.

Figure 6.12 illustrates the performance comparison between different Dry lengths on the two GPU kernels and the CPU cconv() function. Similarly to the impulse response lengths test, the cconv() function performed better than the GPU in all cases and the offline kernel outperformed the real time kernel.

Upon completion of testing with the GTX260 GPU, a new GPU became available. The results from initial simple test cases on the new GPU, an Nvidia Tesla C2050 with compute capability 2.0, are illustrated in figure 6.13. The performance comparison between different lengths of impulse responses for the real time kernel running on the C2050 GPU and the GTX260 GPU is presented in this figure.

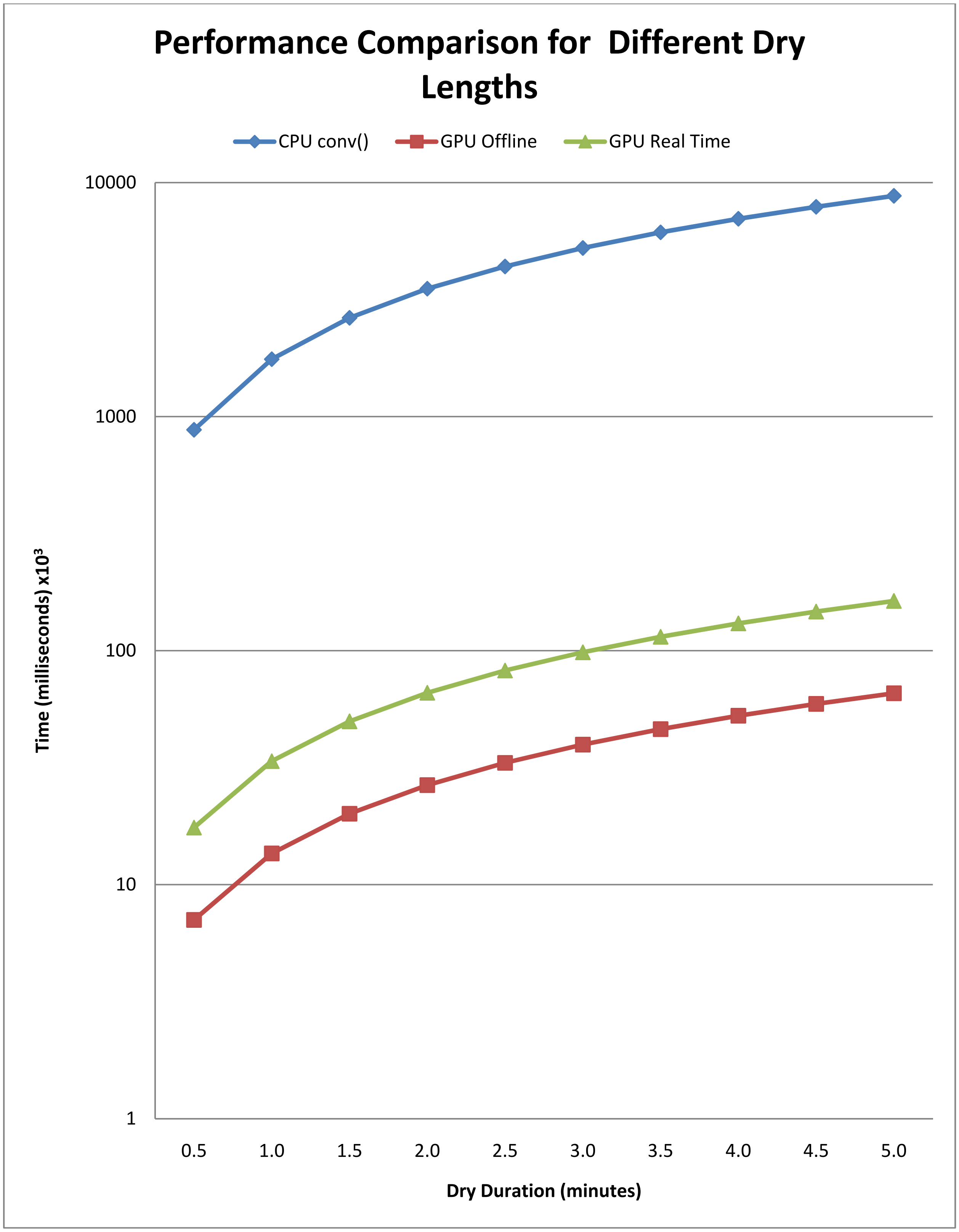

Figure 6.11 – Performance comparison between GPU kernels and CPU conv() function for various Dry lengths

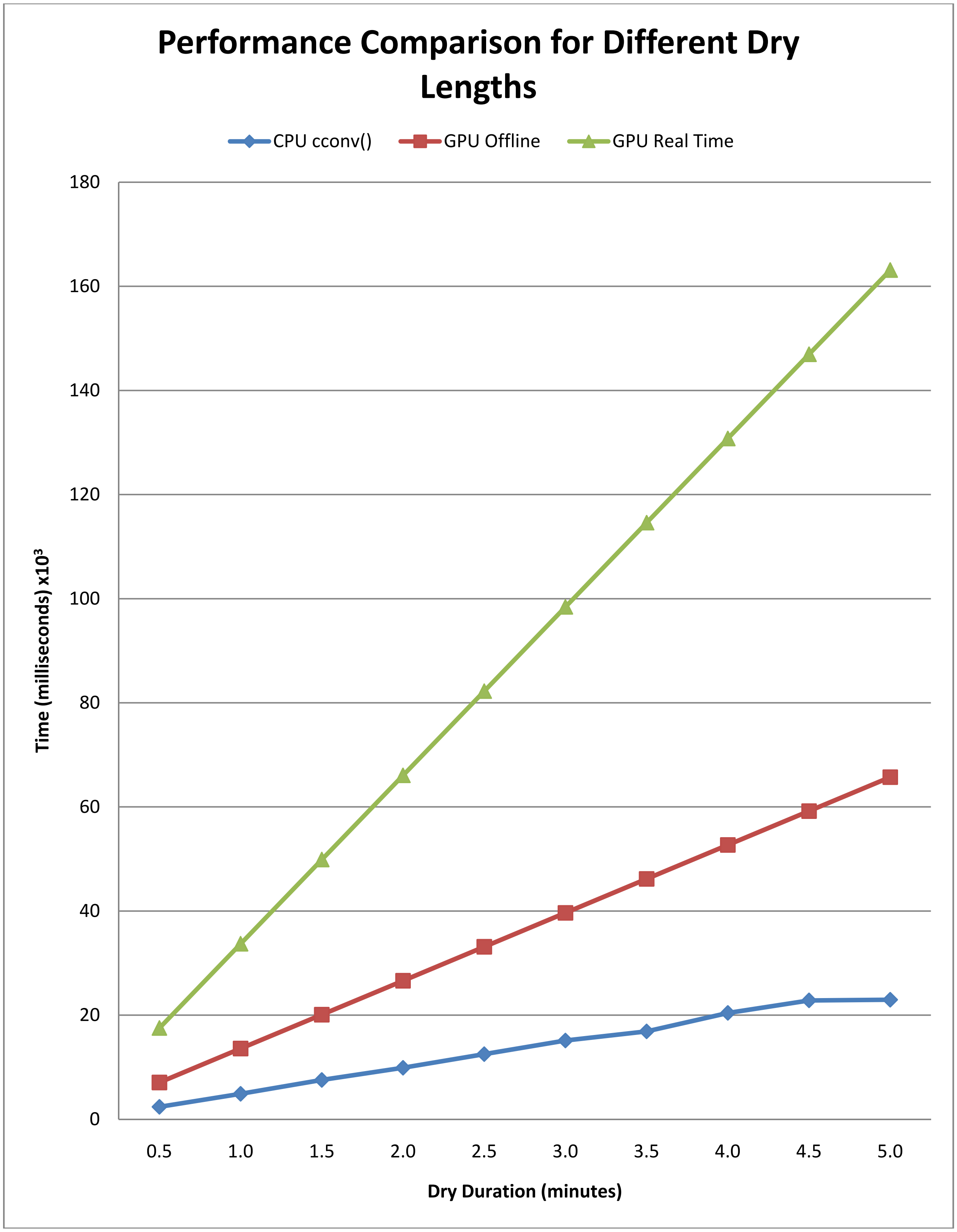

Figure 6.12 - Performance comparison between GPU kernels and CPU cconv() function for various Dry lengths

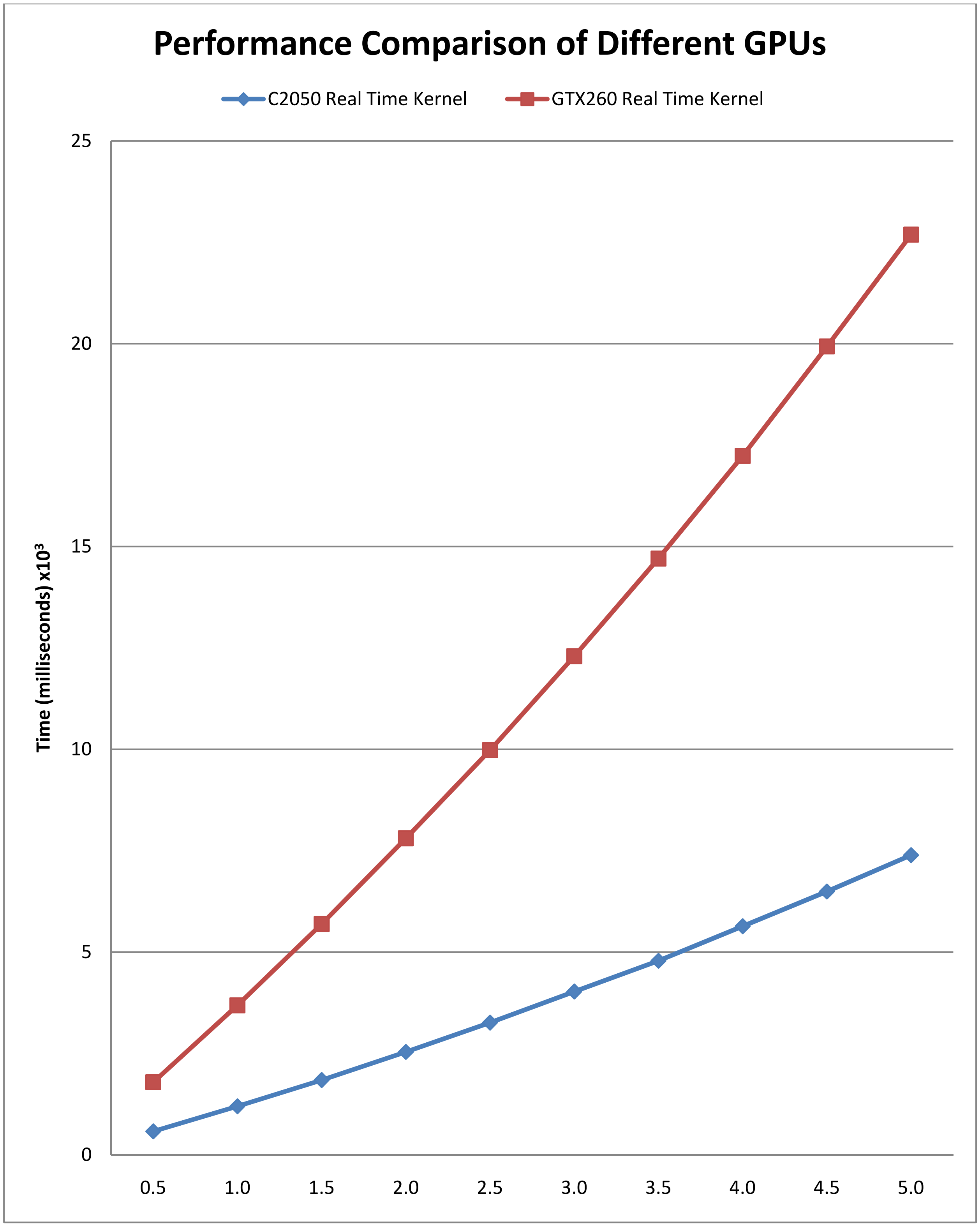

Figure 6.13 – Performance comparison of the real time kernel for a GTX260 and a C2050 GPU

Chapter 7 – Conclusions and Future Work

7.1 – Conclusions

In this research, real time computation of time domain convolution on graphics cards and GPU optimizations were studied. The research investigated the optimization strategies of utilizing shared memory, texture memory, prefetching, loop unrolling, and removal of the outer loop that checks that all Wet data has been calculated.

The results indicate that the offline GPU kernel performs better than the real time kernel by executing approximately 2.4 times faster in all cases. Compared to the CPU conv() function for the impulse response length test, the offline kernel which has 216 cores, performed up to 125 times faster. For the Dry length test, the offline kernel performed up to 133 times faster.

However, it is not always the case that the entire Dry signal is known. In this case, the real time kernel provides sufficient performance over the CPU to warrant its use over the CPU for time domain convolution. The real time kernel performed convolution up to 51 times faster than the CPU conv() function on the test files for the impulse response length test. For the Dry length test, the real time kernel performed up to 53 times faster than the CPU conv() function.

For impulse response lengths of approximately one second or less, the offline GPU kernel matches or performs better than frequency domain multiplication on the CPU. The real time kernel still performs well enough to process approximately 4.6 seconds in real time.

For impulse response lengths greater than that, the kernels will not finish fast enough to meet the deadline and will cause gaps in the audio.

For optimizations, the results are clear. For problems such as time domain convolution, considerable effort should be made on using texture memory and shared memory whenever possible. Texture memory allowed the offline kernel a 1.61 times speed increase and a 1.22 times speed increase for the real time kernel. Shared memory sped the offline kernel up by 1.36 times and the real time kernel by 1.22 times. Using both optimizations together gave an overall speed increase of 1.84 times for the offline kernel and 1.23 times for the real time kernel. Both optimizations provided considerable speedup over accesses to global memory.

For kernels that have inner loops that have no conditionals, manually unrolling the loop is strongly encouraged. For the offline kernel, manually unrolling the inner loop increased performance by 1.75 times that of the kernel already using the shared memory, texture memory, and prefetching optimizations. Compared to the offline kernel with no optimizations, it is running 3.34 times faster.

Prefetching is encouraged only if there is time left in the development cycle, as it did not guarantee a performance boost. When it did boost performance, in the offline kernel, it was only sped up by 1.04 times. Using prefetching in the real time kernel hindered performance by 0.97 times. Similarly, loop unrolling of loops with condition statements is also suggested to only test if time permits. As well as extra register usage, it is not guaranteed to increase performance. Unrolling the inner loop reduced the performance of the real time kernel using the shared memory and texture memory optimizations by 0.90 times. In contrast to the offline kernel, unrolling the loop did not increase the ratio of instructions calculating data and those checking conditional statements. Unrolling this loop also added more conditional checks, which reduced performance.

When writing kernels, it is a good idea to support any number of threads and blocks with an outer loop. After testing many configurations for best performance, if it is possible to determine the number of blocks needed at runtime for the data, it is encouraged to test without the outer loop. Even if there is no increase in performance, it may reduce the number of registers used by the kernel. This can allow more warps to execute simultaneously and allow a higher occupancy. Even if no performance gain occurs, it may still be possible on different GPUs, especially future GPUs, to improve performance since a lower register usage may mean another stream may start execution while the current stream is executing.

Initial testing for the real time kernel on the Nvidia C2050 may indicate that new GPUs with compute capability 2.0 can perform much better than older compute capabilities without any changes to the code. The only change was a compiler option for compute capability 2.0 instead of compute capability 1.3. The C2050 GPU executed the real time kernel approximately 3.1 times faster than the GTX260 GPU. In addition, the C2050 GPU was able to process an impulse response of 14.1 seconds in real time. The addition of a hardware cache and ability to execute multiple streams simultaneously led to this improvement.

7.2 – Future Recommendations

With the exception of the single test case of the C2050 GPU, these kernels were compiled for and executed on GPUs of compute capability 1.3. Compute capability 2.0 has numerous improvements, including a hardware cache for global memory and a choice of a larger shared memory. Every access of a Dry data element is always one element less than before. Because a thread of one index less will have used that data in the last iteration, it is highly likely the value will be in the hardware cache. It would be worth investigating how hardware caching improves performance in more than a single, simple test case, especially of the real time kernel where the Dry data cannot be bound to a texture. In addition, tweaking the code for the new architecture, such as testing a larger shared memory or prefetch cache, would be beneficial to determine how convolution scales to multiple GPUs.

The CPU that the conv() was tested on was perhaps slow by today's standards. Testing on a variety of different CPU types and speeds will provide better performance increase numbers. It would also be beneficial to include special chips designed to do digital signal processing as part of the comparisons.

References

[1] Sound Editor Application. http://audacity.sourceforge.net/

[2] Cardinal, P., Dumouchel, P., Boulianne, G., and Comeau, M. 2008. *GPU Accelerated Acoustic Likelihood Computations.* Paper presented at the Centre de Recherche Informatique de Montréal, Montréal, Canada.

[3] Fabritius, F. 2009. *Audio Processing Algorithms on the GPU.* Master's Thesis. Technical University of Denmark.

[4] Kerr, A. and Campbell, M. R. 2008. *GPU VSIPL: High-Performance VSIPL Implementation for GPUs.* Paper presented at the Georgia Institute of Technology, Georgia Tech Research Institute.

[5] Kirk, D. B. and Hwu, W. W. 2010. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann Publishers, Burlington, MA.

[6] Loy, G. 2007. *Musimathics: The Mathematical Foundations of Music Volume 2.* The MIT Press, Cambridge, MA.

[7] Nexiwave. 2010. *Nexiwave.com and UbiCast Partner to Offer Next-Generation Deep Audio Search.* http://nexiwave.com/

[8] Nieuwpoort, R. and Romein, J. 2008. *Using Many-Core Hardware to Correlate Radio Astronomy Signals.* Paper presented at the Netherlands Institute for Radio Astronomy, Dwingeloo, The Netherlands.

[9] NVIDIA Corporation. *CUDA C Best Practices Guide Version 3.2.* 20 August 2010.

[10] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide Version 3.2.* 9 November 2010.

[11] Real Rooms, 2011. http://www.impulseresponse.org/real_rooms.htm

[12] Sanders, J. and Kandrot, E. 2010. *CUDA By Example.* Boston, MA: Addison-Wesley

[13] Stanford Exploration Project. 2004. *Time domain versus frequency domain.* http://sepwww.stanford.edu/sep/prof/gee/mda/paper_html/node2.html

[14] Wilson, Scott. 2003. *Microsoft WAVE Soundfile Format.* https://ccrma.stanford.edu/courses/422/projects/WaveFormat/

[15] Zeller, C. 2008. *NVIDIA Tutorial CUDA.* http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf

Appendix A – User Manual for Executables

Please see the enclosed CD for executables and application code developed for this thesis. There are two executables, OfflineConvolution.exe and RealTimeConvolution.exe. Both are compiled for GPUs of compute capability 1.3 and higher. Both are executable via the command line using command line arguments. The first argument is the file location of the Dry data. The second argument is the file location of the impulse response. OfflineConvolution.exe requires a third argument, the location to save the output WAVE file. The input files must: 1) be in WAVE files, 2) be in stereo format (contains two audio channels), 3) have equivalent sample rates, 4) have equivalent block align, and 5) have equivalent bit depth (bits per sample).

If either of the input files does not match these five requirements, the application will exit before performing convolution.

VITA

Andrew Keith LaChance was born in Lawrence, Massachusetts and grew up in Kingston, New Hampshire. He moved to Huntersville, North Carolina to finish the last two years of high school. He graduated from Appalachian State University in May 2009 with a BS degree in Computer Science. He entered the graduate program at Appalachian State University in August 2009. After completing his Master of Science degree in Computer Science in May 2011, Andrew LaChance becomes a Software Development Engineer at Microsoft Corporation.