GPGPU PROGRAMMING AND THE PITFALLS OF NAIVE DATA HANDLING

By

BRIAN PATRICK GORDON

Bachelor of Science in Electrical Engineering

Oklahoma State University

Stillwater, OK

2010

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2010

GPGPU PROGRAMMING AND THE PITFALLS OF

NAIVE DATA HANDLING

Thesis Approved:

Dr. Sohum Sohoni

Thesis Adviser

Dr. Louis Johnson

Dr. Damon Chandler

Dr. A. Gordon Emslie

Dean of the Graduate College

ACKNOWLEDGMENTS

I'd like to thank both Dr. Sohum Sohoni and Dr. Damon Chandler for the opportunity to work on the project that became this research topic. Dr. Chandler started the MBVR project within his research lab and handed over the development of the project to me when he and Dr. Sohoni became co-P.I.s on the project. Dr Sohoni provided me with all the equipment and support I could ever need to work on this project.

I'd also like to thank both Dr. Sohoni and Dr. Chandler as well as Dr. Louis Johnson for agreeing to be on my committee and taking the time to accommodate me.

I'd like to thank my fellow researchers in the CAESAR lab for being an ever present source of knowledge whenever I needed guidance. I'd especially like to thank Bradley Pesicka and Jeremy Smith who worked with me directly on the MBVR project.

Finally, I'd like to thank my parents who always provided encouragement and supported my desire to learn.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

While microprocessors are generally designed to handle any type of computation in software, increasingly complex algorithms and workloads can strain the processors limited resources, requiring a significant amount of time to complete the calculations. To compensate, co-processors are designed to offload these demanding computations and perform them with specialized hardware. Floating point co-processors like the Intel 8087 [1], secure socket layer accelerators like IBM's PCI Cryptographic Accelerator [2], and physics accelerators like the Ageia PhysX P1 physics processor are all examples of specialized hardware designed to offload the increasing burden of demanding workloads.

The most notable example of the co-processor found in most consumer computers is the dedicated graphics processor which excels in handing massively parallel work like 3D rendering. The combination of multi-threaded programming and the massive computational power of modern graphics processors can allow programs with high computational requirements to finish in less time. This is especially advantageous for real-time video applications that repeat those computations not only for every frame of the video, but also many times within the same frame. While processing on just the main processor is too slow to meet the real-time requirements of the application, moving the

slowest and easiest to parallelize code to the graphics processor can reduce the processing time to a more acceptable length.

1.1 Motivation

With the high data requirements for modern video transmission, users with limited bandwidth would not be able to receive the data stream fast enough to play the video in real-time. The alternative is to reduce the video quality drastically to reduce the data rate, often resulting in unacceptable video quality. One alternative the author is currently exploring is to animate a 3D model of the object in the video, a human head in particular, to recreate the video sequence. The only data required to recreate the animation is how the model changes to match the source images captured by the camera. This approach is especially advantageous for video conferencing applications with either limited bandwidth or multiple simultaneous connections.

However, the process of fitting the model to an image uses analysis-by-synthesis, a computationally intensive process. One type of deformation changes the model's face and the resulting rendered image is compared against the source image to see if the visual accuracy has improved. Previous incarnations of this process would require several seconds to encode one frame of the video sequence. This is simply too long for real-time encoding and transmission. By moving problematic code to the graphics processor, it is possible to get a significant speedup to the encoding process and achieve the goal of real-time encoding.

1.2 GPU application interfaces.

There are two well-documented and well adopted methods of performing computations on a modern graphics processor. Both of these are advantageous to this project since it already incorporates an application interface in the form of a 3D rendering API and each can easily operate with the output of the 3D renderer.

1. Graphics Shaders. Previously fixed stages of the 3D rendering pipeline have become programmable, allowing programmers to write their own code to create new visual effects. The pixel shader in particular is useful for computations because any program attached to it will run against all the pixels on the final image. When the output image is set to the dimension of the output data, each program executed by the pixel shader calculates the final value for one element of the output matrix
2. CUDA. nVidia's Compute Unified Device Architecture allows programmers to write their own code that will run natively on their graphics cards that support their unified shader architecture. CUDA programs are written in a C style language, compiled using nVidia's own compiler, and then linked into a standard C or C++ program. Unlike the graphic shaders, the programmer can specify the number of threads to allocate to a kernel. CUDA also includes optimized versions of basic math functions for their hardware.

1.3 Contributions

This thesis provides contributions to the field of computer architecture and computer science by exploring the benefits and pitfalls of rewriting a kernel of code for execution

on a highly parallel coprocessor. For the field of computer architecture, this paper examines the hardware utilization of each approach. This includes the efficiency of resource utilization and how the data bottleneck moves with each implementation of the encoder. These finding will help computer architects to design more efficient architectures that utilize large amounts of data and to design more efficient buses for transferring data between processors. This will also help graphic card drivers to make more efficient decisions with data handling.

For the field of computer science, this thesis explores the costs and benefits of utilizing graphics processors to improve the performance of time-constrained programs. By showing how a code segment behaves in each implementation, computer scientists can evaluate how their code will behave with each implementation.

1.4 Thesis Outline

The rest of the thesis is laid out as follows. Chapter 2 will discuss previous work with regard to utilizing both graphic shaders as well as native code generation for graphics cards. This chapter will also cover a brief history of the development of graphics cards and model based video representation.

Chapter 3 will discuss the full implementation of the model based video encoder as well as modifications to the base program to utilize the two methods of executing code on a graphics card. This chapter will also provide a description of the methods used within each implementation to perform each part of the calculations.

Chapter 4 will cover the testing methodology used to analyze each implementation. This will include the hardware used, program configuration, and data gathering methods along with justification for the choices made in this section.

Chapter 5 will present the data collected from the experiments and provide analysis of the results. Analysis will include a look at the results from each implementation including a comparison between each method.

Chapter 6 will discuss the conclusions and future work for the GPU based code as well as the model based video encoder.

CHAPTER II

REVIEW OF LITERATURE

This chapter covers background information about graphics processor architecture and efforts to utilize them for high performance computing.

2.1 Model Based Video Representation

Unlike traditional video representations, model based video representation recreates a video sequence by separately regenerating the video's content using models. This type of video compression was explored by Eisert et. al. [3] as an alternative to image based compression found in traditional video compression. The video encoder analyzes the input video and calculates specific scene parameters that will animate the same models in the decoder to recreate each of the input frames. This method is more space and bandwidth efficient than traditional video encoding for simple subjects like talking heads due to the much smaller amount of data required to recreate the video.

The encoder guesses the scene parameters using analysis-by-synthesis, recreating the head in 3D using a 3D model of a human head and the person's face. The same 3D model used to recreate the person in the video is also used within the encoder to compare against the camera's reference image. Following the process in Figure 2.1, the encoder changes one aspect of the head, renders the changed head, and compares the rendered

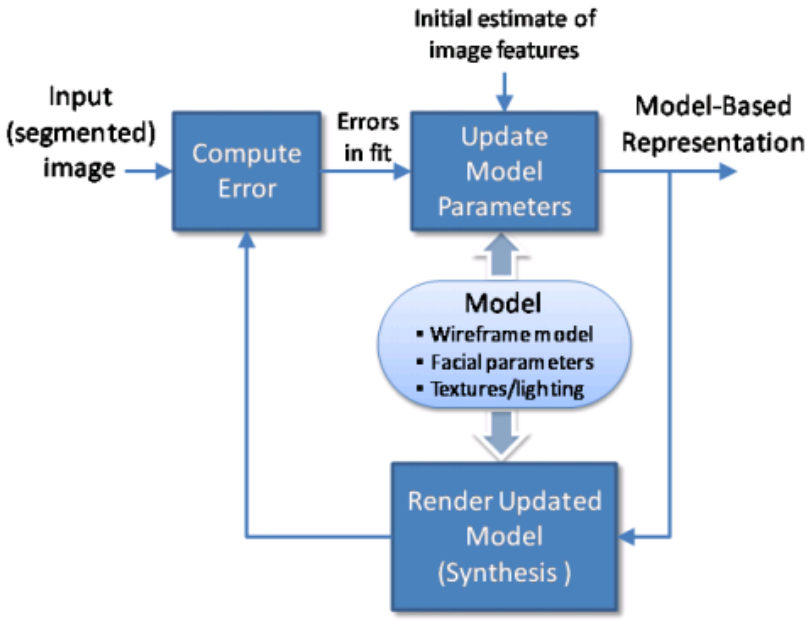

Figure 2.1 – Model Based Video Encoding Block Diagram

image to a frame from the camera's image of the person. If the change makes the model appear more like the reference image, the encoder will continue until that feature matches the reference. The process continues until all the features match the reference image for that frame and the feature modifications, called facial animation parameters, are sent to the receiving end. The next frame of the video is brought in and the process repeats.

2.2 GPU Architecture

To render a 3D scene and capture the resulting scene on a two dimensional rasterized image that is displayed on a monitor, a series of calculations on a large data set need to finish in a reasonable amount of time [4]. First, scene parameters like camera position and perspective are combined into a series of transformational matrices. These matrices, when multiplied with a vector containing the location of a vertex, will change the vertex's location based on the camera position. This process is repeated for every vertex

used in a scene in the vertex engine of the graphics card. Vectors are then calculated from connecting vertices, creating the 3D mesh of the object. This information is handed to the render output processors which calculate the points on the mesh that will be captured by each pixel in the output image.

Once points are chosen, the color of the surfaces affecting a pixel is calculated in the fragment engine by sampling the color of that surface. Most times, this point is located in between sets of vertices, requiring interpolation of the correct color from either the specific color given to the vertices or from an image chosen to texture that image. Other simple post processing alterations like lighting and transparency, also called alpha blending, change the final color of that pixel. Once every pixel color is calculated, the image is sent to the screen. With modern video games displaying hundreds of thousands of vertices and generating 2 megapixel images at 60 frames per second, the volume of computations required would not be possible on general purpose processors. Graphics processors, then, are designed as accelerators for the heavy computation load of the 3D rendering pipeline, which is illustrated in Figure 2.2.

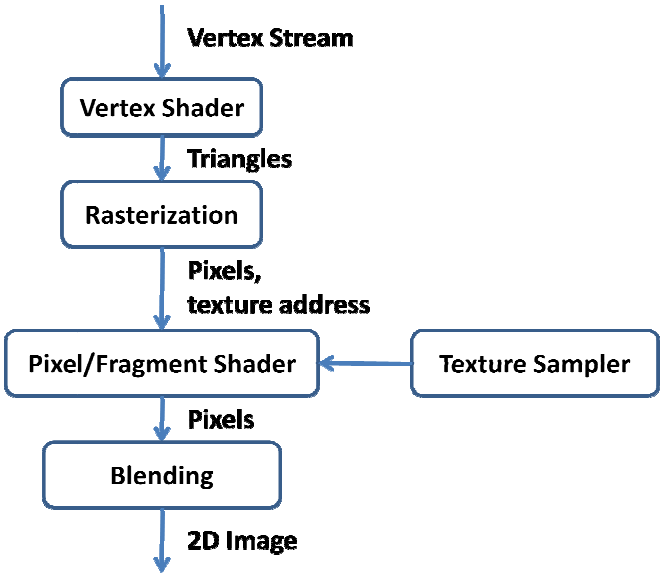

Figure 2.2 - 3D Rendering Pipeline. Vertex information and projection matrices are given to the vertex shader to draw polygons in 3D space. The raster output processor captures the base color of the pixels from the polygons. The pixel/fragment shader performs modifications to the pixel color and the blender combines all of the pixels that occupy the same location on the 2D image based on Z-depth and alpha value.

2.3 Programmable Rendering Engine

Originally, these processors were fixed in their functions, particularly with the handling of vertex data and final pixel color calculation. Over time, more user programmability was added, eventually leading to the ability to develop assembly programs for the vertex and pixel shader engines. Shaders are designed to execute using a single instruction, multiple data (SIMD) approach to parallelize execution among the multiple ALUs. These processors also contain vector pipelines to further parallelize data execution by allowing a single ALU to process multiple sets of data at once [5].

In the 3D rendering pipeline, pixel shaders (or fragment shaders) allow for post-processing after the 3D model is captured on a 2 dimensional plane and undergoes rasterisation to create the individual pixels. Pixel shaders are programs that are executed

in parallel for every pixel generated to change the final value based on any effects the programmer wants. These programs can then use their location on the render surface as well as the color and texture information to determine the final pixel's color. Pixel shaders are often used to determine the final color value of an object by modifying the base color with the lighting and material information of the object.

Pixel shaders have been used to accelerate image processing. Purde et al. [6] utilized pixel shaders to aid the processing of electronic speckle pattern interferometry. Utilizing an ATI 9700 AIW Pro, they were able to speed up their calculations to enough to process 11 frames per second. ATI has also released papers [7-8] detailing how to use DirectX 9 shaders for a range of simple and advanced image processing techniques. All of these papers do note that shader programs are limited in terms of their size and complexity, requiring several programs to perform more complex calculations like Fourier transforms.

2.4 CUDA

Because graphic shaders required learning to program in a new API, graphics card manufactures set out to create a more direct interface to run programs on their graphics processors. nVidia's solution was CUDA which utilizes the C/C++ programming language and their own compiler to create subprograms, called kernels, for their hardware. Standard CPU based programs invoke the CUDA kernels with two mandatory parameters: the number of threads that will be executed in inside each thread block, and the number of blocks inside a grid of blocks. These two numbers determine the total number of threads that will be executed for that kernel. Threads scheduled within each

block will execute in parallel while blocks are scheduled based on the available processing units on the card. This two tier hierarchy gives each thread a unique ID which allows each thread to address the specific data it is supposed to work on [9].

nVidia's unified shader architecture present in their graphics cards from the GeForce 8 series on as well as their supercomputing based Telsa line of cards organize their processors into groups called streaming multiprocessors, or SMs. The number of SMs can vary, determining the tier of the card, but each SM has 8 scalar processors, or SPs, giving the total number of processors advertized on the card. Within each SM, groups of up to 32 threads are executed simultaneously in a grouping called a warp. Warps are scheduled as SPs become available and execute in a single instruction, multiple threaded manner. While the entire warp is fed by a single instruction, the threads within the warp are free to follow their own path of execution based on their branching conditions. Threads that do not follow a branched path are disabled till the paths reconverge [10].

There are many papers that discuss how utilizing CUDA decreased computation time. Zhiyi et al. [11] looked at improving image processing performance using CUDA and saw improvements ranging from 8x up to 200x depending on the processing technique. Wei-Nien and Hsueh-Ming [12] improved the efficiency of the motion estimation step of H.264/AVC encoding up to 12x using CUDA. Changxin et. al [13] implemented an MD5-RC4 encryption algorithm in CUDA and saw a 3x-5x improvement in performance compared to a CPU based implementation. Most of these papers only focus on the workload being implemented in CUDA, while the rest also touch on optimizing their code for CUDA's architecture.

Finally, Amorim et. al. [14] performed an analysis of utilizing both OpenGL shaders and CUDA to accelerate the calculation of a weighted Jacobian iteration. They found that utilizing CUDA produced the greatest increase in speed, but they also investigated how changes in programming style and graphic memory interfacing affected the performance of their program. Unlike the research in this paper, their data originated in system memory and OpenGL was only used to handle their computations.

CHAPTER III

IMPLEMENTAION

This chapter will discuss each of the implementations of the MBVR encoder and the expected benefits of each implementation.

3.1 Initial MBVR implementation

The model based video encoder is an analysis-by-synthesis encoder that tries to match a 3D model of a person's head to an input image from a video camera. The encoder uses a set of facial animation parameters (FAP) that control specific parts of the face and how each part deforms. The encoder changes one FAP, renders the changed head, and calculates the PSNR of the resulting guessed rendering. If the change is closer to the reference image, the program continues changing the model till the rendered image stops improving its guess. If the first direction of search proves fruitless, the encoder moves the FAP in the opposite direction and continues if the encoder sees improvements. The encoder continues with the rest of the FAPs once each one settles on its best value. Once all the FAPs have been optimized, the encoder looks at the total improvement. If there was a significant improvement, the encoder loops through all the FAPs again to look for further improvements. Once improvement in the PSNR falls below a certain threshold, the resulting FAP values are saved for that frame. The next frame of the video is loaded

as the new reference image and the best guess from the previous frame is used as the starting point for the new optimization.

3.2 Graphic Shaders in Direct3D

Unfortunately, the first incarnation of the MBVR encoder required several seconds to converge on a best guess for a single frame. An initial investigation revealed that the peak signal-to-noise (PSNR) error calculation was the bottleneck for performance. Since one of the inputs for the error calculation was originating on the graphics card, and since modern graphics card support simple programs, is made sense to move the slowest part of the encoding process to the graphics card to reduce the encoding time. There are two main advantages to this strategy.

1. Moving data between system memory and graphics memory is slow. The baseline encoder requires the entire rendered image to be copied to system memory for each error calculation.
2. The PSNR calculation is based on the mean square error (MSE) of the reference image and the current guess. The bulk of the calculations are done between individual pixels in each image. These independent calculations are being serialized on the CPU and can be executed in parallel, which the graphics card supports well.

By moving the calculation to the graphics card, both of these bottlenecks can be mitigated and only a single value needs to be copied back to system memory. To calculate the error for each guess using the main microprocessor, the 3D head is rendered to an off-screen buffer called the back buffer. The image is then read from this buffer to

an array of sub-pixel values in the system memory. The MSE calculation takes each sub-pixel value from the reference image and from the rendered image and uses them in the MSE equation

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{x}_i - x_i)^2$$

where N represents the number of pixels, $\hat{x}_i$ represents the reference image element, and x_i represents the trial image element. The MSE value is then fed into the PSNR equation

$$PSNR = 10 \times \log_{10}\left(\frac{MAX^2}{MSE}\right)$$

where MAX is the maximum value for each color component. Since the program is working with unsigned character values, this value is 255. The GPU implementation will move a subset of the MSE calculation into GPU compatible code:

$$MSE' = \sum_{i=1}^{N}(\hat{x}_i - x_i)^2$$

The rest of the MSE calculation will be merged with the PSNR equation to give

$$PSNR = 10 \times \log_{10}\left(\frac{N \times MAX^2}{MSE'}\right)$$

which will still be calculated on the CPU. The C++ code used to calculate the modified MSE is located in Appendix A.

The rendered image using the guessed parameters needs to be stored as a texture for a second rendering pass, which will perform the MSE calculation. This is easily facilitated by copying the contents of the back buffer, only this time the destination target is a texture stored in the video card's memory. Another method of accomplishing this is to render the 3D head to an off-screen rendering surface. An off-screen rendering surface

acts like the back buffer as the render target, but it can automatically store its data into other usable objects. One of the objects that it can use as a storage object is a texture. By rendering to an off-screen rendering surface with a texture attached to store the color information, that texture is automatically generated from the results of the first rendering pass.

The second rendering pass creates a long sequence of squares equal in number to the pixel count of the two images. These squares will contain the mean square error for one pixel in the image. These squares are arranged one behind another as seen in Figure 3.1. Each square is colored with one pixel's color from both the reference image and the synthesized images, which are both mapped as textures to each square. Once the rendering process gets to the pixel shaders, the shader program takes the color information from the two textures and performs the subtraction and squaring part of the MSE calculation. The resulting value is stored in the red channel, thus becoming the new color for that square. The shader program is located in Appendix A.

Once all the shader programs have finished in the second rendering pass, the rendering process uses alpha blending to perform the summation step and generate a final value. Alpha blending is normally used for rendering a transparent object by calculating

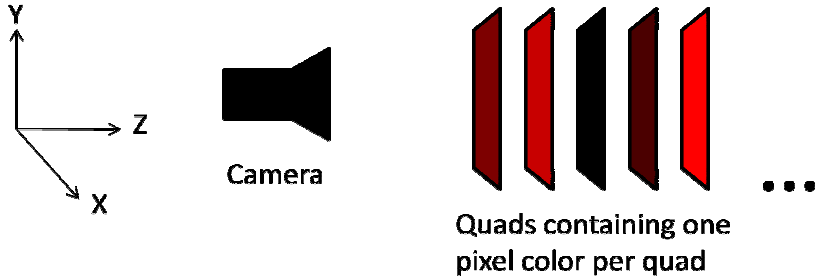

Figure 3.1 - Shader Method Scene Setup. The first quadrilateral fills the camera viewing area while all the other quads are placed directly behind the first. An orthogonal projection is used to prevent the furthest quads from shrinking due to perspective.

the final pixel value on the screen as a combination of the color of the surface closest the camera and the color of every surface behind it [15]. Normally the amount of color used from each polygon is determined by the front polygon's alpha value. However, it is possible to tell the alpha blender to ignore the alpha value and simply add each value without the alpha weighting. The entire rendering pass is done on a one pixel squared off-screen rendering surface that stores its color information as a 32-bit float. The pixel's value, which is now the MSE, is read from this buffer and used for the remainder of the PSNR calculation.

3.3 OpenGL port

In the interest of comparing how different 3D rendering APIs handle data, and because CUDA cannot access pixel information in Direct3D, the base code was ported to OpenGL. OpenGL offers the benefit of running on multiple operating systems whereas Direct3D requires Microsoft Windows. In addition, OpenGL natively supports copying pixel information to a buffer in system memory via DMA [16]. Direct3D required an explicit memcpy statement from a memory mapped address on the graphics card to copy the data from graphics memory to system memory. Utilizing DMA allows the data copy to without requiring the CPU to perform the copy and should provide a large speed improvement if the data is accessible when it is needed.

In order to keep the data files used between every implementation consistent, custom routines were written to read the Direct3D based file containing the 3D mesh for the head. Methods for reading in other required files were copied from the Direct3D code. To enable compilation on any operating system without modification, only cross

platform libraries like the standard C++ libraries, the OpenGL Utility Toolkit (GLUT) [17], and the OpenGL Extension Wrangler (GLEW) [18] were utilized. The error calculations are still executed on the CPU for comparison's sake.

3.4 CUDA

CUDA offers the most flexible interface to create code that runs on nVidia's graphics cards. nVidia's libraries provide familiar memory allocation and copying commands to create and initialize variables inside CUDA. In addition, CUDA offers integration with OpenGL to create or modify data in OpenGL buffer objects, allowing CUDA to perform more complex computations than OpenGL's own vertex or fragment shaders without requiring the main program to handle the data transfer. This is especially advantageous for the encoder since the majority of the data movement occurs between OpenGL and the processor performing the MSE calculation. If the interoperability can move the data from OpenGL to CUDA while staying on the graphics card, the encoder can benefit from the reduction of the large data copy over the slow CPU-GPU link and the encoder can use CUDA to calculate the MSE much quicker.

In order to get the pixel data into an object that CUDA can use, there are two options available to programmers. The first requires the main program to read the pixel information back normally and use cudaMemcpy to copy the data back to CUDA controlled memory on the GPU. The other is to use CUDA's OpenGL interoperability to access OpenGL buffer objects by mapping the buffers onto its own address space. To get the pixel data into a pixel buffer object, the buffer is first bound into the OpenGL workspace, setting itself as the copy target. Then the back buffer's pixel data is read

using the OpenGL glReadPixels function and copied into the pixel buffer. The pixel data is now accessible to CUDA to map the data into CUDA controlled memory. Both methods were implemented to analyze the difference in the data handling and their effects on performance.

The CUDA program is comprised of three separate kernels of code that procedurally process the MSE. The first kernel reads the reference and synthesized images as full integer arrays comprising of 3 bytes of actual pixel data, 1 byte per color channel, and 1 byte of superfluous data. The superfluous data is comprised of the alpha channel, which is manually added to the reference frame since bitmap images do not store this information. This additional padding data is used to align memory reads from global memory along consecutive four byte aligned addresses. By aligning the memory reads with the thread index, the memory controller on the graphics card will coalesce the memory reads from the consecutively indexed threads into a single large memory read [19]. If global memory accesses do not follow this pattern, all the data retrieved from a single coalesced read would be split into individual reads. Because there is a large latency penalty to access the global memory in CUDA, coalescing memory reads is essential to maximizing performance.

After the data is read in, each thread performs the subtraction and multiplication of the three subpixel values for one pixel. The resulting values are added together and stored into high speed, low latency shared memory. Shared memory is a small on-chip memory that is accessible to every thread within a thread block to facilitate the fast sharing of data between threads. To sum all the individual mean squared errors, each thread must add their result to a single location. It is not possible to have each thread

attempt to add their value to a single memory location simultaneously. Since each thread will read the current value from memory at the same time, each thread will add its value to the current value instead of each thread adding to the result of the previous thread's additions, resulting in a write after write hazard.

To accomplish this summation efficiently, half of the threads in a thread block will add the value the other half of the threads to their own. From there, the threads from the first subgroup continue to split into subgroups and sum the values till all the values are accumulated with the first thread result. The first thread then writes the value back to global memory where the next set of CUDA kernels performs the same reduction on the intermediate results. The final kernel combines the last of the intermediate values into a single value that is copied back to the main program and used to finish the PSNR calculation. The entire CUDA kernel is located in Appendix A.

3.5 Computation Time

To compare the raw computation times, the error calculation was isolated and written into separate programs. The calculation was split to the two major computations, the highly parallel subtraction and squaring and the highly serial cumulative summation. Each of these parts and the whole MSE calculation were timed to compare the computation times without the data transfer and other parts of the program.

CHAPTER IV

METHODOLOGY

Due to the lack of simulation software that can simulate modern graphics hardware, all tests were run on a physical machine. The test consisted of executing the programs described in Chapter 3 and analyzing the resource usage on the machine.

4.1 Testing Environment

All tests took place on a Nehalem-based Intel Core i7 920 processor running @ 2.66 GHz. The rest of the system specifications can be found in Table 1. The graphics card used in the experiments is the nVidia GeForce 9800 GTX+. This card features 128 stream processors running at 1.836 GHz and 512 MB of GDDR3 memory operating at 1.1 GHz.

All the programs were tested on Windows XP with service pack 3 using nVidia's closed source driver version 196.21. All of the OpenGL programs were tested using Ubuntu Linux 9.04 using nVidia's closed source driver version 190.42. The Windows

Table 4.1 – System Specifications.

L1 Instruction Cache	32 KB per core, 4-way associative, 64 B lines
L1 Data Cache	32 KB per core, 8-way associative, 64 B lines
L2 Cache	256 KB per core, 8-way associative, 64 B lines
L3 Cache	8 MB shared, 8-way associative, 64 B lines
System Memory	3 GB DDR3 @ 1066 MHz
Chipset	Intel X58

version of the programs were compiled using Visual Studio 2005 using the August 2008 version of the DirectX SDK and the OpenGL Utility Toolkit version 3.7.6. The CUDA implementations also utilized version 1.5.1 of the OpenGL Extension Wrangler to provide access to advanced OpenGL structures and functions. The CUDA implementation was compiled using version 2.3 of the CUDA toolkit.

For the model based video encoder, the encoder first played back a prerecorded set of facial animations called *wow*, a commonly used set in MPEG-4 facial animation research [20-22], and saved the first 300 generated images as 352x288 CIF resolution bitmap files. These images acted as the source video sequence that the encoder will try to match the same 3D scene. These images were stored and accessed from a RAM disk to simulate access from a video device and remove the performance penalty of hard drive access. Since the accuracy of the encoding process is not the subject of this investigation, the FAPs generated from the encoding process were not stored after each set of optimizations.

4.2 Functional Timers

Utilizing the high precision timer in the operating system, functional timers were added to the applications testing the execution time of different implementations of the MSE code. The time taken to execute each function was accumulated for the entire run to even out the microsecond resolution of the timer functions across the entire encoding process. In addition, the Linux application time and the Windows XP application *timeit* timed the execution of each application.

4.3 Performance Counters

To analyze how the system is being utilized, Intel VTune and OProfile collected system data from CPU performance counters built into the processor. These counters are monitored in the background while a target application or environment is running and monitor specific events chosen by the user. Events can include execution time, cache accesses and misses, pipeline stalls, memory accesses, instruction types executed, and off-chip bus accesses. The number of counters on a Core i7 processor is limited to 4, so multiple runs are necessary to collect every type of data available.

VTune [23] monitored execution time of each the Windows programs to monitor where the most time is spent within the program. Since the video encoder is memory intensive, VTune also monitored cache access and hit rates and pipeline stalls for each section. VTune configuration for each session is provided in Table 2 with group numbers denoting the events that were run together.

Table 4.2 – Performance Counter Events

Group	Event Name	Events per count
1	CPU _CLK_UNHALTED.P_THREAD	10000000
1	RAT_STALLS.ANY	10000000
2	MEM_LOAD_RETIRED.LLC_UNSHARED_HIT	100000
2	MEM_LOAD_RETIRED.OTHER_CORE_L2_HIT_HITM	100000
2	MEM_LOAD_RETIRED.LLC_MISS	100000
3	L2_RQSTS.LD_HIT	1000000
3	L2_RQSTS.LOADS	1000000
3	MEM_LOAD_RETIRED.L1HIT	10000000
3	L1D_CACHE_LD.ANY	10000000
4	RESOURCE_STALLS.ANY	1000000
4	RESOURCE_STALLS.RS	1000000
4	RESOURCE_STALLS.ROB	1000000
4	RESOURCE_STALLS.STORE	1000000

OProfile [24] collected system data for the Linux applications. OProfile is an open-source system profiler for Linux that monitors the performance counters on modern Intel and AMD processors.

4.4 Instruction Type Profiling

For additional analysis of the difference between programs, the Pin instrumentation program [25] provided a breakdown of instruction types, memory transactions, and bytes of memory transferred in each program. Pin dynamically inserts C/C++ code inside running programs to gather statistics about the target program while it program executes. One of the sample instrumentations, insmix, provided counts each type of instruction, memory accesses by data size, atomic memory accesses, and stack accesses throughout the entire program and by function. Pin's analysis of the program execution was used to compare the characteristics of the programs and their memory access patterns.

CHAPTER V

FINDINGS

This chapter will discuss the data found through the experiments outlined in the previous chapter. The chapter is divided into four sections. The first three sections will discuss the system performance of the encoder in three different environments. The fourth section will analyze the raw computational performance of each processor on the error calculation.

5.1 Direct3D in Windows

Starting with the Direct3D implementations of the original code, there was a drastic reduction in the total execution time, which is shown in Table 5.1. The CPU version of the code required Direct3D to copy the rendered image to system memory so that main program could calculate the mean square error between the reference image and the rendered image. The Direct3D version of the program showed a large decrease in the total execution time, showing that the GPU based error calculation and the reduction of the amount of data transferred back to system memory. While the reduction in time is

Table 5.1 - Total Execution Time (Windows OpenGL)

	CPU	Direct3D
Total Time (seconds)	463.687	83.562
Frames per second	0.647	3.59

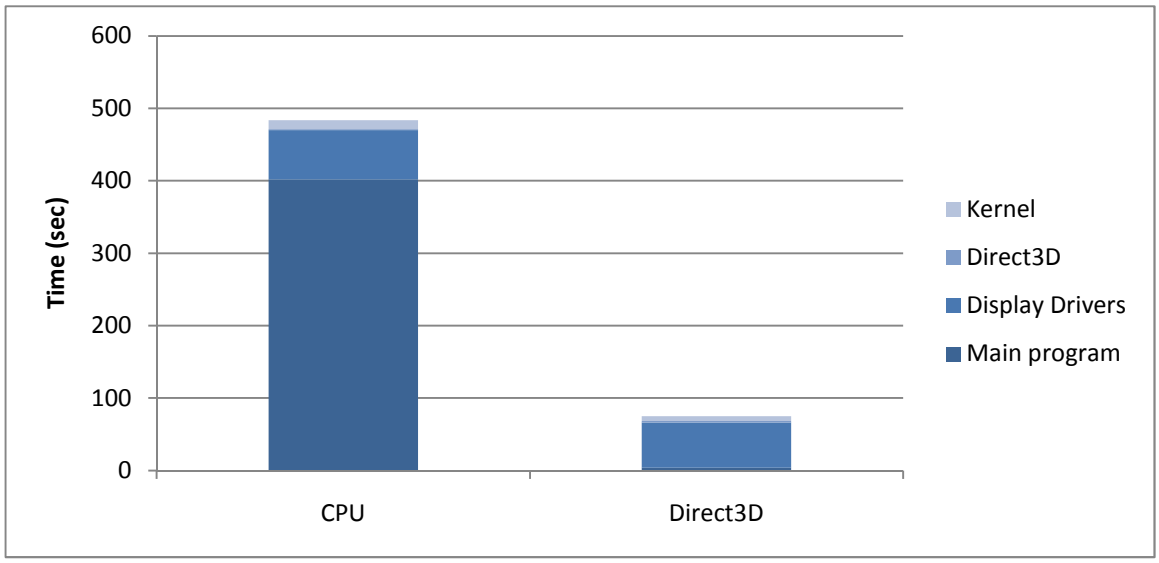

Figure 5.1 – Windows Direct3D Time by Libraries. While the Direct3D code increased slightly from the additional code required to perform the error calculation, the number of cycles used in main program and other support libraries reduced considerably.

substantial, it still did not come close to the desired real time performance goal of 30 frames per second.

Since the speedup was not sufficient, an investigation into the new bottleneck was needed. Looking into where the most time was consumed, based on the unhalted cycle count during the execution, Figure 5.1 shows a clear reduction in the number of cycles within the main program. The only increase in the execution time in any library with the GPU version is the Direct3D library, which is expected since additional function calls were required to perform the calculation on the GPU. However, 83.6% of the time is spent inside the display driver, which remained relatively unchanged.

Moving on to the cache statistics, the drop in the execution time reduced the number of accesses to the cache, as expected. The normalized number of cache access can be found in Figure 5.2. The number of accesses in the main program dropped, signifying that the error calculation was memory intensive and that the movement of that

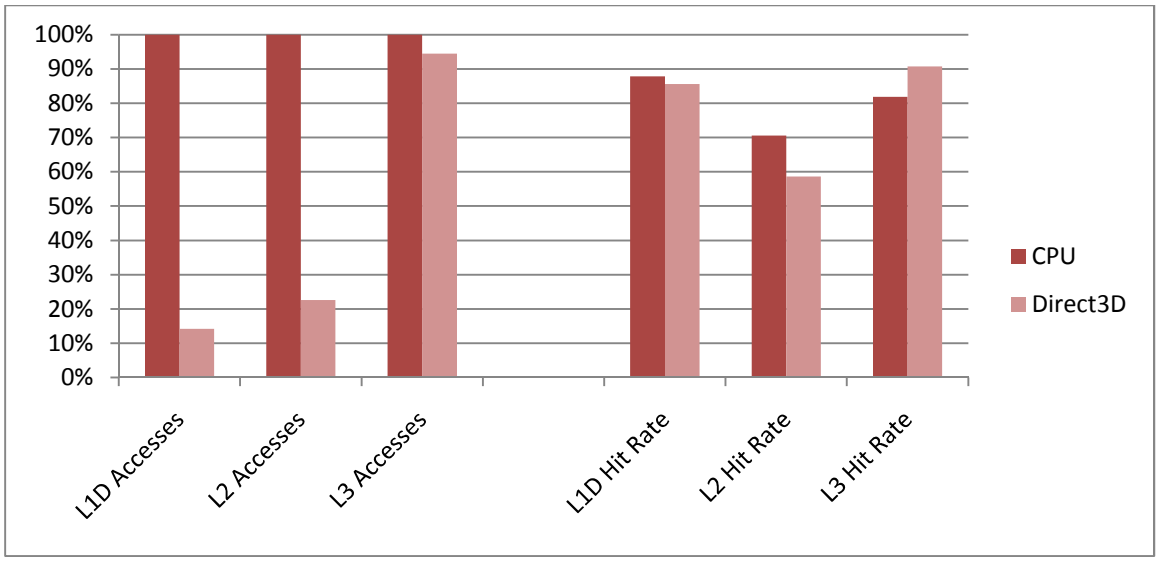

Figure 5.2 - Windows Direct3D Cache Statistics. The bars on the left correspond to the total number of access for each level of the cache. The bars on the right correspond to the hit rate. With the reduction in the total execution time, the number of L1 and L2 cache data access reduced accordingly. The L3 cache remained relatively the same, suggesting that the error computation did not access the L3 cache.

code reduced the burden of the main processor. However, the number of L3 cache access remained relatively the same. It turns out that the display drivers are again the culprit with the majority of the L3 cache accesses. In fact, the display driver's cache activity remains roughly the same between the two versions of the code.

Finally, focusing on the location of the stalls in the CPU pipeline should illustrate what types of operations were slowing the overall execution. In Figure 5.3, the two major sources of stalls were caused from a filled reservation station or an unavailable store unit. Given the highly parallel nature of the MSE code, the reservation station stalls were likely caused by the out-of-order execution engine's attempt to parallelize the code within its own superscalar architecture. The reduction of these stalls lends further evidence to correlation of reservation stalls to the degree of parallelization of the code. The dramatic reduction in the store stalls gives correlation to the amount of data transferred between the main processor and the graphics processor. Since the processor

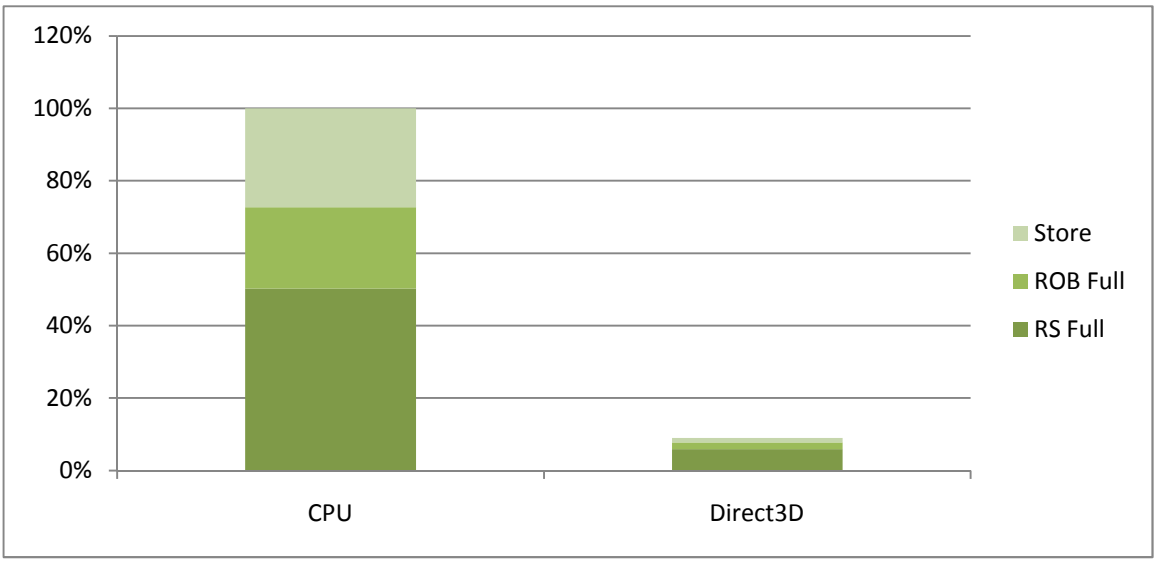

Figure 5.3 - Windows Direct3D Normalized Stalls. Moving the error calculation into Direct3D dramatically reduced the number of processor stalls, especially with the two largest sources of stalls.

was only transferring the 4 byte result from the error calculation instead of the 304 kilobytes in the synthesized image to perform the calculation on the main CPU.

5.2 OpenGL in Windows

Swapping out Direct3D for OpenGL and the shader computations for CUDA, there is a startling reduction in the total runtime for the CPU version of the code. More startling is the increase in total runtime for CUDA, regardless of how data is moved from OpenGL to CUDA.

Focusing on where the time is spent inside the program, Figure 5.4 breaks down the execution time in each of the major libraries. As expected, the main program's execution time reduced when the error calculation moved from the CPU to CUDA.

Table 5.2 - Total Execution Time (Windows OpenGL)

	CPU	CUDA	CUDA w/ PBO
Total Time (seconds)	54.312	74.140	98.359
Frames per Second	5.52	4.05	3.05

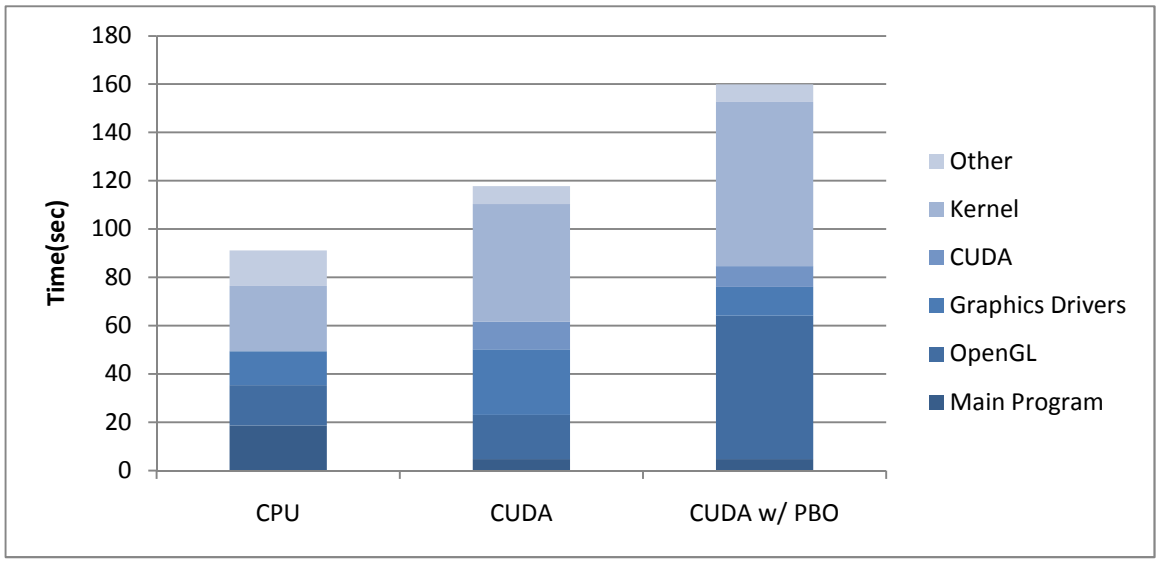

Figure 5.4 – Windows OpenGL Time by Libraries. Even though the CUDA versions were supposed to reduce the total number of cycles, the overhead of utilizing CUDA moved execution time from the main program to the Windows kernel and support libraries.

However, the amount of time spent in the Windows kernel increased when CUDA is introduced to the program. In addition, number of cycles in the graphics driver and OpenGL library increased dramatically depending on which library is moving the synthesized image from OpenGL to CUDA. Based on these results, the process of moving the calculation to the GPU increases the amount of work required of the main processor.

Looking at the cache statistics, some interesting patterns appear. In Figure 5.5, the total number of cache accesses stays approximately the same with the CUDA implementation that did not utilize a PBO to move the image data. However, more of the cache hits moved from the L1 and L2 caches to the L3 cache. The movement of cache access is much more apparent in the second CUDA version. In addition to the lower L1 and L2 hit rates and the higher L3 cache hit rate, the total number of access to all levels is much higher. Employing a PBO to utilize CUDA's OpenGL interoperability as a method

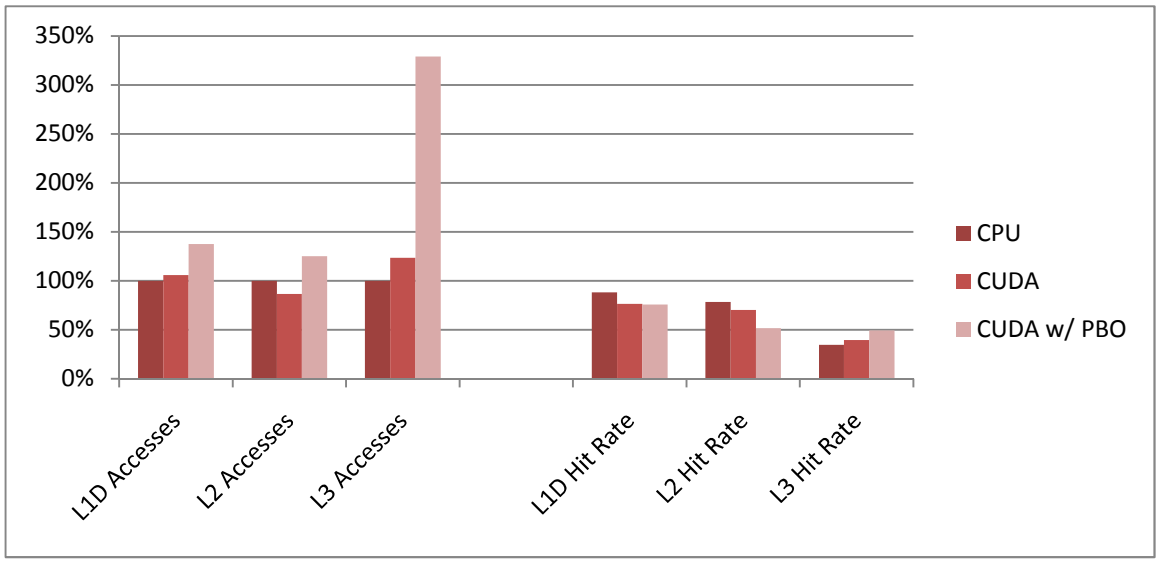

Figure 5.5 – Windows OpenGL Cache Statistics. While the total number of access between the CPU and CUDA without PBO remained the same, both CUDA versions saw lower hit rates. In addition, the CUDA version with a PBO saw substantially more cache accesses.

of moving the pixel data from OpenGL to CUDA appears to be a more inefficient method of moving data.

Finally, the pipeline stalls shed some additional light on the slowdowns. As seen in Figure 5.6, the number of store unit stalls increased dramatically when CUDA was utilized. However, the total number of stalls was lower with the CUDA program that did not use a PBO than the CPU version. With the exception of the store stalls, this version of the encoder appears to be more efficient with the processor. The CUDA version with the PBO, however, showed over twice as many stalls as either of the other programs. In addition, there were more reorder buffer stalls than the total number of stalls in either program. While the OpenGL interoperability offers an easier means for programmers to move data between OpenGL and CUDA, the overhead of utilizing a PBO and the interoperability APIs adds to existing overhead of moving the data.

Figure 5.6 – Windows OpenGL Normalized Stalls. Implementing CUDA increased the number of store unit stalls, but utilizing the OpenGL interoperability built into CUDA more than doubled the total number of stalls.

Based on the all this data, the OpenGL interoperability in CUDA adds substantial overhead to the data copy between the two application interfaces. Compared with the CUDA version without using a PBO, the only difference in the implementation is the method of the data copy. However, the requirement to copy the data from graphics memory to system memory and back to use CUDA negates one of the primary reasons for using CUDA in the first place: reduce the amount of data copied between system memory and graphics memory.

5.3 OpenGL in Linux

Switching to the Linux versions of the OpenGL code, there is a similar increase in total runtime, shown in Table 5.3, between the CPU and CUDA versions of the code that was

Table 5.3 – Total Execution Time (Ubuntu OpenGL)

	CPU	CUDA	CUDA w/ PBO
Total Time (seconds)	56.367	71.929	81.787
Frames per second	5.32	4.17	3.67

seen in the Windows versions of the same programs. In addition, the CUDA version that utilized the OpenGL interoperability was about 10 seconds slower than the version that explicitly forced the pixel data to detour through system memory before jumping back to the graphics card for CUDA.

The number of cycles from the main program drop drastically and the OpenGL and CUDA libraries use more cycles in both of the CUDA programs, as seen in Figure 5.7. This is similar to the results from the Windows version of these programs. However, the total number of unhalted cycles stays fairly consistent between each of the programs. If every version of the encoder uses about the same number of cycles, there has to be a significant number of stalls to account for the additional amount of execution time.

Looking at the cache statics shown in Figure 5.8, every cache level saw a lower hit rate, especially L1 data cache. As a result, cache accesses are moving to the slower, lower levels of the cache. Moreover, the additional number of accesses to the higher

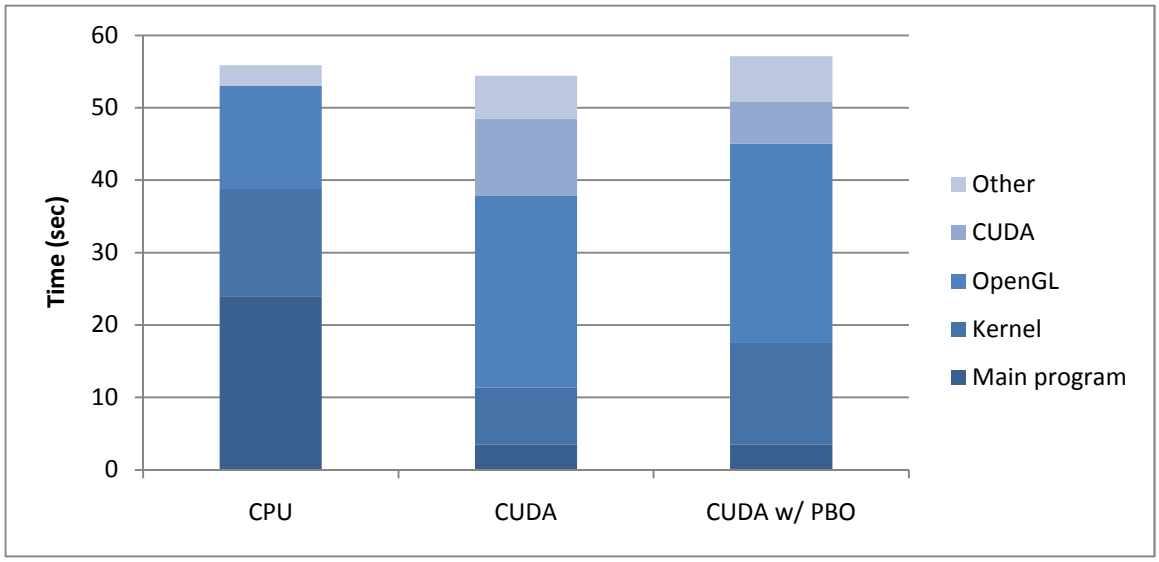

Figure 5.7 – Ubuntu OpenGL Time by Libraries. While every version of the code displayed approximately the same number of unhalted cycles, a large portion of the cycles moved from the main program to the OpenGL and CUDA libraries.

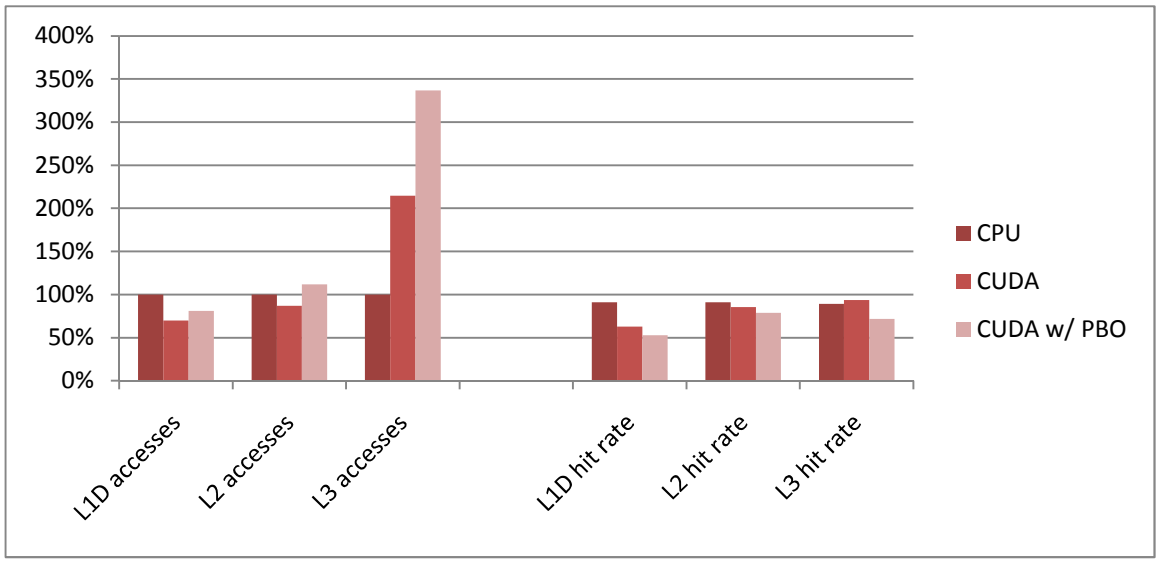

Figure 5.8 – Ubuntu OpenGL Cache Statistics. Cache access numbers are normalized to the number of accesses in the CPU version. With the lower number of accesses and hits in the lower level caches, accesses in the CUDA versions are moving to the higher level caches. This move in the location of the memory reads could account for some of the lost performance.

level caches, especially the level 3 cache, suggests that the backend libraries are caching the large amount of data (352x288 pixels at 4 bytes per pixel to allow memory access coalescing in CUDA) as it moves from the system memory to the graphics memory. If the programs are caching the images in the process of copying them, the rest of the program data is probably getting clobbered as well in the lower level caches, resulting in the lower hit rates in all the caches. Since the entire image is copied to system memory before moving to CUDA, marking the data as uncacheable would only add to the latency since the image would have to come from main memory instead of the cache when then program copies the data to CUDA.

Focusing on the processor stalls shown in Figure 5.9, the main source of stalls in the CPU version of the program come from a full reservation station during the error calculation. This mirrors the results seen in the two Windows programs as the processor is probably trying to parallelize the error calculation. Looking at the CUDA versions, the

Figure 5.9 - Ubuntu OpenGL Normalized Stalls. Moving the error calculation increased the number of processor stalls significantly. While there were additional stalls at the reservation station, the number of stalls due the store unit increased by an order of magnitude.

number of stalls increase 2x-4x over the CPU version with additional stalls coming from the kernel, OpenGL, and CUDA libraries. While the CUDA version without the PBO only saw a slight increase in reservation station stalls, the store unit stalls increased 22x, signifying that CUDA's data copying is causing the additional execution time. The CUDA version with PBO saw a 3x increase in reservation station stalls and a 40x increase in store unit stalls. The apparent inefficiency with data handling can explain why the OpenGL interoperability built into CUDA is slower.

When the CUDA programs were tested with different image sizes, the number of store unit stalls increased with the new number of pixels. As seen in Figure 5.10, both CUDA programs saw a proportional increase in the store unit stalls with the increase in the amount of data required to encode the 300 frames. Based on the relationship of the increases, the CudaMemcpy functions are using the CPU to move the data to the graphics card. In addition, the code utilized the asynchronous variant of the function, CudaMemcpyAsync, to reduce the overhead of the copy. Looking at the libraries used in

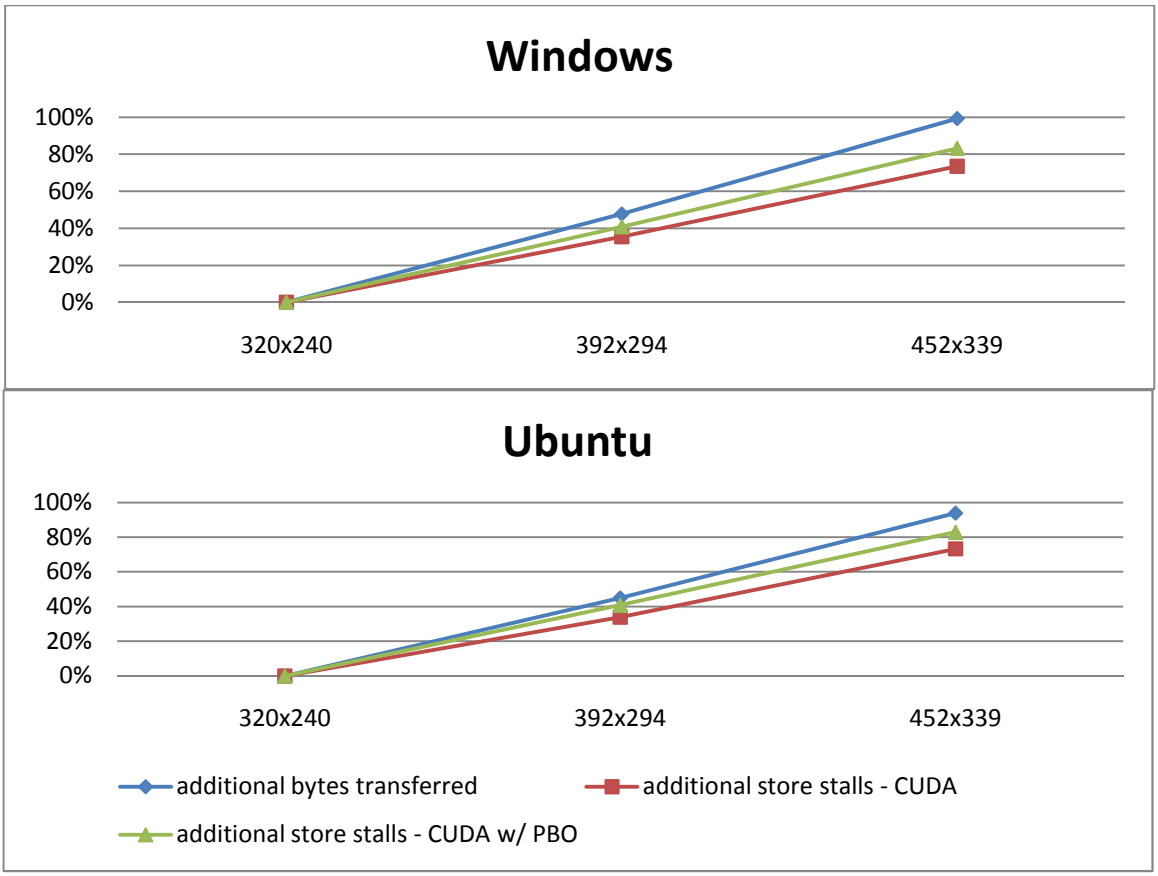

Figure 5.10 – Additional Store Stalls vs. Resolution. All the numbers are normalized to the smallest resolution. While the number of stalls in either program did not scale at the same rate as the data size, the scaling still follows the linear trend with the data size

the program, the multithreading library pthreads was used, despite the fact that the program never explicitly included or invoked that library. Based on these observations, CUDA is probably creating additional threads in the background to service the asynchronous memory copy.

There could be two sources of the store unit stalls present in both CUDA programs. The most likely source is the image data passing through the CPU and main memory as it is copied from OpenGL managed memory to CUDA managed memory. The other source of the stalls could be the CUDA kernels themselves, as these GPU instructions must be copied to the graphics card from main memory. Since processing a larger image would require more instances of the kernel to execute, the additional store

Table 5.4 – Store Unit Stalls in CUDA Programs (10,000 runs)

	Execute Kernel	Transfer & Execute Kernel
Store Unit Stalls	18,000,000	1,334,000,000

unit stalls could have originated from the transfer of the CUDA kernel. To test this, two simple CUDA programs were written to test the effect of the data transfer on the number of store unit stalls. The first program only executed the kernel while the second program copied the equivalent amount of data as one of the rendered images before executing the kernel. After each procedure was completed 10,000 times, the store unit stalls were collected. The stalls are shown in Table 5.4. Clearly, the source of the majority of these stalls originates in the data transfer.

Finally, while no documentation could be found to confirm that normal use of OpenGL's *glReadPixels* used DMA, reading the pixel information into a PBO is will use DMA if it is available. [16] To confirm whether or not the CPU based OpenGL encoder is using DMA to transfer the image data back to the system memory, an additional encoder was written that used a PBO, and subsequently DMA, to copy the data back to system memory. When the PBO is mapped into addressable memory, it will behave like the character array used in the original program, allowing the error calculation code to remain unchanged. If the CPU program that does not use the PBO behaves similarly to the one that does, it will be assumed that the process of reading pixel data back using *glReadPixels* will use DMA, regardless of the target.

Looking at the measurements taken from the *time* command, OProfile, and Pin in Figure 5.11, there is less than 1 percent difference in every metric between the two CPU based programs. In addition, the program that used a PBO consistently came out higher

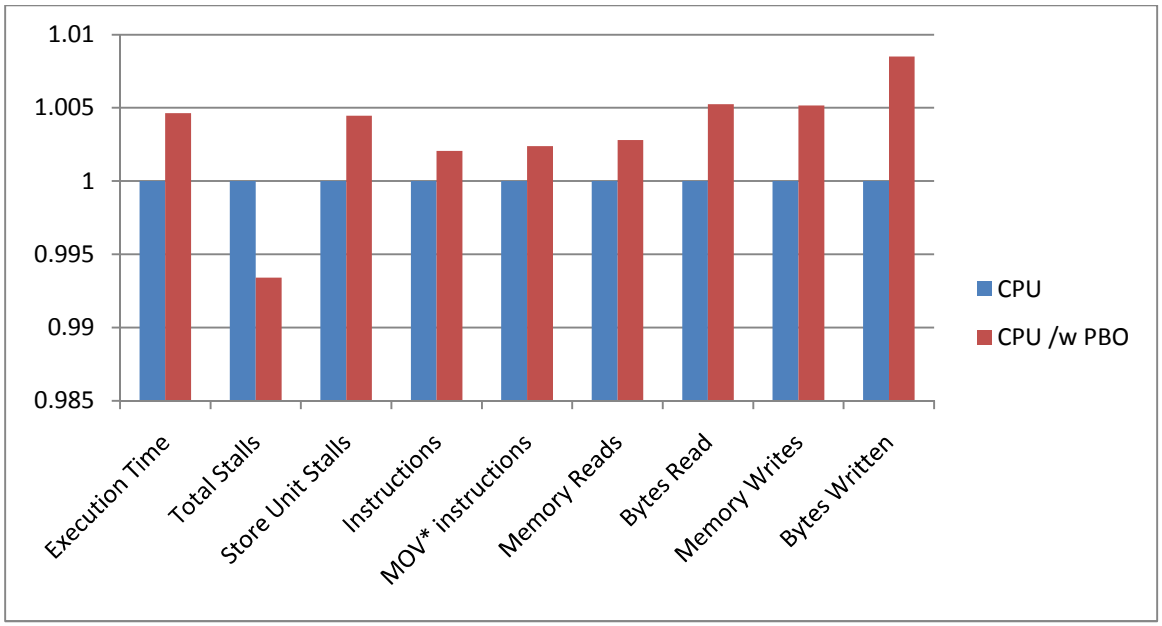

Figure 5.11 – Normalized Comparison of Program Characteristics. There is very little difference between these two programs despite using different means of transferring the pixel data to system memory. Based on this, the program that used a PBO, and subsequently DMA transfers, and the program that did not use a PBO behaved almost identically.

on almost every measurement, which either statistical noise or the small amount of additional code required to implement and use a PBO could account for. If the program that did not use a PBO used the CPU to transfer the pixel data, there should have been substantially more memory transactions within that program. Therefore, it is assumed that *glReadPixels* uses DMA to transfer the pixel data to system memory.

5.4 Computation Time

As expected, the highly parallel subtraction and squaring computation saw a dramatic speedup. Since all the calculations are completely independent of each other, the code parallelized well. Interestingly, there was only a 5% increase in cumulative sum time. The parallelized reduction sum algorithm used allowed the slower shader cores on the GPU to compete with the faster Core i7 processor. In the end, the CUDA version was able to perform the MSE algorithm 70% faster.

Table 5.5 – Time to Complete 100 Error Calculations

	CPU (µs)	CUDA (µs)	Speedup
Subtract & Square	24529	1760	1290%
Cumulative Summation	11201	11777	-4.89%
Complete MSE	24641	14489	70.1%

CHAPTER VI

CONCLUSION

This chapter provides some discussion on the finding from this thesis and provides future means of implementing code on a graphics card.

6.1 Value of Coprocessors

This project has continuously focused on the second half of Amdahl's law: optimize the bottleneck. With the need for 3D rendering, an existing 3D rendering API and a modern graphics card made sense. A software rendering engine would have been the bottleneck of the encoder. With the original Windows program using Direct3D, the bottleneck was the error calculation. Utilizing the computational power of the of the graphics card, the encoder finished in under 1/5 of the original time. While the use of the graphics shaders was not ideal, it did provide a means for performing the error calculation on a more appropriate processor. In addition, the CUDA version of the algorithm could perform the mean squared error computation faster than the CPU due to its highly parallelized code. If any part of a serialized code can be parallelized to any degree, total computation time can be reduced.

However, as long as the data transfer time consumes any time gained from the coprocessor's computation, adding a coprocessor to a solution will not produce any tangible benefits. Due the large data size and the simple calculation, this project did not

see the 70% speedup of the error computation due to the time required to move the image into CUDA's memory space.

6.2 Discovery of Naive Data Handling within CUDA

The biggest surprise of this study was the OpenGL version and its use of a DMA transfer for the pixel data. Since both the Direct3D versions and the CUDA versions implemented some form of a memcpy instruction in their code to copy date to or from the graphics card, it would appear that utilizing the standard C memcpy command created the large number of store stalls seen in the processor. This observation is supported by the fact that the number of store unit stalls scaled with the size of the image being copied. Based on these findings, it is extremely inefficient to tie up a single thread to copy data when DMA seems to provide a more efficient means of moving data. The fact that the OpenGL version that computes the error on the CPU was the fastest version of the code was astounding considering the speed of the comparably coded Direct3D version.

In addition, the lack of transparency with objects created in either OpenGL or CUDA created an inefficient means of moving data between the two libraries that utilize graphics memory. Requiring the OpenGL image to detour through system memory when it is copied to CUDA controlled memory is very inefficient when both the source and destination points reside in memory on the graphics card. In addition, the OpenGL interoperability functions in CUDA proved to be less efficient than explicitly handling the data copy. nVidia needs to drastically improve this aspect of CUDA for any type of graphics library interoperability to be viable.

6.3 Future Work

The data from the performance counters point to the 3D mesh alterations as the next slowest function within the encoder. With the apparent parallelism in changing the location of over 3000 individual vertices, this section would be the next target to move to the GPU. Since CUDA supports the creation of OpenGL vertex information, CUDA could provide an additional speedup to the encoding process if the problems discovered earlier do not overshadow the potential improvements.

In addition, several other general purpose GPU based API have been introduced to make GPU more accessible. During the testing process, nVidia released version 3.0 of their CUDA toolbox [26], allowing CUDA to directly access objects created within OpenGL and Direct3D. Many of these changes were mirrored from the Khronos Group's Open Computing Language (OpenCL) [27] which allows programmers to create code that will run on any supported processors, including CPUs and GPUs. nVidia has already released drivers and libraries to allow OpenCL code to run on their CUDA enabled video cards. Microsoft also added the ability to execute arbitrary code inside of their DirectX 11 framework, calling their API DirectCompute [28]. However, this framework requires Window 7 since DirectX 11 is only available for that version of Windows.

REFERENCES

[1] J. Palmer, "The Intel® 8087 numeric data processor," presented at the Proceedings of the 7th annual symposium on Computer Architecture, La Baule, United States, 1980.

[2] IBM. March 5). *IBM PCI Cryptographic Accelerator*. Available: http://www-03.ibm.com/security/cryptocards/pcica/overview.shtml

[3] P. Eisert, *et al.*, "Model-aided coding: a new approach to incorporate facial animation into motion-compensated video coding," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 10, pp. 344-358, 2000.

[4] J. D. Foley, *et al.*, *Computer graphics: principles and practice (2nd ed.)*: Addison-Wesley Longman Publishing Co., Inc., 1990.

[5] M. Doggett, "Programmability Features of Graphics Hardware," 2002.

[6] A. Purde, *et al.*, "Pixel shader based real-time image processing for surface metrology," in *Instrumentation and Measurement Technology Conference, 2004. IMTC 04. Proceedings of the 21st IEEE*, 2004, pp. 1116-1119 Vol.2.

[7] W. F. Engel, *Direct3d Shaderx: Vertex and Pixel Shader Tips and Tricks with Cdrom*. Plano, TX: Wordware Publishing Inc., 2002.

[8] W. Engel, *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*. Plano, TX: Wordware Publishing Inc., 2003.

[9] D. Luebke, "CUDA: Scalable parallel programming for high-performance scientific computing," in *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, 2008, pp. 836-838.

[10] J. Nickolls, *et al.*, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, pp. 40-53, 2008.

[11] Y. Zhiyi, *et al.*, "Parallel Image Processing Based on CUDA," in *Computer Science and Software Engineering, 2008 International Conference on*, 2008, pp. 198-201.

[12] C. Wei-Nien and H. Hsueh-Ming, "H.264/AVC motion estimation implmentation on Compute Unified Device Architecture (CUDA)," in *Multimedia and Expo, 2008 IEEE International Conference on*, 2008, pp. 697-700.

[13] L. Changxin, *et al.*, "Efficient implementation for MD5-RC4 encryption using GPU with CUDA," in *Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on*, 2009, pp. 167-170.

[14] R. Amorim, *et al.*, "Comparing CUDA and OpenGL implementations for a Jacobi iteration," Universität Graz, Graz, Austria, Technical Report SFB 2008-025, 2008.

[15] O. A. R. Board, *OpenGL(R) Reference Manual: The Official Reference Document to OpenGL, Version 1.2 (3rd Edition)*. Boston: Addison Wesley, 2000.

[16] K. Group, "ARB_pixel_buffer_object," vol. ARB Extension #42, ed, 2004.

[17] K. Group. April 14). *GLUT - The OpenGL Utility Toolkit*. Available: http://www.opengl.org/resources/libraries/glut/
[18] April 14). *GLEW: The OpenGL Extension Wrangler Library*. Available: http://glew.sourceforge.net/
[19] NVIDIA. (2009, CUDA Programming Guide 2.3. Available: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf
[20] F. Lavagetto and R. Pockaj, "An efficient use of MPEG-4 FAP interpolation for facial animation at 70 bits/frame," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 11, pp. 1085-1097, 2001.
[21] *MPEG-4 Facial Animation: The Standard, Implementation and Applications*: John Wiley & Sons, Inc., 2003.
[22] I. S. Pandzic, "Facial motion cloning," *Graphical Models*, vol. 65, pp. 385-404, 2003.
[23] Intel. March 5). *Intel® VTune - Intel® Software Network*. Available: http://software.intel.com/en-us/intel-vtune/
[24] March 5). *OProfile: A System Profiler for Linux*. Available: http://oprofile.sourceforge.net
[25] Intel. April 15). *Pin - A Dynamic Binary Instrumentation Tool*. Available: http://www.pintool.org/
[26] March 29). *CUDA 3.0 Downloads*. Available: http://developer.nvidia.com/object/cuda_3_0_downloads.html
[27] K. Group. March 5). *The Khronos Group: Open Standards, Royalty Free, Dynamic Media Technologies*. Available: http://www.khronos.org/
[28] March 29). *DirectCompute*. Available: http://www.nvidia.com/object/cuda_directcompute.html

APPENDIX A

MSE code – CPU based C++

```
for(int i = 0; i < NUM_SUBPIXELS; i++)
{
  temp  = src_face[i] - dst_face[i];
  MSE  += temp*temp;
}
```

MSE code – Direct3D HLSL Pixel Shader

```
// Pixel shader input structure
struct PS_INPUT
{
    float4 Position   : POSITION;
    float2 Texture    : TEXCOORD0;
};


// Pixel shader output structure
struct PS_OUTPUT
{
    float4 Color   : COLOR0;
};


// Global variables
sampler2D Tex0;
sampler2D Tex1;


// Name: MSE Pixel Shader
// Type: Pixel shader
// Desc: Calculates the mean square error between two pixel from the
//       two texture samplers and returns the error as the color for
//       that pixel.
//       R = MSE result;
//       G,B = 0;
//       Alpha = 1;
//
```

```
PS_OUTPUT ps_main( in PS_INPUT In )
{
    PS_OUTPUT Out;
    Out.Color =  tex2D(Tex0, In.Texture);
    Out.Color -= tex2D(Tex1, In.Texture);
    Out.Color *= Out.Color;

    Out.Color.a = 1;
    Out.Color.r = Out.Color.r + Out.Color.g + Out.Color.b;
    Out.Color.gb = 0;

    return Out;
}
```

MSE code - CUDA

```
/*****************************************************************
*  MSE.cu
*****************************************************************/

#define BLOCK_SIZE_1 96
#define BLOCK_SIZE_2 48
#define MAX_BLOCK_SIZE 512

union pixel
{
  unsigned int iVal;
  unsigned char cVal[4];
};


__global__ static void ComputeMSE1(unsigned int* reference,
                                   unsigned int* guess,
                                   unsigned int num_pixels,
                                   unsigned int* result)
{
  int temp;
  unsigned int x = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;
  unsigned int resultSum = 0;
  pixel ref_pixel, guess_pixel;

  __shared__ unsigned int sharedResult[MAX_BLOCK_SIZE];

  if (x < num_pixels)
  {
    ref_pixel.iVal   = reference[x];
    guess_pixel.iVal = guess[x];

    temp = ref_pixel.cVal[0] - guess_pixel.cVal[0]; // Blue
    resultSum  = __mul24(temp,temp);

    temp = ref_pixel.cVal[1] - guess_pixel.cVal[1]; // Green
    resultSum += __mul24(temp,temp);
```

```
    temp = ref_pixel.cVal[2] - guess_pixel.cVal[2]; // Red
    resultSum += __mul24(temp,temp);
  }

  sharedResult[threadIdx.x] = resultSum;

  __syncthreads();

  unsigned int a = blockDim.x;

  while (a & 0x00000001 == 0)  // While even (divisible by 2)
  {
    a >>= 1;

    if (threadIdx.x < a)  // Parallelized Summation
      sharedResult[threadIdx.x] += sharedResult[threadIdx.x + a];

    __syncthreads();
  }

  if (threadIdx.x == 0)  // Perform serial summation for the rest
  {
    resultSum = 0;

    for (unsigned int i = 0; i < a; i++)
      resultSum += sharedResult[i];

    result[blockIdx.x] = resultSum;
  }
}

__global__ static void ComputeMSE2(unsigned int* result,
                                   unsigned int numberOfsharedResults)
{
  unsigned int x = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;
  __shared__ unsigned int sharedResult[MAX_BLOCK_SIZE];

  if (x < numberOfsharedResults)
    sharedResult[threadIdx.x] = result[x];
  else
    sharedResult[threadIdx.x] = 0;

  __syncthreads();

  unsigned int a = blockDim.x;

  while (a & 0x00000001 == 0) // While even (divisible by 2)
  {
    a >>= 1;

    if (threadIdx.x < a) // Parallelized Summation
      sharedResult[threadIdx.x] += sharedResult[threadIdx.x + a];

    __syncthreads();
  }

  if (threadIdx.x == 0) // Perform serial summation for the rest
```

```
  {
    unsigned int resultSum = 0;

    for (unsigned int i = 0; i < a; i++)
      resultSum += sharedResult[i];

    result[blockIdx.x] = resultSum;
  }
}

__global__ static void ComputeMSE3(unsigned int* result,
                                   unsigned int* final)
{
  __shared__ unsigned int sharedResult[MAX_BLOCK_SIZE];

  sharedResult[threadIdx.x] = result[threadIdx.x];

  __syncthreads();

  unsigned int a = blockDim.x;

  while (a & 0x00000001 == 0) // While even (divisible by 2)
  {
    a >>= 1;
    if (threadIdx.x < a) // Parallelized Summation
      sharedResult[threadIdx.x] += sharedResult[threadIdx.x + a];

    __syncthreads();
  }

  if (threadIdx.x == 0) // Perform serial summation for the rest
  {
    unsigned int resultSum = 0;

    for (unsigned int i = 0; i < a; i++)
      resultSum += sharedResult[i];

    *final = resultSum;
  }
}

extern "C" void launch_kernel(unsigned int* reference,
                              unsigned int* guess,
                              unsigned int num_pixels,
                              unsigned int* result,
                              unsigned int* final,
                              int blockSize1,
                              int blockSize2)
{
    // execute the kernel
    int threadsPerBlock = blockSize1;
    int blocksPerGrid = (num_pixels + threadsPerBlock - 1) /
                         threadsPerBlock;

    ComputeMSE1<<< blocksPerGrid, threadsPerBlock >>>(reference,
                                                      guess,
                                                      num_pixels,
```

```
                                                    result);

    while (blocksPerGrid > MAX_BLOCK_SIZE)
    {
      unsigned int numberOfsharedResults = blocksPerGrid;
      threadsPerBlock = blockSize2;
      blocksPerGrid = (blocksPerGrid + threadsPerBlock - 1) /
                       threadsPerBlock;
      ComputeMSE2<<< blocksPerGrid, threadsPerBlock >>>(result,
                                                numberOfsharedResults);
    }

    ComputeMSE3<<< 1, blocksPerGrid >>>(result, final);
}
```

VITA

Brian Patrick Gordon

Candidate for the Degree of

Master of Science

Thesis: GPGPU PROGRAMMING AND THE PITFALLS OF NAIVE DATA HANDLING

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Amarillo, TX on August, 13 1984, the son of Bruce and Joan Gordon.

Education: Received a Bachelor of Science in Electrical Engineering from Oklahoma State University, Stillwater, Oklahoma in December 2007. Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2010.

Experience: Employed by Oklahoma State University, School of Electrical and Computer Engineering, as a research assistant and a teaching assistant, 2007-Present.

Name: Brian Patrick Gordon Date of Degree: May, 2010

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: GPGPU PROGRAMMING AND THE PITFALLS OF NAIVE DATA HANDLING

Pages in Study: 48 Candidate for the Degree of Master of Science

Major Field: Electrical Engineering

Scope and Method of Study: The purpose of this study was to implement a known bottleneck of a model based video encoder in code that runs on a modern graphics card using various methods. The study then looks the degree of speedup as well as the new bottlenecks for each implementation. The study also compares and contrasts the effectiveness of each method at improving the performance of the encoder.

Findings and Conclusions: While moving the highly parallel error calculation to the graphics card reduced the time required to perform the computation, the program still ran slower. Through analysis of the processors performance counters, the image data appears to move through system memory even though its origin and destination are in graphics memory. This inefficient handling of the data that should remain in graphics memory is removing any potential speedup from using the graphics card as a highly parallel coprocessor to accelerate the slowest part of the original program.

ADVISER'S APPROVAL: Dr. Sohum Sohoni