ENABLING NEW USES FOR GPUS

by

Matthew D. Sinclair

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Science

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2011

To my Dad, who isn’t around to see this day. I know you’re looking down on me and smiling.

To my Mom, for always supporting me.

To Ann, for continually accepting the excuse that I was in the lab and sticking by my side.

ACKNOWLEDGMENTS

Let me preface this by saying that there are many people who have helped me reach this point and I will never be able to thank all of them. For those who have helped me and are not acknowledged here, my sincere apologies.

First, I must thank my adviser Karu Sankaralingam. Three years ago Karu took a chance on a kid with no research experience. Then, even after my first couple research projects didn't pan out, he continued to believe in me and provide me with opportunities to learn and grow. Kare has been instrumental in molding me into the researcher and person I am today. Simply put, I would not be where I am now without his help.

I must also acknowledge and thank Richard Townsend, my co-adviser from the Astronomy Department. Rich has also provided me with many learning opportunities, especially involving CUDA, and placed his faith in me to work on GRASSY. He has also repeatedly allowed me to use his GPUs for experiments, even when I wasn't explicitly running experiments for GRASSY. For this, I am forever grateful. Scheming to win the Tesla C2050 GPU at GTC will always be a highlight of my Masters!

The staff here at Wisconsin, both in the ECE and CS Departments, has been top notch throughout my six years here. I would especially like to thank Tim Czerwonka from the CSL. Tim has repeatedly helped me out with getting the proper tools and equipments installed on the machines I needed them on, in an extremely timely fashion. Whenever something went wrong or went down, Tim was there to lend a hand, and I can't think him enough for his help.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

LIST OF TABLES v

LIST OF FIGURES vi

ABSTRACT vii

1 Introduction 1

2 Understanding GPU Texture Memory Capabilities 6

2.1 Texture Memory Background 7
2.2 Related Work 9
2.3 Texture Memory Performance Analysis 10
2.3.1 Microbenchmark Construction 11
2.3.2 Experimental Setup 12
2.3.3 Results 16
2.4 Conclusions 19

3 Utilizing Texture Memory: GRASSY 20

3.1 GRASSY Background and Problem Statement 21
3.2 Porting the Code from the CPU to the GPU 26
3.2.1 Texture Packing 27
3.2.2 Interpolation Decomposition 27
3.2.3 Division of Labor 28
3.2.4 Pseudo Code 28
3.2.5 Optimizations 30
3.3 Evaluation 31
3.3.1 Methodology 31
3.3.2 Results 31
3.4 Related Work 34

Page

3.5 Mathematical Analysis . . . 34
3.6 Summary . . . 36

4 Porting CMP Benchmarks to GPUs . . . 38

4.1 Background and Related Work . . . 39
4.2 GPU Implementations . . . 42
4.2.1 Streamcluster . . . 42
4.2.2 Blackscholes . . . 43
4.2.3 Fluidanimate . . . 44
4.2.4 Swaptions . . . 44
4.3 Methodology . . . 45
4.3.1 Verification and Performance Testing . . . 45
4.3.2 System Specifications . . . 46
4.4 Results and Analysis . . . 46
4.4.1 Streamcluster . . . 47
4.4.2 Blackscholes . . . 48
4.4.3 Fluidanimate . . . 51
4.4.4 Swaptions . . . 53
4.5 Summary . . . 54

5 Designing a Suite of Challenging Benchmarks for GPUs . . . 57

5.1 Benchmarks for Future GPU Architectural Research . . . 57
5.2 Analyzing Challenging GPU Benchmarks . . . 59
5.2.1 Overview . . . 59
5.2.2 Characterization . . . 61
5.2.3 Data Analysis . . . 61
5.2.4 Limitations . . . 63
5.3 Summary . . . 63

6 Conclusion . . . 65

LIST OF REFERENCES . . . 67

DISCARD THIS PAGE

LIST OF TABLES

Table Page

2.1 Systems Used Specifications (Systems are used throughout the paper). 13

3.1 Definitions of quad variables. 24

4.1 Background information on implemented PARSEC CMP benchmarks. 41

5.1 Benchmark effective IPC (challenge benchmarks shaded) 58

5.2 Detailed challenge benchmark analysis (kernels in numeric-alpha order, except NNW, which is in layer order). 60

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

2.1 GPU (Tesla) Architecture, reproduced from [37]. 7

2.2 Texture Processing Cluster (TPC), reproduced from [37]. 8

3.1 Pictorial representation of analyzing stars with direct modeling. Further details are available in [46]. 21

3.2 Discretizing a pulsating star. 22

3.3 Steps in spectral synthesis process. 23

3.4 Schematic comparison of CPU and GPU spectral synthesis. 26

3.5 GRASSY GPU results vs. serial and parallel CPU implementations. 32

3.6 GRASSY GPU results for several different GPUs vs. CPU implementations. . . . 33

3.7 GRASSY GPU results for same age components as CPU. 33

4.1 Streamcluster GPU speedups over serial CPU implementation. 47

4.2 Blackscholes GPU speedups over serial CPU implementation. 49

4.3 Fluidanimate GPU speedups over serial CPU implementation. 51

4.4 Fluidanimate GPU speedups versus serial CPU implementation for a varying number of GPU kernels. 52

4.5 Swaptions GPU speedups versus serial CPU implementation. 54

ABSTRACT

As graphics processing unit (GPU) architects have made their pipelines more programmable in recent years, GPUs have become increasingly general-purpose. As a result, more and more general-purpose, non-graphics applications are being ported to GPUs. Past work has focused on applications that map well to the data parallel GPU programming model. These applications are usually embarrassingly parallel and/or heavily utilize GPU architectural features such as shared memory and transcendental hardware units. However other GPU architecture components such as texture memory and its internal interpolation feature have been underutilized. Additionally, past work has not explored porting CMP benchmarks to GPUs; if GPUs are truly becoming a general-purpose architecture, they need to be able to execute general-purpose programs like CMP benchmarks, especially programs that do not map well to the data parallel paradigm, with high performance. This thesis focuses on enabling these new uses for GPUs by implementing new use applications on GPUs and then examining their performance. For those benchmarks that do not perform well, we explore what bottlenecks still remain that prevent them from obtaining high performance.

Chapter 1

Introduction

As graphics processing unit (GPU) architects have made their pipelines more programmable in recent years, GPUs have become increasingly general-purpose. As a result, a wider range of programmers have been able to utilize GPUs to gain significant performance increases over CPU implementations [38]. Programmers are attracted to implement their applications on GPUs because GPUs offer tremendous potential speedups over CPU implementations. GPUs offer large speedups because they are massively parallel and allow many computations to be performed simultaneously. However, programs usually need to map well to the GPU programming model to effectively achieve these speedups.

There are many examples of porting non-graphics applications to GPUs and obtaining impressive speedups over CPU implementations. For example, Yang et. al. report speedups of 80x for image histograms and more than 200x for edge detection [58]. Genovese reports speedups of 10x to 60x for 1D convolutions [20]. Govindaraju et. al. report speedups of 8x to 40x over optimized CPU implementations for discrete Fourier Transforms [23]. Silberstein et. al. report speedups of 2700x for random data and 270x for real-life data for sum-products [48]. Tolke and Krafcyzk see two orders of magnitude speedup for a 3D Lattice Boltzmann (LBM) CFD application [53]. Smith et. al. and Han et. al. port networking applications to the GPU and see speedups of 9x over serial (2.3x over parallel) CPU implementations [49] and 4x over optimized CPU implementations [27], respectively. Townsend also reports speedups of 205x over serial CPU implementations and 25x over parallel CPU implementations for CU-LSP, a Lomb-Scargle periodogram (LSP) for spectral analysis code for scientific applications [51].

Finally, Xu and Mueller see very large speedups for a bilinear filtering algorithm for iterative CT scanning on GPUs [57].

In general, programmers have been able to obtain high performance on GPUs by porting applications that map well to the GPU's data parallel, single instruction multiple data (SIMD) programming model. These applications are often embarrassingly parallel and do not have control flow issues. Additionally, programmers have regularly utilized GPU architectural features such as shared memory and transcendental hardware units for specialized mathematical operations to increase performance. The example applications from the previous paragraph have these features: Han, Genovese, and Smith's applications exploits the massive parallelism available on the GPU. Govindaraju, Silberstein, Tolke, and Yang's applications are highly parallel and use shared memory. Townsend's CU-LSP application uses transcendental hardware units for sine and cosine calculations, and Xu's bilat also benefits from the fast transcendental units, such as exponents, available on the GPU [35]. Clearly, these applications map well to the GPU and use its computational resources to obtain high performance. These applications also represent uses of the GPU that became easier to implement after GPU pipelines became more general-purpose.

The goal of this thesis is to identify further new uses for GPUs, especially new uses that can use underutilized components of the GPU architecture and applications that were previously unsuited for GPU execution; based on these results we identify some new uses for GPUs. Finally, we explore what applications remain as challenging benchmarks for GPUs.

Thesis Organization

One feature of the GPU that has not been explored for non-graphics use is texture memory interpolation. A few of the previous examples [48, 49, 57] explore using texture memory, but none of these applications exploit the full computational power of textures, which can perform interpolations internally for "free" in hardware. GPUTeraSort [22] and PSBN [24] are further examples of applications that uses texture memory for caching, but don't use the internal interpolation feature of texture memory. In Chapter 2, we explore the theoretical maximum

obtainable performance when using texture memory by analyzing the texture unit and find that algorithms which heavily utilize the internal interpolation of the texture memory can obtain large speedups. Our key insight is that the texture memory can be used as an interpolation engine for non-graphics applications, freeing the GPU cores from computation work. Based on these results, we implement GRASSY, an asteroseismology application that heavily utilizes textures to perform nearest neighbor interpolation in tables of pre-calculated intensity data. By mapping GRASSY's pre-calculated intensity data to the GPU texture memory, we are able to obtain impressive speedups over optimized CPU implementations. GRASSY is representative of algorithms that can benefit significantly from this new use of GPUs. Chapter 3 describes our implementation of GRASSY and our study of the upper and lower bounds of texture memory interpolation performance. We show that an application needs to be dominated by interpolation to prevent the texture memory from rapidly becoming a bottleneck, a problem that is exacerbated by trends in newer GPUs.

While many scientific workloads, such as CU-LSP and bilat, have been ported to GPUs and found to achieve impressive speedups, these applications fit the GPU paradigm extremely well. However, it isn't obvious if other workloads that do not map well to the SIMD paradigm can also obtain high performance on GPUs. In the second direction of this thesis, we look at how well conventional CMP benchmarks can execute on a GPU. For the GPU to become a truly general-purpose architecture, it must be able to execute general-purpose CMP applications with high performance. In Chapter 4, we describe a study on porting four CMP benchmarks from the PARSEC benchmark suite to GPUs to explore how their general-purpose programming constructs perform on GPUs. With the exception of blackscholes, these benchmarks do not fit the SIMD paradigm that GPUs historically require. This is reflected in the poor performance we see for these benchmarks. In general, we found that the PARSEC CMP benchmarks do not port very well to GPUs. However, understanding why these benchmarks perform poorly allows us to understand if there are fundamental bottlenecks in the GPU architecture that prevent benchmarks like these from obtaining high performance.

In Chapter 5, we further examine the bottlenecks in our CMP benchmarks and extend our findings from Chapter 4 across many benchmark suites. We create a suite of challenging benchmarks for GPUs. We also explore what the bottlenecks to obtaining high performance for these benchmarks on GPUs are and classify all the benchmarks under the broad categories of insufficient available parallelism, poor control flow, and memory access contention. By identifying a suite of challenging benchmarks, we enable architects to explore architectural changes to future generations of GPUs.

All our results are based on measurements taken on real GPUs; we describe the systems used in Subsection 2.3.2. Additionally, throughout this thesis, all of our implementations use CUDA, Nvidia's proprietary GPU programming language [1, 2]. We use CUDA because it is a common GPU programming language. Limitations of the studies in this thesis include potentially unoptimized algorithms in our benchmarks and features that have changed in newer releases of GPU programming environments that which we did not retest for. Throughout the thesis, we note places in our implementations that could potentially be improved by using newer generations of the programming environments.

Thesis Contributions

The contributions of this thesis are:

- We explore the theoretical maximum obtainable performance when using the internal interpolation capability of texture memory by analyzing the texture unit.
- We explore the use of texture memory interpolation in non-graphics GPU programs by implementing GRASSY, an asteroseismology program that heavily utilizes texture memory interpolation to achieve large speedups.
- We implement a series of general-purpose CMP benchmarks on the GPU to examine the performance of non-data parallel programs on the increasingly programmable pipelines of GPUs.

- We identify and analyze a set of challenging benchmarks that perform poorly on current GPUs.

Overall, we find that several new uses for the GPU, especially for applications that utilize the internal interpolation of texture memory. These uses are enabled by the increasingly general-purpose nature of the GPU. For the applications that do not perform well, we identify the sources of their bottlenecks, as well as what changes to the GPU architecture can be made to enable these programs to execute efficiently.

Chapter 2

Understanding GPU Texture Memory Capabilities

Texture memory is an important component of the GPU architecture that is commonly used in graphics. However, it has been used sparingly in non-graphics GPU applications. When the texture unit has been used in non-graphics applications, it has often been used as a cache to offset the lack of a direct cache in pre-Fermi Nvidia GPUs, although a few applications, predominantly in the financial industry, have used its internal interpolation. Texture memory has the ability to perform nearest neighbor interpolation internally, which provides an efficient and untapped source of computational power.

If we can find a way to properly harness the texture memory's internal interpolation, then non-graphics applications can benefit from its use and obtain improved performance. In this chapter, we seek to examine how we can more effectively utilize texture memory and its internal interpolation feature in non-graphics and general-purpose GPU applications. To do this, we examine the tradeoffs to using texture memory interpolation by microbenchmarking the performance of the texture unit. We find that applications that heavily utilize the internal interpolation feature of texture memory are ideal candidates for using texture memory interpolation.

In the rest of this chapter, we first provide background information on the texture unit in Section 2.1, then we address the non-graphics, general-purpose applications that have already used texture memory in Section 2.2. Next, in Section 2.3 we analyze the texture memory through microbenchmarking. Finally, we conclude in Section 2.4.

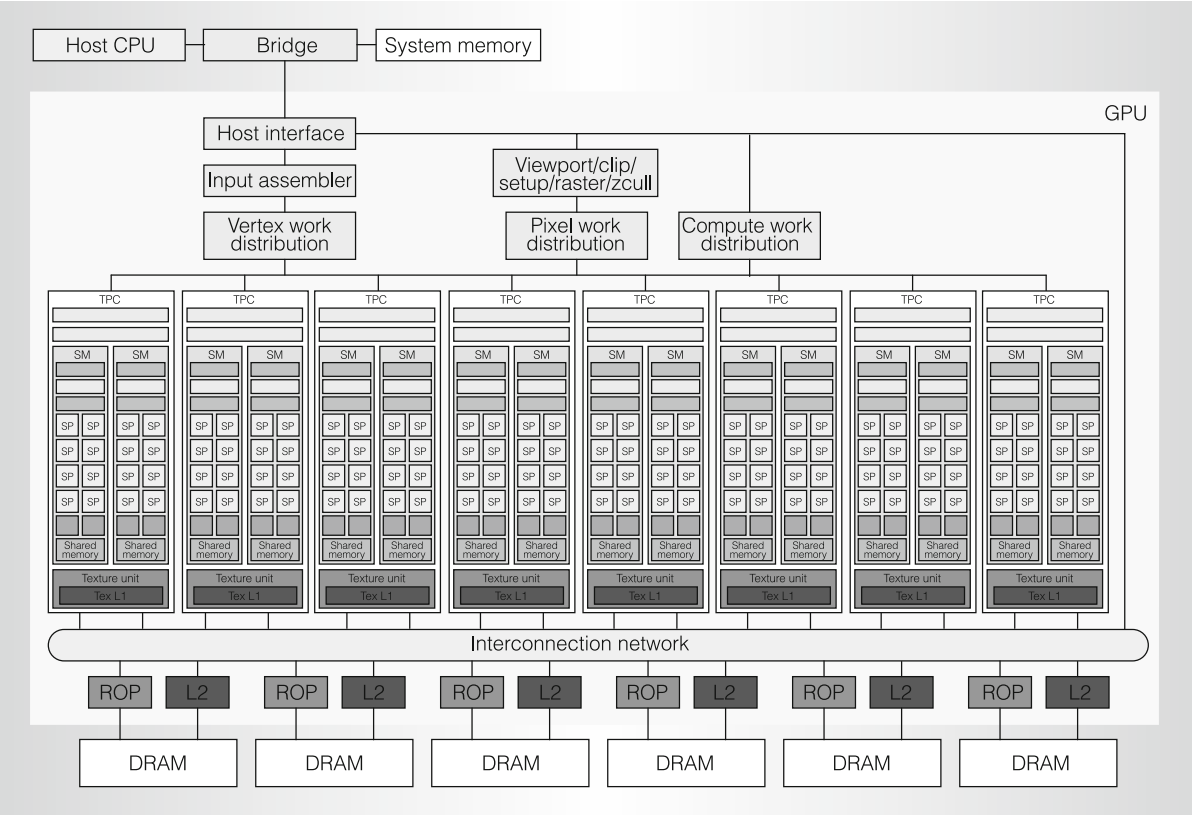


Figure 2.1: GPU (Tesla) Architecture, reproduced from [37].

2.1 Texture Memory Background

In this thesis, we focus on Nvidia GPUs. Figure 2.1 is high-level schematic of the Nvidia Tesla GPU architecture. We refer the reader elsewhere for more details on how specific components in the figure operate [37]. Most importantly, the figure shows that the GPU is a massively parallel architecture that is designed to have many threads operating on data simultaneously. The architecture is broken into Texture Processing Clusters (TPCs), an example of which is shown in Figure 2.2. TPCs contain several Streaming Multiprocessors (SMs), as well as two Special Functional Units (SFUs) which are responsible for performing transcendental calculations like sine and cosine in hardware, a local texture cache and local shared memory. Each SM contains eight Streaming Processors (SPs), which are ALU-like units. In the example in

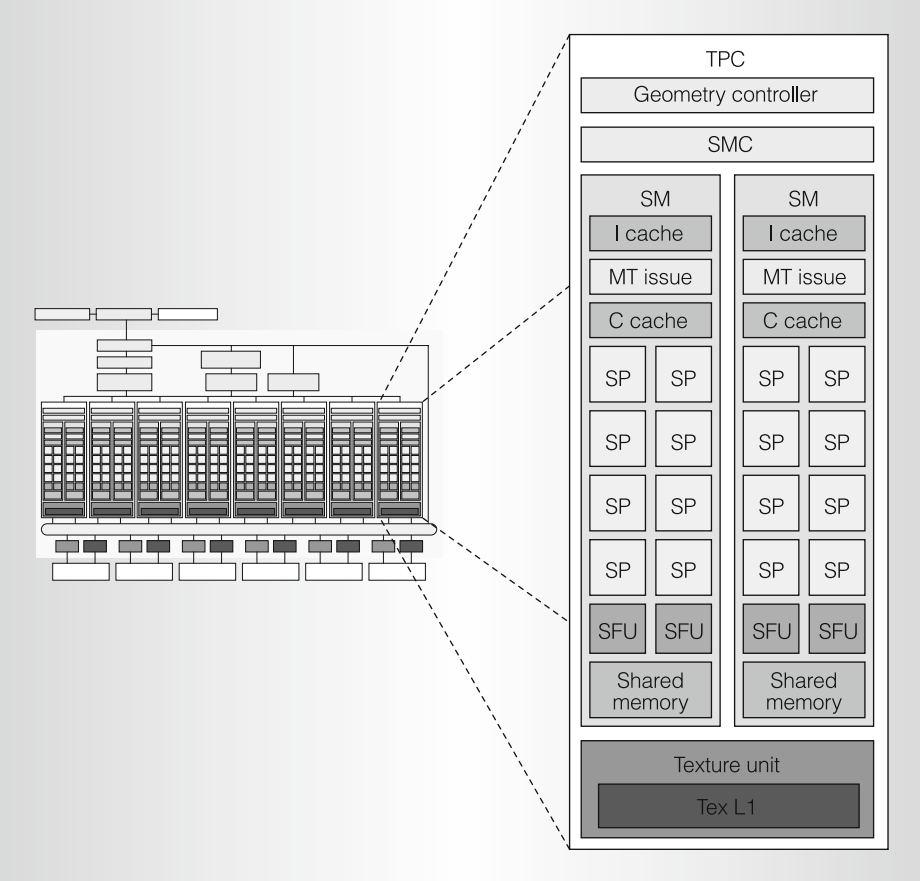


Figure 2.2: Texture Processing Cluster (TPC), reproduced from [37].

Figure 2.2, the TPC contains two SMs[1], which share the local texture cache. All of the TPCs are linked together via an interconnect, which also connects the TPCs to global memory.

The global memory contains the texture memory reference when it is being used. The texture memory layout, including the caches on the TPCs, is optimized for 2D locality. Also, it does not reduce DRAM fetch latency in the GPU [2]. Instead, texture memory is designed for streaming fetches with a constant latency. Thus, hitting in the texture cache will artificially increase the number of texture requests (and thus the bandwidth) the texture memory can simultaneously service, because those requests won't need to be serviced by the global texture memory (and therefore other requests can be). Another unique feature of texture memory is

[1]Different generations of Nvidia hardware have different numbers of SMs per TPC.

that it performs interpolation internally in hardware. If the user requests a data point that is in between the data points in the texture memory, it will perform nearest neighbor interpolation[2] between the data points in texture memory that are closest to the inputted data point and return the interpolated value. Because this interpolation is done internally in hardware, we can consider it to be done for "free" as compared to performing these calculations using the GPUs SPs or SFUs. Finally, texture memory is read-only during a GPU kernel's execution. Further information on textures can be found elsewhere [25, 40, 31, 16].

To make texture memory visible to the programmer, we first need to declare a texture reference. The texture reference declares a region of the GPU's global memory to be texture memory. After loading data into it on the host, we can access it in our GPU code by using the texture reference we initially declared, along with the coordinates we want to read from the texture unit. To access a texture, CUDA uses a standard, opaque routine that returns a float (called a texel) with the value from the desired coordinates, interpolated if necessary.

However, there are also some challenges to using textures in CUDA. For example, while textures return a float, the interpolated results are only guaranteed to be 9 bit precise values due to the precision of the internal interpolation. In [52] we show that this is not a concern for our GRASSY platform, but it could be an issue for other applications that require more than 9 bits of precision. Additionally, while CUDA supports 3D textures, their size is severely limited, as is specified in Appendix G.1 of the CUDA programming guide [2]. Finally, CUDA does not support textures with more than three dimensions.

2.2 Related Work

There has been some past work on using texture memory for non-graphics applications, which we discuss here. This work generally falls into one of two categories: applications that use texture memory for caching (i.e. as a lookup table) and applications that use texture memory interpolation.

[2] Linear, bilinear, or trilinear interpolation depending on if 1D, 2D, or 3D textures are being used, respectively.

Many applications have used texture memory as a cache/lookup table to obtain speedups. Because pre-Fermi Nvidia GPUs did not have a true caching scheme, using textures to cache data allowed data to be placed much closer to the cores and increase performance, including some of the examples mentioned in Chapter 1 [22, 24, 48, 49, 57]. While all of these applications use texture memory for caching, none of these applications use the internal interpolation feature of texture memory. GPUTeraSort and PSBN were created before GPU pipelines became more programmable when all data elements were treated as texture texels. Thus, textures were not used in the same ways that we use them. Additionally, Sobel Filter from the CUDA SDK [2] and Li's Lattice Boltzmann computation implementation [36] utilize textures for lookup table purposes: SobelFilter uses OpenGL texture memory for thresholding and lookups and Li's implementation stores packets for its LBM as textures to perform fast lookups.

There have also been several applications, mostly in the financial industry, that utilize the free internal interpolations that texture memory can provide. This work is most similar to the work we explore in this chapter and Chapter 3. Bennemann et. al. implement pricing algorithms on the GPU and speed up their local volatility models by using bilinear interpolation in the texture unit [4]. This technique has also been used by others [5, 6]. Nord uses textures to speedup calculations for capped basket options [42]. GJK, is a rigid body detection algorithm that is commonly used for collision detection and resolution of convex objects [35]. In [47], they map the GJK support maps to texture units and use it both as a lookup table and use it to interpolate the distances of a vertex from a center. In general, while some previous work has used texture memory interpolation, our study provides application-independent insight into when using texture memory interpolation is useful.

2.3 Texture Memory Performance Analysis

In this section, we explore the behavior and performance capabilities of texture memory. To do this, we use microbenchmarking, which draws inspiration from GPU texture microbenchmarks from other domains [13, 55]. Our goal in these experiments is to characterize the behavior of the texture unit of the GPU. To do this, we ran our experiments on a Nvidia Quadro FX 580 GPU and a Nvidia Tesla C1060 which, as stated in Table 2.1, are capable of sustaining a maximum of 16 texture lookups/cycle and 80 texture lookups/cycle. The results from these experiments will allow us to better understand how to optimally use texture memory in order to obtain higher performance. The metric used in all the experiments is **throughput (lookups/cycle)** of the texture unit.

2.3.1 Microbenchmark Construction

We constructed a parametrized microbenchmark and varied different access parameters in order to understand the behavior of the texture unit and characterize its performance. By varying these parameters, we expect to see how different inputs affect the texture unit's throughput. The input parameters we varied are:

- stride between texture unit accesses
- number of threads per thread block
- number of threads blocks used

Varying the stride between texture cache accesses shows how the texture unit performs for accesses with different distances between them. These three parameters are important because they affect texture memory performance by causing increased competition for resources, affecting locality of accesses, and allowing for better hiding of misses. For example, a stride that causes all accesses to be in the same bank of the texture cache will perform poorly because it will cause thrashing. Varying the number of threads per block affects the performance of the

```
1  texture <float, ...> texRef; // Preloaded with randomized values
2  __global__ void accessTextCache(float * outArr, int stride)
3  {
4      // local variables
5      int i = 0;
6      float retVal = 0.0f;
7      // perform lookups 500K times
8      for (i = 0; i < 500,000; ++i)
9      {
10         // Note: thread_index differs for each thread
11         retVal += tex1D(texRef, thread_index * stride);
12     }
13     // write retVal to an output array to prevent the compiler from optimizing
14     // out the texture lookups (could also use a volatile variable)
15     outArr[thread_index] = retVal;
16 }
```

Listing 2.1: Sample Code for 1D Texture Microbenchmarking

texture unit. When there are more threads per block, there are more opportunities to hide texture cache misses, but there may also be more capacity misses because of the increased number of threads. Finally, having more blocks of threads should better utilize all the SMs on the GPU, which should allow better hiding of texture cache misses. However, having too many blocks may cause congestion. Varying these parameters should help us find an ideal number of thread blocks and threads per thread block to obtain high performance.

All the experiments were designed to be texture unit limited, so that the performance would not be significantly affected by other parts of the GPU architecture. Listing 2.1 shows an example of how a 1D texture microbenchmark might look (with some bookkeeping omitted). The kernel accesses the texture unit repeatedly in an unrolled loop[3] and the total execution time of the kernel is used to calculate the throughput.

[3]Loop unrolling ends up not contributing to our results, because the compiler doesn't unroll the loop due to excessive code expansion. Thus, we exclude the loop unroll pragma from our sample code to avoid confusion.

Table 2.1: Systems Used Specifications (Systems are used throughout the paper).

	GPUs			
GPU Parameter	*Tesla C1060*	*Quadro FX 580*	*NVS Quadro 295*	*GeForce 8400 GS*
CUDA Capability	1.3	1.1	1.1	1.1
Streaming Multiprocessors (SMs)	30	4	1	1
Streaming Processors per SM (SPs/SM)	8	8	8	8
Max Texture Lookups per cycle	80	16	4.4	3.6
Max # of Registers per thread block	16K	8K	8K	8K
Clock Frequency	1.3 GHz	1.12 GHz	1.3 GHz	1.4 GHz
Memory Bandwidth	102 GB/s	25.6 GB/s	11.2 GB/s	6.4 GB/s
Global Memory	4 GB	512 MB	256 MB	512 MB
Shared Memory per thread block	16 KB	16 KB	16 KB	16 KB
	CPUs			
Attached CPU Parameter	*2 Xeon E5345's*	*Nehalem i5*	*Xeon E5520*	*2 Xeon E5345's*
# Cores	8	4	4	8
Clock Frequency	2.33 GHz	1.2 GHz	2.26 GHz	2.33 GHz

2.3.2 Experimental Setup

Throughout this thesis, we ran our experiments on the systems in Table 2.1. In each chapter and section, where applicable, we will specify which of these systems were used for those specific experiments. In general, a subset of these systems were used for specific experiments because they were not all available or purchased at the time the experiments were run.

For our microbenchmarking experiments, we used the Quadro FX 580 and Tesla C1060 platforms. We chose these GPUs because they support the most texture lookups per cycle of all four GPUs. Additionally, these GPUs have a wide enough disparity in maximum texture lookups per cycle that we should be able to see any discrepancies in performance. We perform a series of three experiments to help characterize the behavior of the texture unit. In these tests,

we vary the parameters we introduced at the start of this section to demonstrate how texture unit performance varies. We performed the following experiments:

1. We use a 1D texture, vary the stride sequentially, and use constant number of thread blocks and threads per thread block.

2. We use a 1D texture and vary the stride, number of thread blocks, and number of threads per thread block by powers of 2.

3. We use a 2D texture, and vary the stride, number of thread blocks, and number of threads per thread block by powers of 2.

The texture data structure we used in this experiment is 4 MB in size for all experiments. We chose this size so the working set would be larger than the cache size; thus making it likely that the data will be evicted from the texture cache between accesses to it, which prevents performance from being artificially inflated by the texture caches containing the entire working set. No threads attempt to access a memory location outside the bounds of the texture data structure.

In order to provide a baseline performance level to calibrate the performance of our various access patterns against, we created a “constant” lookup texture access pattern. In this baseline experiment all threads perform lookups to the same texture location by accessing a constant index. Therefore, only one cache miss per TPC will occur (when the first thread from a TPC tries to access that index). After this miss, the rest of the accesses will be hits in the cache and data will be shared amongst all the TPCs. The constant test provides a ceiling on attainable performance approximately equal to the maximum throughput of the GPU.

In comparison, all three experiments perform a single lookup in the texture unit for each thread, similar to the code shown in Listing 2.1. Each thread accesses a different index in the texture cache, which allows us to determine what the performance of the texture unit is when there is no sharing of texture-cached data between threads, a much more realistic approach. Each thread repeats its lookup 500K times to prevent compulsory misses from dominating the

latency measurements and to provide a good steady-state performance measurement. Because 1D and 2D textures are mapped differently to the texture unit, we expect that they will obtain different performance results. Thus we perform both 1D and 2D texture experiments to examine the difference in performance between 1D and 2D textures.[4]

The sequentially varying stride experiment (Experiment #1) was designed to test a 1D texture along a spectrum of strides ($\in$ [1,128]) to show which distance between accesses would provide maximal performance for a given number of blocks and threads per block. The stride represents the distance between consecutive accesses in the texture unit. For example, for a stride of two, the first thread's index in the texture unit will be location 0 (texel 0), and the second thread's index in the texture unit will be location 2 (texel 2) and so on. The goal of this experiment to see if varying stride between accesses affects performance. We performed this test for two configurations: A) 128 thread blocks with 128 threads per thread block and B) 512 thread blocks with threads per thread block (both configurations have varying strides). We chose these configurations because they represent commonly used configurations in GPU programming. Additionally, both data points have enough thread blocks and threads per block to hide the latency delays caused by misses in the texture cache.

Experiment #2 expands on experiment #1. However, instead of varying only the stride for a 1D texture, we also varied the number of thread blocks and threads per block. The goal of this experiment is to find the optimally performing combination of thread blocks, threads per thread block, and stride between texture unit accesses for a 1D texture. We vary these parameters by powers of two instead of sequentially as we did in experiment #1, because we are more interested in the overall trends in this experiment.

Experiment #3 tests the performance of a 2D texture. Similar to experiment #2, the stride between accesses, number of thread blocks, and number of threads per thread block vary by powers of two. We use multiple access patterns in this experiment. Since we're using a 2D

[4]There are two experiments for the 1D textures and one for 2D textures because the two 1D experiments are a subset of the 2D texture experiment tests.

texture unit now, we gather the results by accessing the texture unit in both the X and Y dimensions. The four access patterns we tested are:

- Striding sequentially in X dimension (a stride of 1), constant value in Y dimension
- Striding sequentially in Y dimension (a stride of 1), constant value in X dimension
- Strides of greater than 1 in both dimensions,
- Strides of greater than 1 in the X dimension while using a constant value in the Y dimension.

Our goal experiment #3 was to find the optimally performing combination of threads per thread block, number of thread blocks, and stride between texture unit accesses for 2D textures. We also wanted to find which of our access patterns provided the best performance for a 2D texture.

2.3.3 Results

The results for the baseline experiment, for both GPUs, always obtain approximately maximum throughput, as expected, and thus are not reported on further.

Quadro FX 580 Results

Experiment 1 (1D Texture, Sequentially varying stride): The results are relatively constant regardless of stride – the maximum variation in throughput for all tested strides is 0.005 texture lookups/cycle, which is so small (0.04%) that it can be ignored. All data points achieved between 14 and 15 texture lookups/cycle, which is very close to the peak of 16 texture lookups/cycle. This shows that stride doesn't significantly affect performance when many thread blocks are used. However, when fewer thread blocks are being used, stride between accesses affects performance more significantly because there are fewer blocks to hide the latency of the misses when data is accessed non-sequentially, a result we confirmed in the other experiments.

Experiment 2 (1D Texture, Varying stride, number of thread blocks, and number of threads per block): In this experiment we observed that we can sustain more than 15 lookups/cycle when we have 32 or more blocks of threads. With 128 thread blocks or more, we can sustain approximately 16 lookups/cycle. This shows that having many thread blocks to hide the latency of texture cache misses is important to attaining high performance. The stride between accesses again had an insignificant effect on the throughput because we have enough threads to hide the latency of misses. When there are fewer thread blocks, stride affects performance more significantly, which confirmed our results from Experiment #1. Overall, we found that there was not a single ideal combination of stride, threads per block, and thread blocks, but rather a range where we could obtain approximately maximal performance.

Experiment 3 (2D Texture, Varying stride, number of thread blocks, and number of threads per block): For a 2D texture, we observed that we can sustain approximately 16 lookups/cycle when 256 thread blocks or more are used. The maximum performance of the 2D texture is slightly worse than that of the 1D texture in the second experiment. In general, the results for sequential accesses, striding in one direction, and striding in both directions are approximately the same. The best results for the 2D texture experiment were also slightly skewed to fewer threads per block than the 1D texture results. It is possible that this occurs because more thrashing happens in the texture cache when we're trying to access elements in multiple dimensions, which means more elements are being accessed.

Tesla C1060 Results

Experiment 1 (1D Texture, Sequentially varying stride): Similar to the results on the Quadro FX 580, in the results obtained for this experiment on the Tesla C1060 we observe that the throughput is relatively constant when varying the stride sequentially and that it does not significantly affect performance when there are numerous thread blocks and threads per block being used. Both data points achieve between 74 and 75 lookups/cycle, which is the same percent of peak that this experiment obtained on the FX 580 (94%). As seen in the Quadro results,

these results also show that the stride affected performance more significantly when few thread blocks and/or threads per block were used.

Experiment 2 (1D Texture, Varying stride, number of thread blocks, and number of threads per block): In these results, we observe that we can sustain approximately 79 lookups/cycle for 256 blocks of threads or more. This again demonstrates the importance of using lots of blocks to hide the latency of misses in the texture cache in order to attain high performance. Additionally, the Tesla is a larger GPU than the Quadro, so it makes sense that it requires more thread blocks than the FX 580 needed to approach its maximum throughput, because more work needs to be done to fill all of the Tesla's hardware resources.

Experiment 3 (2D Texture, Varying stride, number of thread blocks, and number of threads per block): For the 2D texture experiment on the Tesla C1060, we observe that we can sustain more than 78 lookups/cycle when we use 256 blocks or more. Similar to the Quadro performance results, the performance for a 1D texture on the Tesla is slightly higher than the performance for a 2D texture. In general, the results for all access patterns were approximately the same. The results of this experiment confirm that having lots of thread blocks, each with many, but not always the most, threads per block, is key to obtaining near-maximum throughput from the texture unit.

Overall, the results from the Tesla C1060 experiments are extremely similar to the results obtained for the Quadro FX 580. While the GPU's maximum throughputs differ, both GPU's texture units exhibit asymptotic behavior approaching their maximum possible throughputs. Thus, the results for the Tesla confirm the results found for the Quadro FX 580. Additionally, our sustained memory bandwidth is well below the peak memory bandwidth of both GPUs. This shows that we're able to saturate the texture memory and achieve peak throughput without saturating the GPU memory.

Observations

These results of our experiments provide several key takeaways:

- The stride between accesses doesn't significantly affect performance when there are enough thread blocks to hide the latency of texture cache misses. If the accesses were not strided, performance could potentially be even worse, because there could be less locality.

- Stride between accesses **does** affect performance when there are fewer thread blocks, because the latency of texture cache misses cannot be hidden effectively.

- Having many threads blocks and many threads per thread block improves performance by allowing the latency of misses to be better hidden. Having more threads per thread block also potentially occupies more SMs on the GPU and leads to better utilization.

- In some cases, such as with 2D textures, using slightly fewer threads per block than the maximum provides better performance, likely by reducing thrashing and contention for texture cache resources.

2.4 Conclusions

In this chapter, we have demonstrated that tremendous speedups can be obtained in non-graphics applications through the use of texture memory and its internal interpolation. Internal interpolation performs nearest neighbor interpolation in hardware on the GPU, which means these calculations do not need to be explicitly done by the GPU processing cores and instead are provided for free by the texture hardware. Previous work largely ignores the ability to use texture memory interpolation. Thus, the use of texture memory in non-graphics applications represents a new use for GPUs that has been enabled by the increasing generalization of GPU pipelines and the emergence of easier to program GPU programming languages like CUDA.

Our analysis in Section 2.3 showed that programs that use internal interpolation heavily are good candidates for using texture memory to obtain speedups. In the next chapter, we utilize this information and demonstrate how impressive speedups can be obtained for GRASSY, a non-graphics application that utilizes texture memory interpolation.

Chapter 3

Utilizing Texture Memory: GRASSY

Our analysis from the microbenchmarking in Section 2.3 showed that GPU programs that make heavy use of interpolation are ideal candidates for using texture memory. In this chapter, as one application case study, we implement an asteroseismology application called GRASSY. GRASSY heavily utilizes interpolation in texture memory to obtain speedups near 100x on GPUs over serial CPU implementations and 33x over parallelized CPU implementations. Thus, GRASSY represents a non-graphics application that benefits significantly from texture interpolations.

In order to understand our results, we mathematically analyze the limits of texture performance. We find that texture memory can offer tremendous potential speedups, but that the majority of the work being done in the application must be interpolation for the texture unit to provide speedups. If the majority of the work being done in the program is not done by the interpolations, then most problems will rapidly become bandwidth limited.

In the remainder of this chapter, we first provide background on asteroseismology, explain the spectral synthesis problem, and provide a more formal basis for the interpolation problem in Section 3.1. Next we describe the details of our GPU implementation called GRASSY in Section 3.2. After this, we show that GRASSY obtains large speedups over CPU implementations in Section 3.3. In Section 3.4 we discuss related work, then in Section 3.5, we create a mathematical framework based on our results in order to understand the ultimate upper and lower bounds on performance for texture memory interpolation. Finally in Section 3.6 conclude and discuss how our results affect future directions for this avenue of GPU computing research.

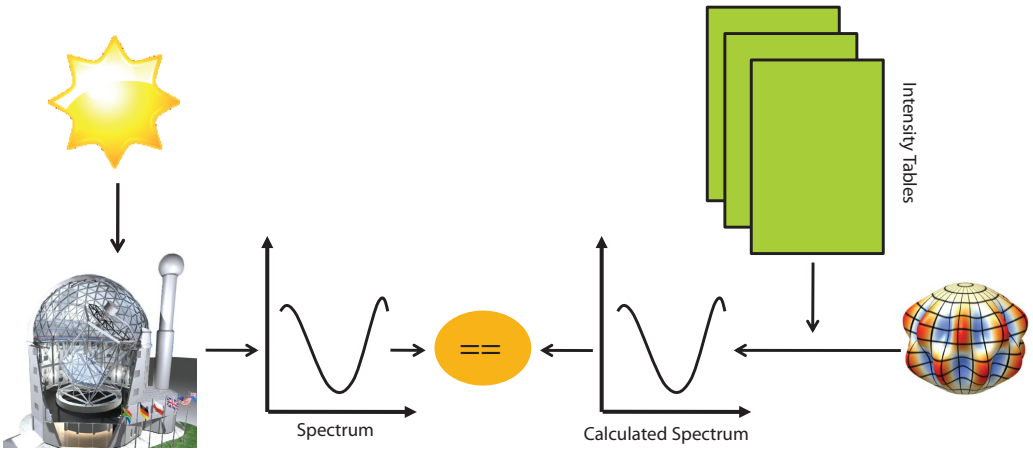


Figure 3.1: Pictorial representation of analyzing stars with direct modeling. Further details are available in [46].

3.1 GRASSY Background and Problem Statement

Asteroseismology

Asteroseismology is a powerful technique for probing the internal structure of distant stars by studying time-varying disturbances on their surfaces. A direct analogy can be drawn to the way that the study of earthquakes (seismology) allows us to infer the internal structure of the Earth. We associate each fragment with a set of values that we perform calculations on. Due to the recent launch of space satellites devoted to discovering and monitoring these surface disturbances in hundreds of thousands of stars, paired with ground-based telescopes capable of high-resolution, high cadence follow-up spectroscopy, asteroseismology is currently enjoying a golden age. However, there remains a significant computation challenge: how do we analyze and interpret the veritable torrent of new asteroseismic data?

To date, the most straightforward and accurate analysis approach is direct modeling via spectral synthesis (discussed in greater detail later in this section), an example of which is shown in Figure 3.1. Given a set of observations of the time varying radiant flux received from a star, we attempt to construct a sequence of synthetic spectra to reproduce these observations. Any given model depends on the assumed parameters describing the star and its

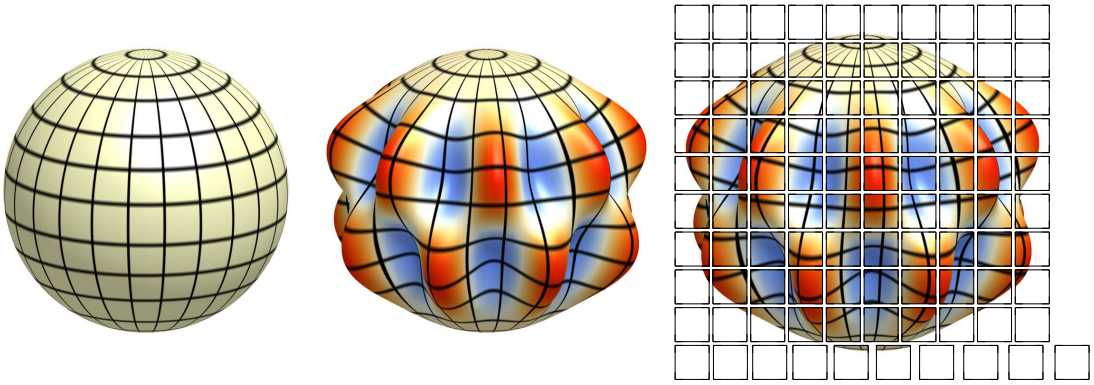


Figure 3.2: Discretizing a pulsating star.

surface disturbances. Using an optimization strategy we find the combination of parameters that best reproduces the observations, ultimately allowing us to establish constraints on the stellar structure.

Spectral Synthesis

Stellar spectral synthesis is the process of summing up the Earth-directed radiant flux from each region on the visible hemisphere of a star. A typical procedure, such as is seen in Figure 3.2, comprises the following steps:

1. *Mesh building.* Decompose the stellar surface (Figure 3.2) into a triangle mesh (Figure 3.3a). A quad [T, g, v, μ] (see Table 3.1) is associated with each mesh vertex (Figure 3.3b).

2. *Mesh rendering.* Rasterize the view-plane projection of every triangle to produce a set of N_{frag} equal area fragments (Figure 3.3c). Bilinear interpolation between triangle vertices is used to calculate a quad [T, g, v, μ] for each fragment.

3. *Flux calculation.* Evaluate the radiant flux spectrum $F(\lambda_j)$ on a uniformly spaced grid of N_λ discrete wavelengths with $\lambda_j = \lambda_0 + \Delta * j$ $(j = 0, ..., N_\lambda - 1)$ for each fragment (Figure 3.3d).

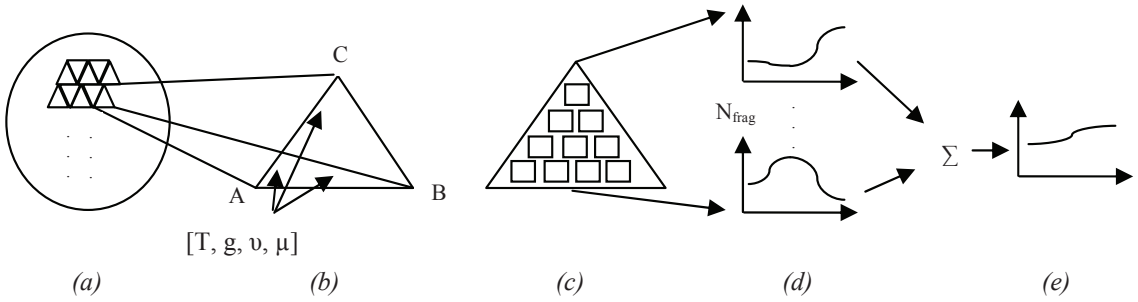


Figure 3.3: Steps in spectral synthesis process.

4. *Flux aggregation.* Sum the N_{frag} flux spectra to produce a single synthetic spectrum representing the radiant flux for the whole star (Figure 3.3e).

Steps (1) and (2) are performed once for a star and/or set of stellar and disturbance parameters. The main computational cost comes in step (3), which we now discuss in greater detail.

Over the wavelength grid λ_j the radiant flux is calculated as:

$$F(\lambda_j) = (A^2/D^2) * I(T, g, \lambda_j^{rest}, \mu) \tag{3.1}$$

Here, A is the area of the fragment, D the distance to the star, and $I(...)$ the specific intensity of the radiation emerging at rest wavelength λ_j^{rest} and direction cosine μ from a stellar atmosphere with effective temperature T and surface gravity g (see Table 3.1). The rest wavelength is obtained from the Doppler shift formula:

$$\lambda_j^{rest} = \lambda_j * (1 - v/c) \tag{3.2}$$

where v is the line-of-sight velocity of the fragment and c is the speed of light.

Calculating the specific intensity $I(...)$ requires detailed modeling of the atomic physics and energy transport within the stellar atmosphere and is far too costly to do on the fly. *Instead, we*

Table 3.1: Definitions of quad variables.

Variable	Description
T	Local effect. temperature – measures net radiant flux through the atmosphere.
g	Local surface gravity – measures the stratification of the atmosphere.
v	Projection of local surface velocity onto the line of sight.
μ	Projection of local surface normal on the line of sight.

evaluate this intensity by performing four-dimensional linear interpolation in pre-calculated tables of specific intensity data.

Flux Calculation and Aggregation

Here we establish a more formal basis for the interpolation problem. An example of CPU-oriented pseudo code for the flux calculation and aggregation steps of the the spectral synthesis procedure (steps 3 and 4) is given in Listing 3.1. Input to the code is a stream of quads from the mesh building and rasterization steps (steps 1 and 2 in the spectral synthesis process); the output is the aggregated synthetic spectrum. The high arithmetic and memory-access costs of the 4D specific intensity interpolation are evident in the 16-term weighted summation appearing toward the end of the code – this expression must be evaluated $N_{frag} * N_{\lambda}$ times during the synthesis of a single spectrum.

Problem Statement

For typical parameters (N_{frag}, $N_{\lambda} \approx 5,000 - 10,000$) a synthetic spectrum typically takes only a few seconds to generate on a modern CPU. However, with many possible parameter combinations, the overall optimization can be computationally expensive, taking many weeks in a typical analysis. The primary bottleneck lies in the 4-D specific intensity interpolations described in steps 3 and 4 of the spectral synthesis process; although algorithmically simple, their arithmetic and memory-access costs quickly add up. This issue motivates us to pose the following question:

```
1  float I_tables[][][][]; // Pre-calculated specific intensity tables
2  calc_and_agg_flux (input stream[])
3  {
4      float flux[N_lambda]; // Initialized to zero
5      foreach quad in input stream do
6      {
7          for j = 0..N lambda  1 do
8          {
9              // Rest wavelength
10             float lambda = lambda_0 + delta_lambda*j;
11             float lambda_rest = lambda*(1 - quad.v/speed_of_light);
12
13             // Accumulate flux
14             flux[j] += (A*A)/(D*D) *
15                        interpolate_I(quad.T, quad.g, lambda_rest, quad.mu);
16         }
17     }
18     return flux;
19 }
20
21 float interpolate_I (T, g, l, mu)
22 {
23     // Locate position in tables (assume increments in T, g, l and mu are 1.0)
24     i_T = floor(T);
25     i_g = floor(g);
26     i_l = floor(l);
27     i_m = floor(mu);
28     // Set up weights
29     w_T = T - i_T;
30     w_g = g - i_g;
31     w_l = l - i_l;
32     w_m = mu - i_mu;
33     // Do the 4-D intensity interpolation (16-term weighted summation)
34     I = (1-w_T)*(1-w_g)*(1-w_l)*(1-w_m)*I_tables[i_T ,i_g ,i_l ,i_m] +
35         (1-w_T)*(1-w_g)*(1-w_l)*(w_m)*I_tables[i_T ,i_g ,i_l ,i_m+1] +
36           ...
37         (w_T)*(w_g)*(w_l)*(1-w_m)*I_tables[i_T+1,i_g+1,i_l+1,i_m] +
38         (w_T)*(w_g)*(w_l)*(w_m)*I_tables[i_T+1,i_g+1,i_l+1,i_m+1];
39
40     return I;
41 }
```

Listing 3.1: CPU-oriented pseudo-code for the flux spectrum computation and aggregation.

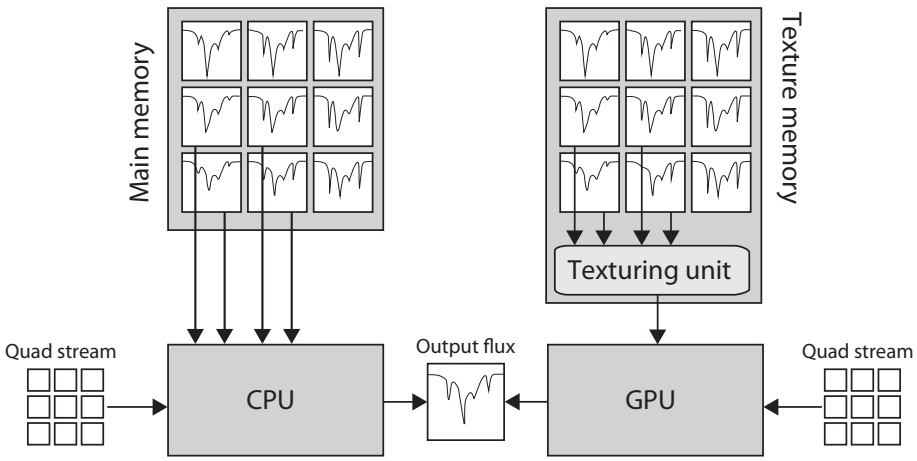


Figure 3.4: Schematic comparison of CPU and GPU spectral synthesis.

How can we use GPUs to accelerate interpolations in the precomputed specific intensity tables?

In addressing this question, we have developed GRASSY (GRaphics Processing Unit – Accelerated Spectral SYnthesis) – a hardware/software platform for fast spectral synthesis that leverages the texture interpolation functionality in CUDA-capable GPUs (see Figure 3.4). By utilizing textures, we can perform interpolations for "free" internally in hardware and reduce the overall runtime for this system significantly.

3.2 Porting the Code from the CPU to the GPU

In this section, we introduce the GRASSY spectral synthesis platform. To leverage the texture interpolation functionality in CUDA-capable GPUs, some modifications to the procedure outlined in Listing 3.1 are necessary. At a high-level, we map the interpolation data that is located in main memory in the CPU implementation to texture memory on the GPU. In particular, we decompose the lookups into a series of 2-D interpolations, which are then implemented as fetches (with bilinear filtering) from 2-D textures. This brings the dual benefits of efficient memory access due to use of the texture cache and "free" interpolation arithmetic provided by

the linear filtering mode. One potential pitfall, however, is the limited precision of the linear filtering. In previous work we have shown that these issues are not a problem for GRASSY, because the error due to the lose in precision is unimportant when contrasted against uncertainties in the tabulated intensities (due to theoretical approximations) and against the inherent noise in the observations against which synthetic spectra are compared [52].

3.2.1 Texture Packing

Because CUDA does not support arrays of texture references (which would allow us to place each table into a separate texture), GRASSY packs the 2-D intensity tables $I(\lambda^{rest}, \mu)$ into a single 2-D texture[1]. For a given interpolation, the floating-point texture coordinates (x, y) are calculated from λ, μ, and the table index k via:

$$x = (\lambda - \lambda_0^{tab})/\Delta * \lambda^{tab} + 0.5 \tag{3.3}$$

and

$$y = \mu * (N_\mu^{tab} - 1) + k * N_\mu^{tab} + 0.5 \tag{3.4}$$

Here, N_μ^{tab} is the direction cosine dimension of each intensity table, while λ_0^{tab} is the base wavelength and $\Delta\lambda^{tab}$ is the wavelength spacing (not to be confused with λ and $\Delta\lambda$). The offsets by 0.5 come from the recommendations of Appendix F.3 the CUDA programming guide of [1]. Note that these two equations are not the actual interpolation, but rather the conversion from physical (wavelength, direction cosine) coordinates to un-normalized texture coordinates.

3.2.2 Interpolation Decomposition

The 4-D to 2-D interpolation decomposition is enabled by a CPU-based pre-processing step, whereby quads [T, g, v, μ] from step (2) of the spectral synthesis procedure are translated

[1]We considered using a 3-D texture, but the dimension limitations imposed by CUDA are too restrictive.

into 10-tuples $[v, \mu, k_0, k_1, k_2, k_3, w_0, w_1, w_2, w_3]$. A single 10-tuple codes for four separate bilinear interpolations, in the 2-D tables $I(\lambda^{rest}, \mu)$ represents the specific intensity for a given combination of effective temperature and surface gravity. The indices $k_0, ..., k_3$ indicate *which* tables to interpolate within, while the weights $w_0, ..., w_3$ are used post-interpolation to combine the results together. We convert the quads to 10-tuples on the CPU as a pre-processing step, because the overhead to doing so is minimal and we reuse it on the GPU for all wavelengths.

3.2.3 Division of Labor

To take advantage of the massive parallelism offered by the GPU, we map a single wavelength to each GPU thread. Since the computations for each wavelength are independent of those for the other wavelengths, this allows our algorithm to maximize the performance it can obtain on the GPU. Every thread block processes the entire input stream of N_{frag} 10-tuples, to build up the aggregated flux spectrum for the wavelength range covered by the block. Upon completion, this spectrum is written to global memory, where it is subsequently be copied back to the CPU.

Additionally, to maximize texture cache locality, GRASSY groups calculations into N_b batches of $N = N_\lambda / N_b$ consecutive wavelengths. These batches map directly into CUDA thread blocks, with each thread in a block responsible for interpolating the intensity, and accumulating the flux, at a single wavelength. To ensure adequate utilization of GPU resources, N_b should be an integer multiple of the number of SMs in the device.

3.2.4 Pseudo Code

Listing 3.2 provides pseudo code for our CUDA kernel implementation within GRASSY, which handles the flux calculation and aggregation. Kernel arguments are the input stream of 10-tuples and a pointer to an array in GPU global memory where the resulting flux spectrum will be placed. Each thread evaluates its own wavelength on the fly from the index j, which in turn is obtained from its thread and block indices.

```
1  texture <float, ...> texRef; // Pre-loaded with 2-D intensity tables
2  __global__ void calc_and_agg_flux (input stream[], float dev_flux[])
3  {
4      // Thread wavelength
5      int j = blockIdx.x*N + threadIdx.x;
6      float lambda = lambda_0 + delta_lambda*j;
7
8      // Initialize flux
9      float flux = 0.0f;
10     foreach tuple in input stream do
11     {
12         // Rest wavelength
13         float lambda_rest = lambda*(1 - tuple.v/SPEED_OF_LIGHT);
14
15         // Accumulate flux
16         float I = tuple.w_0*interpolate_I(lambda_rest, tuple.mu, tuple.k_0) +
17                   tuple.w_1*interpolate_I(lambda_rest, tuple.mu, tuple.k_1) +
18                   tuple.w_2*interpolate_I(lambda_rest, tuple.mu, tuple.k_2) +
19                   tuple.w_3*interpolate_I(lambda_rest, tuple.mu, tuple.k_3);
20
21         flux += (A*A)/(D*D)*I;
22     }
23
24     // Write flux to global memory
25     dev_flux[j] = flux;
26 }
27
28 float interpolate_I (l, m, k)
29 {
30     float x = (l - LAMBDA_0_TBL)/DELTA_LAMBDA_TBL + 0.5f;
31     float y = m*(N_ MU_TBL - 1) + k*N_MU_TBL + 0.5f;
32
33     return tex2D(texRef, x, y);
34 }
```

Listing 3.2: Pseudo code for the unoptimized GRASSY kernel

3.2.5 Optimizations

Though not shown in the pseudo code in Listing 3.2, we found several optimizations could be applied to our implementation. First, to maximize the locality on the GPU, because our wavelengths are batched into groups, we can use shared memory for storing 10-tuple data and converting the 10-tuples into texture coordinates (i.e. doing the flux summations). This increases performance because now we usually access local memory instead of global memory. We also found that using loop unrolling helped with the performance of our shared memory calculations too, which in turn improved GRASSY's performance even further, as others have also shown [45]. Third, we reordered expressions in our kernels to eliminate overhead.

We found that our original implementation had very low texture cache hit rates (approximately 1%). We believe this was due to thrashing in the texture cache caused by many threads attempting to access values there. Even though some of the threads were attempting to access the same values, the values they were trying to access were getting evicted by other threads in between accesses. To solve this problem, we added synchronization points to our kernels between the accesses to the texture cache. Effectively, this forced the threads to be accessing the texture cache more closely together in time, which helped prevent thrashing, since threads that accessed the same values in the cache were accessing it around the same time (as a result our texture cache hit rate improved to above 90%). Finally, we converted our float variables into float4 variables to improve locality and allow their accesses to be coalesced.

A final optimization we made was to break our kernel into two separate kernels: a calculation kernel and an aggregation kernel. The calculation kernel is responsible for performing the interpolations in texture memory and calculating the flux for each wavelength, but writes its results into a partial array. This allowed us to aggregate the local flux immediately. This kernel utilizes the aforementioned optimizations such as synchronization between texture accesses and shared memory. The aggregation kernel then aggregates the local fluxes into the global flux array. By splitting the kernels apart, we are able to reduce contention writing to the output flux array by summing the flux locally first. The aggregation kernel uses fewer threads than the

calculation kernel to allow each thread to aggregate a fixed amount of the calculated fluxes into the correct locations in the output flux array and so that each thread has enough work to do.

The most important of these optimizations, in terms of performance, were using shared memory for aggregating the flux locally, splitting the kernel into two separate kernels, and synchronizing the threads before texture accesses to guarantee better texture cache hit rates.

3.3 Evaluation

3.3.1 Methodology

Our testing and validation platform is a Dell Precision 490 workstation, containing two 2.33 GHz Intel Xeon E5345 quad-core CPUs and a Tesla C1060 GPU (see Table 2.1). The workstation runs 64-bit Gentoo Linux (kernel 2.6.25) and uses version 3.1 of the CUDA SDK. As a CPU-based comparison code, we adopt a highly optimized Fortran 95 version of the KYLIE code [50], running on the same workstation. We run this serial version of the code and a parallellized version that uses OpenMP. We use specific intensity tables based on the OSTAR2002 and BSTAR2006 grids of model stellar atmospheres [33, 34], with intensities calculated using the synspec package. The input stream of 10-tuples is generated from a modified version of the BRUCE code [50]. Finally, we ran these tests for 42K wavelengths, 8K fragments, and increments in the specific intensity tables of 0.08 Angstrom. All results use total execution time in ms as their metric.

3.3.2 Results

The results in Figure 3.5 demonstrate that GRASSY obtains impressive speedups over both CPU implementations. The original GPU implementation does not include any of the optimizations discussed in Section 3.2.5. Despite this, it still outperforms both CPU implementations. Our optimized GPU implementation is approximately 100x faster than the optimized serial implementation and 33x faster than the parallelized version. If we look at the results per wavelength (not shown), the same trends hold. Additionally, we observe that the total execution

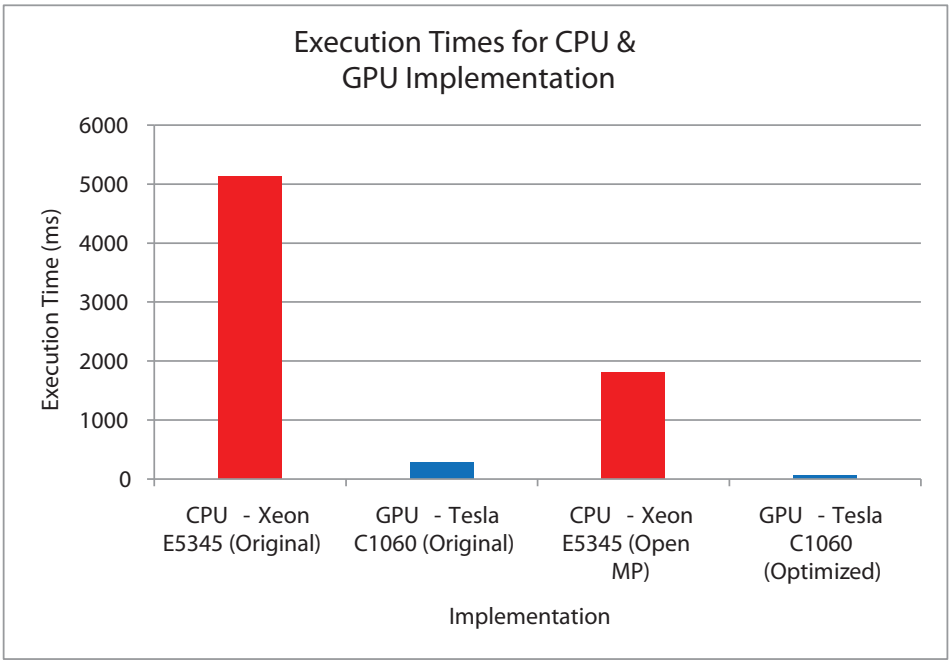


Figure 3.5: GRASSY GPU results vs. serial and parallel CPU implementations.

time scales linearly with the number of wavelengths. This is an intuitive result, because our application is embarrassingly parallel and we have not yet saturated the bandwidth of the texture caches.

To look at how GRASSY performs on GPU platforms other than the Tesla C1060, we ran GRASSY on several other GPUs (all GPUs listed in Table 2.1). The results in Figure 3.6 show that all the Nvidia GPU platforms all outperform both the serial and parallel CPU implementations. The amount by which each platform outperforms the CPU implementations depends on the GPU. This demonstrates how GPU performance does not always port from GPU to GPU. It also shows the value in having a higher-powered (and more expensive) GPU: the GPU performance increases as the GPU adds more SMs (and SPs) because the GPU can perform more operations in parallel than the smaller GPUs with fewer SMs. The results from these experiments also demonstrate that we can obtain performance increases over parallelized CPU implementations even when we're not using a high-powered GPU like the Tesla C1060.

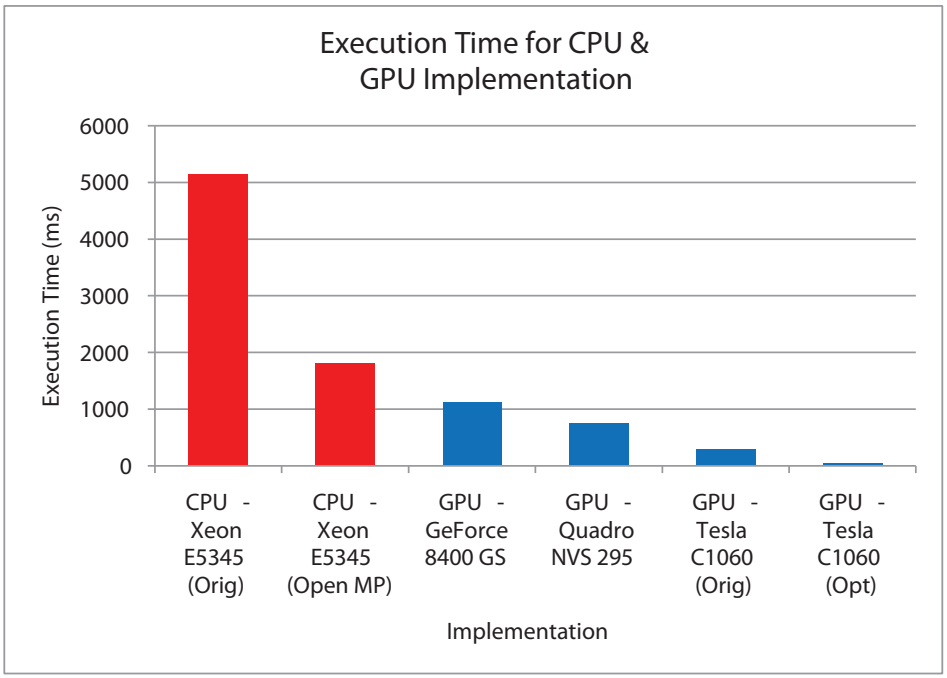


Figure 3.6: GRASSY GPU results for several different GPUs vs. CPU implementations.

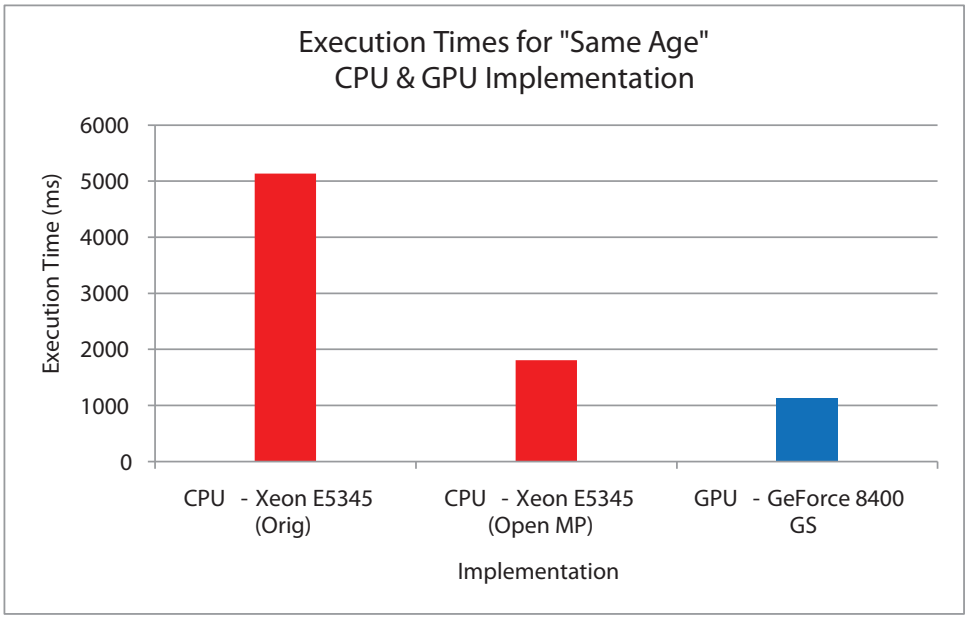


Figure 3.7: GRASSY GPU results for same age components as CPU.

A common complaint with GPU programming is that they run their tests on new GPUs and compare them to older CPUs, which artificially makes the performance of the CPU seem better [35]. To address this concern, we ran our results on an Nvidia GeForce 8400 GS GPU (see Table 2.1), which is approximately the same age as our Xeon E5345 CPU. It is important to note that the 8400 GS is not a top of the line GPU (it performed the worst of all the GPUs in Figure 3.6), unlike the Xeon E5345 CPU, which is a fairly high powered CPU for the time. Despite this, the 8400 GS still outperforms both the serial (4.6x better) and parallel (1.6x better) CPU implementations. Here we again see that we can outperform a relatively high-end CPU for a parallelized code with a low-end graphics card.

3.4 Related Work

The most relevant work in terms of asteroseismology is the BRUCE/KYLIE suite of modeling codes [50]. One example of typical analysis using the BRUCE/KYLIE codes is presented in [39]. Other works in astronomy that have used GPUs include [26, 59]. In terms of experimental data that GRASSY will analyze, current asteroseismic satellite missions include [17, 28, 44]. The sources for the spectral intensity data we use are [33, 34].

3.5 Mathematical Analysis

Previously, we looked at some intuitive access patterns, as well the performance our GRASSY platform obtains when using texture memory interpolation. In order to understand the ultimate upper and lower bounds on the performance of texture memory interpolation, we developed a simple mathematical framework based on the results we obtained in Sections 2.3.3 and 3.3.2.

For a given GPU, let us assume that it has *T* texture units and *C* compute units. Additionally, if the GPU is given work *W*, assume that it can complete the work in W_T cycles if the work is done on the texture unit, and W_C cycles if the work is done on the compute units instead (which would require accesses to global memory, performing interpolations manually, etc.). Because the compute unit-based approach requires more accesses to global memory, it requires more

bandwidth than the texture-based approach. Additionally, assume that the GPU performs N lookups to complete a given task, regardless of the memory location(s) the lookups use. Given these assumptions, we obtain the following equations:

$$executionTime_{Texture} = (W_T * N)/T$$
$$executionTime_{Compute} = (W_C * N)/C$$

Looking at the ratio between these execution times:

$$\begin{aligned} executionTime_{Texture}/executionTime_{Compute} &= ((W_T * N)/T)/((W_C * N)/C) \\ &= (W_T * C)/(T * W_C) \\ &= (W_T/W_C) * (C/T) \end{aligned}$$

Because one texture lookup internally does four multiply-accumulate operations for the interpolation of a 2D texture (these operations would need to be explicitly done on the compute units), we conclude that, for 2D textures, $W_T/W_C = 1/4$. Additionally, for the GPUs we've examined, the C/T ratio is 2:1 or 3:1. This presents an issue: one wants to utilize the superior compute power of the texture unit, but they are then constrained by the small number of texture units available on the GPU. If the majority of the work being done is not interpolation, then *most problems that use texture units will rapidly become* **bandwidth limited**, because having fewer texture units puts more of a strain on the global texture memory to satisfy requests to the texture unit.

Each Texture Processing Cluster (TPC), which is usually made up of two or three SMs (depending on the generation of GPU), has a local texture cache. A hit in this texture cache reduces demand on the global texture memory but does not reduce latency; thus, hitting in the local texture cache allows more texture requests to be satisfied simultaneously. This can be seen from the experiments performed in Section 2.3.3, which rapidly become bandwidth limited and approach an asymptote of performance at the maximum throughput. Thus, to

more effectively utilize the texture unit and its “free” interpolations, we need to have problems in which a large percentage of the work is interpolation, and in which lots of interpolations are performed, because the interpolation provides us with a lot of work that we’d have to do manually otherwise. Otherwise, the texture unit will become a bottleneck and increased performance over non-texture implementations will be difficult to obtain.

3.6 Summary

GRASSY is an asteroseismological application that is dominated by performing nearest neighbor interpolation in tables of pre-calculated intensity data. In this chapter, we have shown that mapping its pre-calculated intensity data to the GPU texture memory allows us to obtain impressive speedups over optimized CPU implementations. The results also demonstrate that we can take advantage of the “free” computations that GPU texture memory provides, something that has mostly been unexplored in past literature. Additionally, even a lower-end GPU that is approximately the same age as our CPU provides a speedup over parallel CPU implementations. Overall, these results represent a significant increase in performance that allows us to reduce the runtime of our system significantly.

GRASSY will likely have a significant impact on the field of asteroseismology, as the analysis throughput and/or resolution of parameter determinations can now be greatly increased without huge increases in runtime, in turn allowing stronger constraints to be placed on the internal structures of stars. We believe that the benefits of accelerated spectral synthesis will also extend to other fields of astrophysics such as the analysis of rotationally flattened stars, the generation of combined spectra for clusters or entire galaxies of stars (so-called population synthesis), and other techniques both inside and outside of astrophysics that rely on interpolation.

Our mathematical analysis in Section 3.5 shows the characteristics that programs need in order to benefit significantly from the internal interpolation provided by texture memory. If programs do not use a significant amount of interpolation, they will likely find that the texture memory becomes a rapid bottleneck, due to the design of the GPU architecture (the

TPCs design specifically). As GPUs have become more programmable over the years, GPU architects have decreased the ratio of texture caches to compute cores (increased the number of SMs per TPC). This means that texture units will more rapidly become a bottleneck.

However, for programs that benefit significantly from interpolation significant performance improvements can be obtained. GRASSY is an example of a program that heavily utilizes interpolation in its calculations; it is dominated by performing nearest neighbor interpolation in tables of pre-calculated intensity data. By mapping this pre-calculated intensity data to the GPU texture memory, we are able to obtain impressive speedups over optimized CPU implementations. Programs with similar characteristics to GRASSY should be able to benefit significantly from the use of texture memory interpolation for non-graphics applications, which is a new use for GPUs.

Chapter 4

Porting CMP Benchmarks to GPUs[1]

As we mentioned previously, GPUs are massively parallel architectures that are becoming increasingly general-purpose, use the data parallel programming model and offer tremendous speedups if an applications' algorithm maps well to the data parallel programming model. Many scientific workloads map well to the GPU programming model, as we discussed in Chapter 1. However, it is neither clear nor obvious if other workloads and applications that aren't data parallel also map well to the GPU. If applications that are not data parallel can be mapped to the GPU programming model, then the GPU could become a truly general-purpose architecture.

Additionally, the computer architecture community is currently facing a challenge: where is the next major increase in general-purpose program performance going to come from? As the number of transistors on a chip continue to increase, there are increased opportunities for on-chip parallelism, but it is unclear how architects should effectively harness it. Some have postulated that a GPU-like architecture is the answer to these questions; for this to be true, GPUs will need to effectively execute general-purpose programs with high performance. The goal of this chapter is to examine if applications that have traditionally been targeted for CMPs can be easily mapped to GPUs.

In this chapter, we successfully ported four general-purpose CMP benchmarks from the PARSEC benchmark suite [7, 8] to GPUs using CUDA SDK 2.3 [1] and evaluated their performance. The benchmarks we implemented were streamcluster, blackscholes, fluidanimate, and

[1]This chapter is based on project work done for CS/ECE 757, during the Spring 2010 Semester, under the supervision of Professor Mark Hill.

swaptions. More details on these programs and their GPU implementations are presented in Sections 4.1 and 4.2. By porting these PARSEC benchmarks to GPUs, we were able to examine GPU features in relation to CMP programs. Specifically, we found what features of general-purpose programs were well-suited to the GPU architecture. Perhaps more importantly, we found the features that hindered high performance execution on a GPU and the corresponding bottlenecks that precluded speedups when these features were present.

The rest of this chapter is as follows. In Section 4.1 we provide background on the PARSEC benchmark suite, the benchmarks we implemented, and related work in the area. In Section 4.2 we discuss our GPU implementations of the benchmarks. In Section 4.3 we outline our testing methodology and system information. In Section 4.4 we present and analyze our results. Finally, in Section 4.5 we conclude.

4.1 Background and Related Work

We chose to implement benchmarks from the PARSEC suite over other CMP benchmark suites like SPLASH-2 [56] because recent work has shown that PARSEC scales significantly better than SPLASH-2 [9, 12]. While PARSEC also has some scalability issues, they are less severe than those of SPLASH-2. Additionally, SPLASH-2 was developed over fifteen years ago and is no longer representative of workloads that future architectures will face, especially in terms of data set size. Compared to SPLASH-2, PARSEC also has a much more diverse application set and contains more emerging workloads. These features are important because they allow us to analyze the performance of modern and emerging applications on GPUs. We chose a CMP benchmark suite because GPU benchmark suites generally contain only programs that work well on GPUs, whereas we wanted to explore both programs that work well and those which might pose problems for a GPU implementation. Table 4.1 contains an overview of relevant information about the benchmarks we implemented on GPUs.

We present some brief background information on the ported benchmarks here:

- Streamcluster is a data mining algorithm that solves the on-line clustering problem. It requires a heuristic solution since the exact solution is computationally intractable. Further

information on the algorithm can be found elsewhere [43]. Streamcluster was chosen because it has a moderate amount of parallelism and lots of synchronization. We expect that streamcluster's low amount of data sharing between threads will allow it to avoid issues with synchronizing data between GPU threads, which is important because GPUs lack an efficient global synchronization mechanism. Finally, we wanted to explore how an application without significant amounts of parallelism would perform on a GPU, where abundant parallelism is usually essential for obtaining high performance.

- Blackscholes is a financial algorithm that uses the Black-Scholes partial differential equation (PDE) to calculate prices for European stock options. The key idea is that the value of the option fluctuates over time with the actual value of the stock. The Black-Scholes PDE calculates this value over time, but because it has no closed form solution, it needs to be solved numerically. It has abundant parallelism and uses the SIMD programming model. Further information on the Black-Scholes algorithm can be found elsewhere [11, 30]. We selected blackscholes because it had abundant amounts of parallelism and uses the SIMD programming model, which makes it ideal for implementing on a GPU. Thus, blackscholes represents a good sanity check – it should obtain high performance on GPUs.

- Fluidanimate simulates interactions of an incompressible fluid by breaking the fluid into particles and assigning groups of particles to cells. In-depth information on the algorithm can be found elsewhere [41]. Compared to other PARSEC benchmarks, fluidanimate has less synchronization. However, fluidanimate still requires synchronization points between various stages of its calculations and requires the use of atomics to update memory, which are important features for general-purpose applications that GPUs must be able to execute well.

- Swaptions is a financial analysis program that calculates prices for a portfolio of swaptions using a Monte Carlo simulation to compute the prices. Swaptions also has a PDE that must be solved numerically. Further information on the algorithm can be found

Table 4.1: Background information on implemented PARSEC CMP benchmarks.

Benchmark	**Domain**	**Parallelization Granularity**	**Working Set Size**
blackscholes	Financial Analysis	coarse	small
fluidanimate	Animation	fine	large
streamcluster	Data Mining	medium	medium
swaptions	Financial Analysis	coarse	medium

elsewhere [29]. A unique feature of swaptions is that it has very coarse and limited parallelism. We chose swaptions to see how a program with a limited amount of parallelism but a large amount of floating-point calculations would perform on a GPU. While GPUs are good at floating-point calculations, they generally needs lots of parallelism in order to perform well.

Some work on implementing the PARSEC benchmarks on GPUs has been done previously. Rodinia implemented a small portion of streamcluster in a GPU kernel [14]. This kernel was heavily optimized, and was shown to provide significant speedups. By implementing only this small portion on the GPU, they were able to avoid dealing with several issues our implementation faced. However, their results only analyzed the speedup of their kernel, as opposed to measuring the total speedup of the entire program. Because of this, it was difficult to gauge the impact of their optimization on the overall program.

Kolb and Pharr implemented blackscholes on a GPU [32]. However, their implementation differs significantly in three areas from our implementation. First, their implementation uses a randomly generated sequence of stock options instead of reading in options from an input file as the PARSEC implementation does. Second, their implementation converts the arrays that are used in the PARSEC implementation for risk rate and volatility calculations into constants. Making these arrays constants significantly reduces the overhead of copying data between the CPU and GPU and limits their program to only using a single risk rate and volatility. Third, their program used a fixed number of thread blocks and threads per thread block, whereas we

have varied these numbers based on the number of options we are using. It is possible that they are using a fixed number of thread blocks and threads per thread block because they have a fixed number of options per program run, and thus they found these operating points to be optimal for their implementation. We incorporated some of their optimizations that reduced the number of necessary mathematical operations into our implementation, as we found they provided a significant performance increase.

4.2 GPU Implementations

In this section we present details on our GPU implementations for all four benchmarks. The results for all of these implementations can be found in Section 4.4. It is important to note that, in all of our implementations, we sought to make small modifications to the current algorithms instead of making wholesale algorithmic changes.

4.2.1 Streamcluster

Because streamcluster has moderate amounts of parallelism and a significant amount of inter-thread synchronization, its GPU implementation uses a single large kernel. On the CPU, the stream of data points is broken into subsets of 200K points. If there are more than 200K points, the subsets are run sequentially through the kernel. The GPU kernel is responsible for all of the calculations streamcluster performs to find the centers. The significant number of inter-thread synchronization points was a major issue with implementing streamcluster on a GPU because CUDA does not provide a mechanism to synchronize across thread blocks (i.e. no global synchronization mechanism). Thus, our streamcluster implementation was limited to a single block of threads. Since we have up to 200K points in a single kernel, each thread works on multiple data points.

Our GPU implementation of streamcluster also required the use of a random number generator on the GPU. Since CUDA does not provide a standard random number generator to use we modified a previous solution [54][2]. This required a significant amount of time and testing.

[2]Nvidia has since created the CURand library to deal with this issue.

A second issue we encountered was CUDA's lack of kernel support for C++[3]. To solve this, we converted all of the C++ code that the kernel needed to execute into C code. One positive aspect we found was that it was easier to think about how to write broadcast and wait global synchronization mechanisms when writing GPU code. Streamcluster uses broadcast and waits to have a single master thread operate on the data points, after which is signals the others threads that they can now safely continue to operate on the data. Because we were using a single block of threads, we could check if we were the master thread or not, and operate on the data if and only if we were the master thread. Meanwhile, the other threads simply waited at a barrier for the master thread to reach them, at which point they could successfully execute once again. Of course, this was only possible because we only used one thread block. Overall, it was much simpler to reason about and write this code than it was with pthreads.

4.2.2 Blackscholes

Of the four CMP benchmarks we implemented on GPUs, blackscholes was best suited to take advantage of the features of GPUs, because it has abundant parallelism without inter-thread synchronization. Additionally, it performs a significant number of floating-point computations per thread, which allows it to effectively hide memory latencies. Our GPU implementation performs the PDE approximation in the kernel. Each GPU thread was assigned to a single stock option. For the maximum data set size, we had ten million threads, which provides significant amortization of memory latency.

Initially, our GPU design for blackscholes copied all of the data needed to execute blackscholes into global memory on the GPU. This is inefficient, because accessing global memory on the GPU is slow. To optimize our design, we instead placed the data into the texture cache memory on the GPU. While this required more overhead to copy the data from the CPU to the texture cache, it significantly decreased memory access time of our GPU code because we perform caching on the GPU. As mentioned in Section 4.1, we also incorporated Kolb and Pharr's

[3]Nvidia has also improved this over time, but it is still an issue.

optimized mathematical operations to further improve performance by performing more fused multiply-adds and fewer total mathematical operations [32].

4.2.3 Fluidanimate

The synchronization points between various stages of the particle interaction calculations was the major design feature that we needed to work around in our fluidanimate GPU implementation. As mentioned previously, CUDA does not have a mechanism for synchronizing between thread blocks, so to achieve good performance it was important to avoid performing these synchronization points on the GPU. To get around these interstage synchronization points, we created multiple kernels, one for each of the six stages of the particle interaction calculations. This allows us to implicitly synchronize at the host CPU in between kernel executions. Implementing our code in this manner meant that at the end of each kernel, our code would return from the GPU to the CPU, creating an implicit synchronization point. While there is a cost to returning to the CPU from the GPU, we were able to avoid synchronizing repeatedly in the kernel.

Implementing a kernel for each stage allowed us to vary the number of threads for each kernel based on the amount of parallelism present in that stage. In two of the stages, there were atomic operations that we could not avoid by returning to the GPU. In these cases, we implemented a custom mutex using an atomic Compare-and-Swap, which was necessary because CUDA atomic operations didn't support floating-point atomic operations.[4]

4.2.4 Swaptions

While swaptions and blackscholes perform similar tasks, swaptions has significantly less parallelism than blackscholes does. Additionally, we didn't think that a single large kernel (similar to our streamcluster implementation) would work well for three reasons. First many of swaption's functions do not have very many computations. Second, the functions have significant amounts of thread divergence. Third, the functions require significant amounts of

[4]The Nvidia Fermi architecture has added some floating-point atomic support.

memory transfer from the host. Effectively, these issues mean that swaptions is less algorithmically able than blackscholes to hide memory access latency and keep high SIMD efficiency by avoiding branches. Thus, we chose to instead implement a smaller kernel that performed the PDE approximation calculations, which we felt was best suited to executing on the GPU. However, swaptions suffered from a general lack of parallelism – the maximum number of total threads was 528. One of these kernels required the use of a random number generator on the GPU, so we used the same random number generator that we used for streamcluster.

4.3 Methodology

4.3.1 Verification and Performance Testing

Our first priority in testing the GPU implementations of the PARSEC benchmarks was ensuring that the GPU implementations obtained the correct results. Our primary means of ensuring correctness was via comparison to the results obtained by the PARSEC pthreads and serial versions for the same input sizes. However, in the cases where a random number generator was used on the GPU, it was not possible to verify the correctness of our results by comparing them to the results from PARSEC. To verify that our implementation was correct, we passed in identical constant numbers (instead of random numbers) to both the PARSEC implementation and our GPU implementation, then made sure that the results matched.

To compare the performance of our GPU implementations to the CPU implementations, we implemented a timing metric in our GPU implementations to measure execution time of the entire program. We also measured the execution time of each individual kernel and the data transfer times, but do not present those results. To ensure that we were making direct comparisons with the PARSEC results, we replaced the PARSEC timing metric with the same metric we used in our GPU implementations.

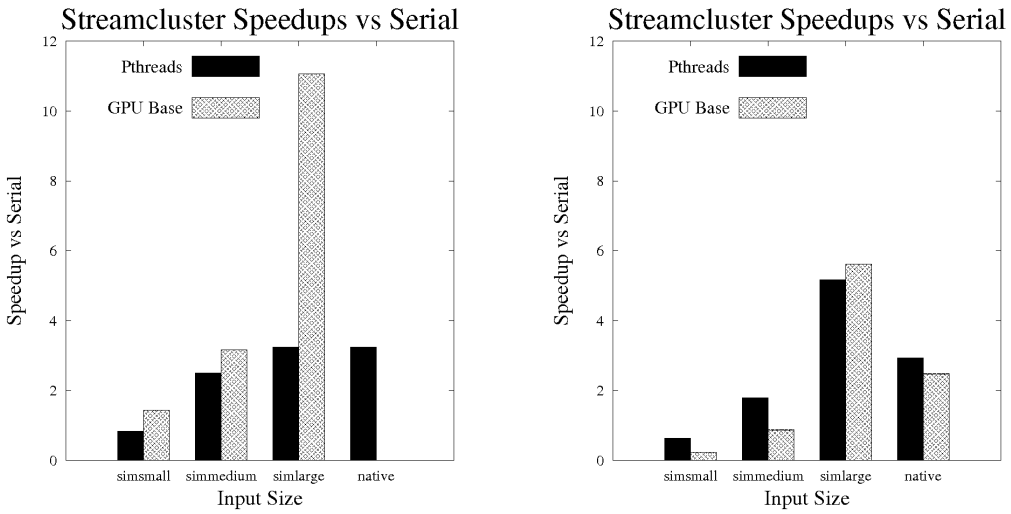
4.3.2 System Specifications

Our implementations were done in Nvidia's CUDA SDK 2.3. To obtain performance results, we ran our GPU implementation, the PARSEC pthreads implementation, and the PARSEC serial implementation on two of the systems from Table 2.1. The first system is the Nvidia Quadro FX 580 GPU and Intel Nehalem i5 Quad-core CPU. The second system is the Nvidia Tesla C1060 GPU and 2 Intel Xeon quad-core CPUs[5]. Running the tests on two different systems enabled us to obtain results on the significantly more powerful Tesla GPU. In addition, the Tesla GPU has more memory than the FX 580, which allowed us to run some of the larger experiments that couldn't be run on the FX 580. We were also interested in seeing if our results remained constant over different GPUs with significantly different computational power (which our GRASSY results indicated wasn't the case).

4.4 Results and Analysis

All reported speedups are normalized to the execution time of the PARSEC serial implementation on that system. For clarity, the results for the first testing system (FX 580 GPU and Nehalem Quad-core CPU) are labeled with "On FX 580 GPU and Quad-core CPU" and the results for the second testing system (Tesla C1060 GPU and Xeon 8-core CPU) are labeled with "On C1060 GPU and 8-core CPU." For both systems, the results were averaged over ten runs. Additionally, the PARSEC pthreads and our GPU implementations were run for a varied numbers of threads for the benchmarks that did not use a number of threads based on the input size. The results presented here represent the number of threads we found obtained the best performance for that benchmark. The results are presented over several input sizes (simsmall, simmedium, simlarge, and native), except for blackscholes, which uses the same data set sizes, but lists its results by the number of inputted options for clarity. These data set sizes vary per program and increase in size from left to right. The data sets are explained in detail elsewhere [7, 10]. In the next four subsections, we discuss the results for each benchmark.

[5] We refer to this as an eight core machine hereafter.



(a) On FX 580 GPU and Quad-core CPU. The pthreads results use 4 threads. The GPU results use 256 threads.

(b) On C1060 GPU and 8-core CPU. The pthreads results use 8 threads. All GPU results use 512 threads except for native, which uses 256 threads.

Figure 4.1: Streamcluster GPU speedups over serial CPU implementation.

4.4.1 Streamcluster

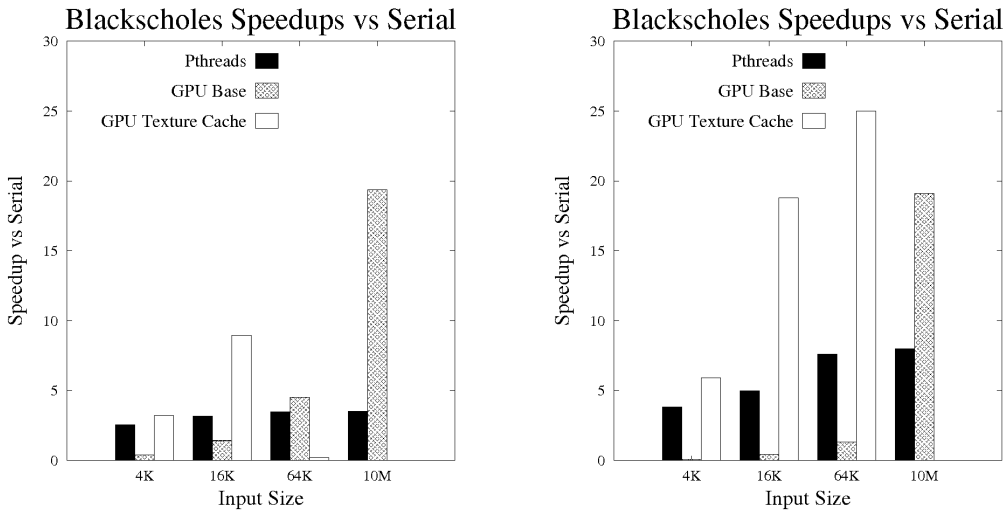
Figure 4.1a contains the results for streamcluster when it was run on the FX 580 GPU and Nehalem Quad-core CPU. As the input size increases, the GPU implementation's performance continues to increase through the simlarge input size. The poor performance for smaller input sizes occurs because there are only a few computations being performed per thread. Since we have numerous synchronization points, relatively little work gets done per synchronization point when there is little work to do. However, as the number of data points increase with the input size, there are more points per thread and more computations can be done between synchronization points, which minimizes the impact of the synchronization points, and high performance is obtained for the simlarge input size. For the native test size, the FX 580 does not have enough GPU memory so results for this data point cannot be obtained.

Figure 4.1b contains the results for streamcluster when it was run on the Tesla C1060 GPU and Xeon 8-core CPU. Because the C1060 has more memory than the FX 580 it is able run the native test size. The baseline results are also worse than those on the other systems (not shown). In general, the GPU results differ significantly than those obtained on the other system. For the simsmall and simmedium test sizes, the GPU implementation does not provide a speedup over the pthreads version. Additionally, for the simsmall test, the GPU implementation does not even provide a performance improvement over the serial implementation. These results are more in-line with the results we were expecting as compared to those obtained on the FX 580, because these small test sizes do not perform enough work per synchronization point per thread. However, for the simlarge input size, there is enough work per thread such that a significant amount of work can be done per synchronization point. This trend does not continue for the native test, which indicates that the simlarge test size provides an optimal computation to synchronization ratio for the GPU. Overall, for streamcluster we conclude that the amount of work being done per synchronization point is the key metric.

It is important to note that these results for the FX 580 GPU and Nehalem Quad-core CPU were obtained when X11, the network graphical user interface, was turned on. When X11 was turned off, performance decreased for all input sizes. When X11 is turned off, GPU performance on the FX 580 system decreases by roughly 2x. We believe this issue occurred due to modifications needed to make CUDA SDK 2.3 atomics run correctly with Fedora 12, the operating system on that machine.

4.4.2 Blackscholes

Figure 4.2a contains the results for blackscholes when it was run on the FX 580 GPU and Nehalem Quad-core CPU. These graphs match the intuition we had for blackscholes: as the number of threads increase, the performance of the unoptimized GPU implementation increases. As the number of options increases, the number of threads increases proportionately which allows us to hide the latency of accessing global memory more effectively. Performance of the unoptimized GPU implementation is poor for the smaller input sizes because there are



(a) On FX 580 GPU and Quad-core CPU. The pthreads results use 4 threads. The number of GPU threads is proportional to the input size.

(b) On C1060 GPU and 8-core CPU. The pthreads results use 8 threads. The number of GPU threads is proportional to the input size.

Figure 4.2: Blackscholes GPU speedups over serial CPU implementation.

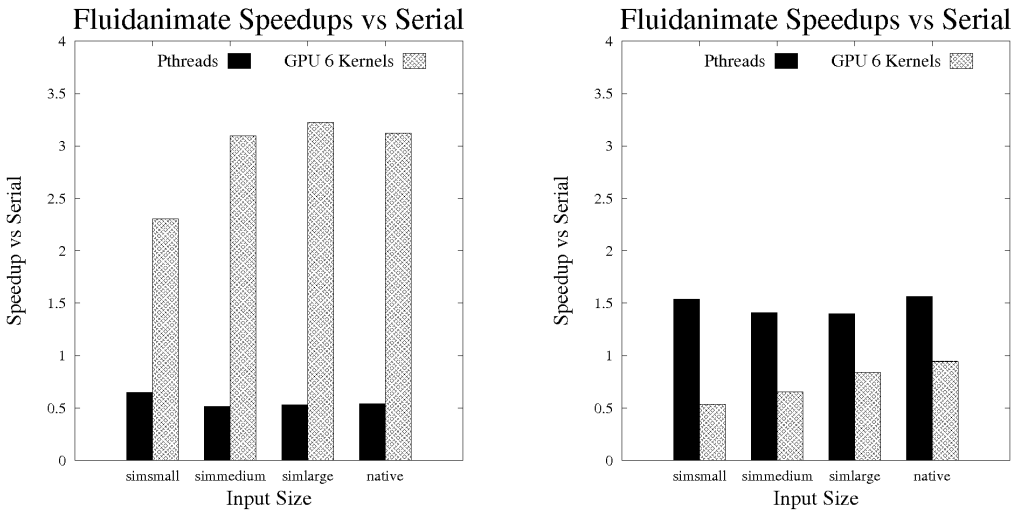
not enough threads to hide the latency of accessing main memory. Performance of the unoptimized GPU implementation overtakes performance of serial between the 4K and 16K options tests and exceeds it for all larger input sizes. The performance of the unoptimized GPU implementation passes the performance of pthreads between the 16K and 64K options tests, and exceeds it for all larger input sizes.

The performance of the optimized texture cache GPU implementation exceeds that of the serial, pthreads, and unoptimized GPU implementations immediately. This is because we have decreased our latency to access memory on the GPU significantly by accessing texture memory instead of global memory; accessing texture memory is much faster than accessing global memory because the data is cached nearby. Thus, using the texture memory like a lookup table allows us to cache the data we're accessing nearby the cores and improve performance. This is another new use of texture caches, albeit one that has been explored somewhat previously [48, 57]. However, once the number of options increases to 64K, the texture cache starts to have

capacity misses, since it can no longer hold the entire working set and must swap data with the global texture memory. It also starts to exhibit thrashing, because all the threads that are accessing it are requesting different data, data which cannot all be stored locally at these input sizes. At this point, the performance of the optimized texture cache implementation decreases significantly, back to the performance of the serial version. Finally, at the largest input size, the texture cache is no longer able to allocate the amount of texture cache memory necessary to run the kernel, so it is unable to produce results. However, for the smaller input sizes, the optimized GPU implementation performs extremely well, providing a significant speedup over all other implementations.

Figure 4.2b contains the results for blackscholes when it was run on the Tesla C1060 GPU and Xeon 8-core system. The results for the unoptimized version closely mirror those of the unoptimized version on the other system. As the number of threads increase, the performance of the unoptimized version also increases, as was seen before. One difference is that the performance of the unoptimized version does not exceed the performance of the pthreads version until after the 64K test.

Similarly, the optimized version outperforms all other implementations for the smaller sized inputs, but again is unable to run the largest input size. However, because the texture memory on the C1060 is larger than that on the FX 580, the performance does not decrease until after the 64K test size, because we can still store all of the requested data locally at that point. This demonstrates that having a more powerful GPU can increase performance significantly in some cases. Another interesting result we observed for both systems is that the pthreads implementations achieved their optimal performance when they were using one thread per core. Having a single hardware thread per core usually utilizes the hardware the best while providing the lowest overhead, so this result matched our expectation. Overall, we find that blackscholes maps well to the SIMD paradigm and that having many threads helps hide memory latencies and increases performance. Finally, using textures caches the data and brings it closer to the SPs, which further improves performance for smaller input sizes.



(a) On FX 580 GPU and Quad-core CPU. The pthreads results use 4 threads. The GPU results use 512 threads.

(b) On C1060 GPU and 8-core CPU. The pthreads results use 8 threads. The GPU results use 16K threads.

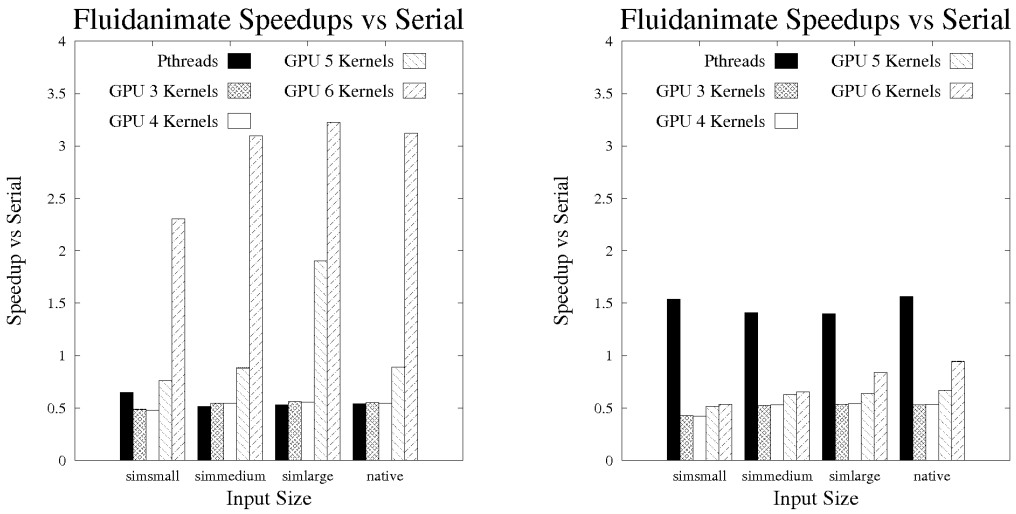
Figure 4.3: Fluidanimate GPU speedups over serial CPU implementation.

4.4.3 Fluidanimate

The fluidanimate results in Figures 4.3a and 4.3b only include the results for pthreads and the GPU implementation with six kernels, because the six kernel implementation was found to have the best performance of all of the GPU implementations, which used varying numbers of kernels[6]. Thus, in general in this section, it is assumed that the GPU implementation uses six kernels. For reference, the graph comparing the performance of the GPU implementations for the varying number of kernels can also be found in Figure 4.4.

Figure 4.3a contains the results for fluidanimate when it was run on the FX 580 GPU and Nehalem Quad-core CPU. The results obtained for the GPU implementation show that it provides a modest speedup over the serial version and the pthreads version for all input sizes. This

[6]In later tests we found that 3 kernels provided the best performance. We note this but do not show the updated results.



(a) On FX 580 GPU and Quad-core CPU. The pthreads results use 4 threads.

(b) On C1060 GPU and 8-core CPU. The pthreads results use 8 threads.

Figure 4.4: Fluidanimate GPU speedups versus serial CPU implementation for a varying number of GPU kernels.

shows that, for certain GPUs and certain programs, using multiple kernels can provide a performance increase. However, because the performance increase is relatively modest, which may dissuade programmers from implementing a program like fluidanimate on a GPU. It should also be noted that the performance of pthreads on this system was very poor, which makes the speedups obtained for the GPU implementation interesting.

Figure 4.3b contains the results for fluidanimate when it was run on the C1060 Tesla and 8-core CPU. The results obtained on this system differ significantly from the results obtained on the other system, but they match our intuition much better. While the performance of the pthreads version does increase as compared to the performance obtained on the other system, it is still relatively poor, barely better than the performance of the serial implementation. Additionally, the speedups seen on the other system for the GPU implementation are not seen in this

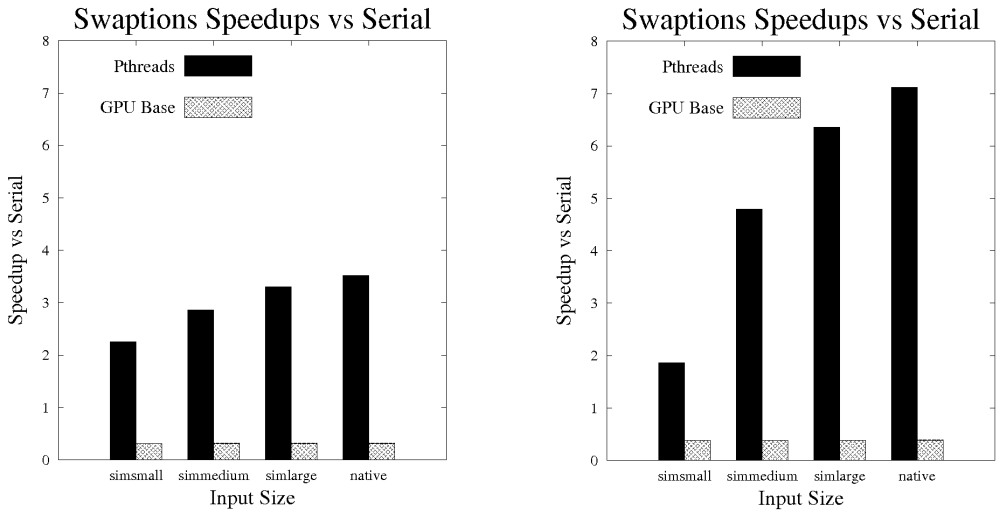
case. In fact, the GPU implementation fails to achieve a performance increase over the serial version for any of the test sizes.

There are several likely causes for fluidanimate's poor performance. GPU optimizations are often specific to a GPU, and may not perform as well on another GPU, this may be a possible cause here. Additionally, this implementation exhibits thread divergence, which significantly decreases performance. It also has register pressure for some of the larger kernels, which limits the maximum number of threads we can execute in that kernel. Finally, the decreased performance on this machine also show that atomic operations on the GPU are costly, especially for floating-point numbers. It is likely that these results are also skewed by the same X11 issue that plagued the streamcluster results, as fluidanimate exhibited synergistic behavior with streamcluster in relation to X11. When X11 is turned off, there are much more moderate performance increases in performance as size increases and the GPU performance never exceeds that of the serial CPU implementation.

4.4.4 Swaptions

Figure 4.5a contains the results for swaptions when it was run on the FX 580 GPU and Nehalem Quad-core CPU. The GPU implementation results are extremely poor for all test sizes. This is likely because swaptions has very limited parallelism due to inherent limitations in the algorithm itself. Additionally, the GPU implementation suffers from thread divergence, register pressure, and dynamic loop bounds. Thread divergence is caused by conditional statements being executed on the GPU. Dynamic loop bounds prevented us from achieving acceptable performance when we implemented other kernels on the GPU. Register pressure limits the number of threads we can execute, which decreases the already limited amount of parallelism even further. Finally, because we have such limited parallelism, the overhead of copying data between the CPU and GPU can't be amortized effectively. The results for the other kernels we implemented for swaptions only decreased performance further, so they have been omitted.

Figure 4.5b contains the results for swaptions when it was run on the C1060 Tesla and 8-core CPU. These GPU results mirror the results obtained on the other system nearly exactly.



(a) On FX 580 GPU and Quad-core CPU. The pthreads results use 8 threads. The GPU results use 528 threads.

(b) On C1060 GPU and 8-core CPU. The pthreads results use 32 threads. The GPU results use 528 threads.

Figure 4.5: Swaptions GPU speedups versus serial CPU implementation.

The only difference is that pthreads scales slightly better. This is likely due to this system having more cores, which means the threads do not need to compete for resources on the same cores.

4.5 Summary

In general, we found that the PARSEC CMP benchmarks do not port very well to GPUs. The notable exception is blackscholes, due to it is embarrassingly parallel nature. The use of texture memory in blackscholes provided further increases in performance by allowing data to be accessed locally instead of being accessed from main memory. The performance of our streamcluster implementation improved as the number of data points increased (through the simlarge input size), because the synchronization point to computation ratio improved. The

performance is maximized on both systems in the simlarge case, a behavior that is also exhibited by the PARSEC pthreads implementation, which signifies that this input size maximizes the computation to synchronization points ratio. Fluidanimate's performance was improved through the use of multiple kernels, which took advantage of the host as an implicit synchronization point, but suffered because GPU floating-point atomics perform very poorly. Additionally, other issues like thread divergence and register pressure also contribute to fluidanimate's overall poor performance. The large gap in performance obtained on the two systems also shows the instability of GPU optimizations when applied to different GPUs. Finally, swaptions performed poorly across all input sizes. This is likely due to the low amount of parallelism it has, as well as the its high cost of transferring memory between the CPU and GPU and the high cost of accessing GPU global memory when there aren't sufficient threads to hide the latency of accessing memory. In addition, swaptions suffers from thread divergence, register pressure, and dynamic loop bound issues.

Some of the bottlenecks we encountered seemed to stem from fundamental limitations of the algorithms, which cause our GPU implementations to perform poorly. Many of these algorithms were designed to operate with only a few threads, whereas GPUs operate best when there are thousands of threads to hide the latency of memory accesses. Thus, our approach of incrementally modifying the PARSEC benchmarks to execute on GPUs may not have been the ideal approach. It may be the case that we would be able to achieve better performance by starting from scratch and designing a heavily multithreaded algorithm that fits the problem specifications would be a better approach.

Additionally, the poor performance we obtained are partially due to implementation-specific issues in CUDA, such as the lack of global synchronization and how memory transfers between the CPU and GPU are structured. Because CUDA does not offer a way to pipeline memory transfers such that one could be transferring data to one part of a buffer while reading from a different part of the same buffer, this is a roadblock to increasing performance. However, there are some cases where writing code for a GPU was simpler than writing the same code on a CPU, such as using a broadcast-and-wait when a single thread block is used.

Overall, we found that, in general, CMP benchmarks do not map uniformly well to GPUs. While getting code to return functionally correct answers was not extremely difficult, significant optimizations are often required to achieve high performance GPU programs, especially for programs that aren't explicitly data parallel. Unfortunately, we found that this process is often non-trivial and sometimes non-intuitive. Ryoo, et. al. report similar conclusions from their study [45]. CMP benchmarks represent a potential new use for GPUs, but they are unable to execute efficiently on current GPUs. There are two approaches that can help address this problem. First, significant algorithmic changes can help execute applications like this on GPUs with high performance. Second, we can make changes to the GPU architecture to help alleviate the bottlenecks of these applications. In this thesis we focus on identifying what features of the GPU need to change in order to enable this new GPU use.

Chapter 5

Designing a Suite of Challenging Benchmarks for GPUs

In this chapter, we expand on the results from Chapter 4 and find a general set of applications that do not execute well on current GPUs. Our goal is to identify what the truly difficult and poor performing GPU workloads are and what makes them challenging. The applications we identify, which we call *challenge benchmarks*, enable potential new uses of GPUs. These applications are often limited by program parallelism, control flow issues, and stalls due to memory accesses. Without sufficient parallelism (threads), GPU workloads cannot exploit the massively parallel architecture. Control flow issues can further limit performance if the workload does follow the GPU SIMD paradigm. Finally, the performance impact of long memory latencies and limited memory bandwidth is significant if it is not hidden by concurrent threads. To identify specific bottlenecks to address in future GPU research, we analyze our set of challenge benchmarks and identify their performance limiting factors.

The remainder of this chapter is organized as follows. In Section 5.1, we identify a set of challenging benchmarks. Next, in Section 5.2, we analyze the performance limiting factors for these challenge benchmarks. Finally, we conclude in Section 5.3.

5.1 Benchmarks for Future GPU Architectural Research

We begin our search for challenge benchmarks with a survey of the benchmarks listed in Table 5.1. We obtained these benchmarks from the GPGPU-Sim suite [3], Rodinia suite [15], PARSEC suite [8], and other suites [18, 21]. We limited our search to CUDA benchmarks as CUDA is supported by both GPU hardware and the GPGPU-Sim simulator [3] and is one of

Table 5.1: Benchmark effective IPC (challenge benchmarks shaded)

	Benchmark	Abbreviation	Input Size Used	Effective IPC
GPGPUsim	BlackScholes	BLK	400M	202
	AES Cryptography	AES	256KB	184
	StoreGPU	STO	192KB	184
	Ray Tracing	RAY	256x256 image	159
	Coulumbic Potential	CP	200 atoms, 256x256	147
	Libor Monte Carlo	LIB	15 options, 4K paths	129
	3D Laplace Solver	LPS	100x100x100	104
	Fast Walsh Transform	FWT	8M elements	—
	gpuDG	DG	N=6, 2 steps	—
	Weather Prediction	WP	10 timesteps	43.2
	Neural Network	NNW	28 digits	12.4
	N-Queens Solver	NQU	10 queens	8.5
	Mummer	MUM	200 queries/30K entries	3.8
	Breadth First Search	BFSG	64K nodes	3.7
Rodinia	Cellular Automata	CELL	1024x32, 8×	228
	Kmeans	KM	494K objects	193
	Hotspot	HOT	512x512x2	191
	Leukocyte	LKT	10 frames	180
	PathFinder	DYN	8192x8192x32	169
	SRAD 2	SRAD2	402x458, 10×	158
	Gaussian	GAU	dim = 512	139
	LU Decomposition	LUD	dim = 256	135
	ParticleFilter	PFT	10000 particles, 10 frames	116
	Streamcluster	SC	65K points	90.5
	SRAD 1	SRAD	402x458, 10×	86.9
	Backprop	BPP	64K elements	82.5
	HW Tracking	HWT	10 frames	81.2
	Heartwall	HW	5 frames	81.2
	Comp Fluid Dyn	CFD	97K data points	74.9
	Breadth First Search	BFS	1M nodes	44.5
	Nearest Neighbor	NNB	42K records, 4 files	7.4
	Needleman-Wunsch	NW	4K elements	4.0
	Myocyte	MYO	100 ms, 100×	1.6
PARSEC	Fluidanimate	FLD	100 frames, 4K cells	0.2
	Swaptions	SWP	64 swaptions, 20K sims	3.8
Other	S3D [18]	S3D	4K points	—
	Mummer++ [21]	MMP	200 queries/33K entries	0.3

the most widely used GPU programming languages. GPGPU-Sim and Rodinia are commonly used CUDA benchmark suites. PARSEC is a heavily used multicore benchmark suite that we discussed in the previous chapter. Rather than making wholesale algorithmic changes, our implementations modify existing algorithms for GPU execution. The remaining two benchmarks are cited in previous work as challenging workloads [19].

In Table 5.1, we list these benchmarks, input sizes used, and our observed *effective IPC*. The effective IPC is the IPC using only useful instructions per cycle (e.g., ignoring masked instructions due to warp divergence) and is found using GPGPU-Sim with a Tesla C1060-like configuration (see Table 2.1). The peak IPC for this system is 240. We classify any benchmark with overall or per kernel effective IPC less than 40% of the peak (96) as a *challenging benchmark*; these benchmarks are shaded in the table. A third of the GPGPU-Sim benchmarks and just over half of the Rodinia benchmarks are classified as challenging. Performance information for DG, FWT and $S3D$ are missing due to GPGPU-Sim runtime or compilation errors; we include $S3D$ in the challenge benchmark suite based on hardware profiling results. Finally, our study focuses exclusively on kernels executed on the GPU – CPU work is ignored.

5.2 Analyzing Challenging GPU Benchmarks

5.2.1 Overview

In this section, we perform a detailed characterization of the challenge benchmarks using the GPGPU-Sim simulator [3]. By analyzing this data, we identify bottlenecks across the benchmarks due to parallelism, control flow, and memory limitations.

Our analysis includes a detailed characterization and data analysis using the GPGPU-Sim simulator. Together, this allows us to identify and analyze performance bottlenecks in the challenging benchmarks. Our study focuses on an Nvidia Tesla C1060-like GPU, although we expect the conclusions to hold for similar GPU architectures. The C1060 has 30 SMs with eight SPs each for a peak compute capability of 240 instructions per cycle (IPC). Our simulations use the GPGPU-Sim simulator (version 2.1.1b), which produces detailed statistics while modeling

Table 5.2: Detailed challenge benchmark analysis (kernels in numeric-alpha order, except NNW, which is in layer order).

			Available Parallelism			Control Flow		Memory			
	Effective	**Kernel**		*Threads per*	*Total*	*Avg. Threads*		*Accesses*	*DRAM BW*	*Stalled*	**Anticipated**
Kernel	**IPC**	**Time**	*Blocks*	*Block*	*Threads*	*per Warp*	*Serial*	*Coalesced*	(*GB/s*)	*for Memory*	**Bottlenecks**
BFS_1	4.87	92%	1954	512	1000448	10	25%	56%	70	76%	WP, ST
BFS_2	104.28	8%	1954	512	1000448	27	4%	97%	34	33%	LAT
BFS_G	3.69	100%	256	256	65536	10	25%	50%	69	66%	WP, ST
BPP_1	12.07	66%	4096	256	1048576	11	0%	88%	23	76%	WP,BW
BPP_2	132.94	34%	4096	256	1048576	12	0%	93%	41	11%	—
CFD_1	72.02	89%	506	192	97152	31	0%	76%	93	64%	BW
CFD_2	173.56	2%	506	192	97152	32	0%	94%	80	9%	BW
CFD_3	100.79	0%	506	192	97152	32	0%	94%	65	5%	—
CFD_4	82.09	9%	506	192	97152	32	0%	94%	91	62%	BW
FLD_1	0.81	0%	19	256	4864	14	11%	7%	47	96%	LAT, WP, BP, ST
FLD_2	0.16	40%	32	256	8192	3	39%	4%	14	—	LAT, BP, WP, ST
FLD_3	1.49	0%	32	256	8192	8	3%	13%	67	88%	LAT, WP, BP
FLD_4	0.12	58%	32	356	8192	3	51%	3%	13	40%	LAT, WP, ST, BP
FLD_5	1.22	0%	19	256	4864	13	12%	7%	43	94%	LAT, WP, BP
FLD_6	2.49	0%	19	256	4864	19	7%	94%	25	79%	WP, BP
HW	81.17	100%	51	512	26112	25	1%	91&	86	26%	BW, BP
HWT	81.22	100%	51	512	26112	25	1%	91%	86	26%	BW, BP
MUM	3.75	100%	196	256	50176	8	37%	77%	52	58%	WP, ST
MMP	0.28	100%	1	256	256	8	26%	30%	4	44%	BP, WP, ST
MYO	1.60	100%	4	32	128	25	0%	0%	14	91%	BP, LAT
NNW_1	42.59	3%	168	169	28392	27	0%	90%	64	65%	LAT
NNW_2	11.96	19%	1400	25	35000	25	0%	83%	83	91%	BW
NNW_3	0.12	78%	2800	1	2800	1	100%	0%	80	47%	TP, WP, ST, BW
NNW_4	0.11	1%	280	1	280	1	100%	0%	68	44%	TP, WP, ST
NNB	7.40	100%	938	16	15008	16	0%	22%	98	86%	LAT, WP, BW
NQU	8.53	100%	256	96	24576	26	6%	90%	0	43%	ST
NW_1	4.14	49%	1 to 127	16	16 to 2032	11	5%	83%	6	82%	WP, BP, TP
NW_2	3.91	51%	1 to 127	16	16 to 2032	11	5%	83%	5	82%	WP, BP, TP
SC	90.52	100%	128	512	65536	30	5%	93&	61	46%	BW
$SRAD_1$	205.02	0%	450	512	230400	31	0%	94%	41	2%	—
$SRAD_2$	207.52	0%	450	512	230400	32	0%	94%	57	4%	—
$SRAD_3$	53.01	41%	450	512	2304000	18	17%	93%	7	3%	WP, ST
$SRAD_4$	116.52	32%	1 or 450	512	512 or 2304000	32	0%	42%	67	76%	BP
$SRAD_5$	91.69	21%	450	512	2304000	32	0%	93%	78	57%	BW
$SRAD_6$	98.47	6%	450	512	2304000	32	0%	94%	89	54%	BW
SWP	3.78	100%	1	512	512	21	1%	94%	15	18%	BP, WP
WP	43.22	100%	72	64	4608	25	3%	83%	55	65%	TP
	KEY	Available Parallelism	⇒	TP: Threads per Block,		BP: Blocks per Kernel					
		Control Flow	⇒	WP: Parallelism within Warp,		ST: Serial Execution					
		Memory Accesses	⇒	BW: Memory Bandwidth,		LAT: Memory Latency					

the general-purpose functionality of GPUs, including SMs, SPs, registers, memory, memory controllers, interconnect, and local, shader, texture, and constant memory.

5.2.2 Characterization

In Table 5.2, we present the detailed workload characterization for each kernel in the challenging benchmarks using data from the GPGPU-Sim simulator. The first three columns give general information about the kernel: name, effective IPC, and percent of GPU time spent in that kernel. Note that some kernels have IPCs greater than 96; we include all kernels from a challenging benchmark even if the particular kernel performs well. The next eight columns are divided into three sets: Available Parallelism, Control Flow, and Memory. Section 5.2.3 discusses these columns in detail, and is organized similarly. The last column gives our diagnosed bottlenecks based on intuition from the upcoming data analysis.

5.2.3 Data Analysis

The following likely GPU bottlenecks are included in Table 5.2, with abbreviations listed in the table key:

5.2.3.1 Available Parallelism

GPUs achieve high performance by running many concurrent threads on their massively parallel architecture, but the total number of threads can be limited by the number of blocks in the kernel (BP) or the number of threads per block (TP). Block and thread level parallelism is limited by the fraction of the algorithm that has been parallelized and the problem size. In our table, we consider a kernel parallelism limited if there are fewer than ten thousand total threads (each SM is less than half full), and observe that 12 of the 38 kernels are limited by available parallelism.

5.2.3.2 Control Flow

The single-instruction, multiple-thread (SIMT) architecture of GPUs makes control flow divergence a limiting factor for performance. We quantify the impact of thread divergence by measuring the average number of active threads in a warp over all warp issues. We further measure the average number of warp issues with only a single thread, which indicates serial

execution, synchronization, atomic operations, or extreme thread divergence. Nineteen of the kernels have fewer than 25 active threads per warp (WP) and twelve of them have more than 10% of their issue cycles with only a single active warp (ST).

5.2.3.3 Memory Accesses

Limited caching and heavy cache contention make GPUs dependent on many accesses to main memory, and the long latencies may not always be hidden by the heavy multi-threading if parallelism is limited or there are many memory accesses. We observe that ten of the kernels use more than 70% of the total 102 GB/s of memory bandwidth (BW). It is important to note that DRAM performance can slow with more than 70% utilization due to queuing effects and memory access bursts. Further, we suspect that nine benchmarks with few coalesced memory accesses and many stalls for memory accesses are slowed by the long latency of memory accesses (LAT).

5.2.3.4 Applying Bottlenecks to the Benchmarks

Given these bottlenecks, we can examine the results from Table 5.2 and look at how these bottlenecks apply specifically to individual benchmarks. We present a few examples here:

- $fluidanimate$: In the previous chapter, we discussed some of the bottlenecks that our implementation of fluidanimate faced: thread divergence, register pressure for larger kernels, and poor performance of atomics. The results from our study with GPGPU-Sim confirm these findings and point to several additional bottlenecks. First, the limited number of threads in all of the kernels but the last one cause memory latency to be an issue; this is not surprising because the kernels are not embarrassingly parallel. Additionally, the amount of parallelism available with a warp and within a block is suboptimal. This is likely due to the atomics serializing execution and the thread divergence.

- WP: WP only uses 4K threads and 64 threads per block, so it makes sense that this benchmark is limited by the amount of parallelism within a block. On GPUs, we want

to have as many threads per block as possible to help hide memory latency and stalling; we also want to have more than 4K total threads if possible, which points to the need for more parallelism within the benchmark.

- BFS (Rodinia): The Rodinia version of BFS uses 2 separate kernels and iteratively moves down a graph in breadth-first fashion. Each iteration of the loop represents one level of the graph being examined. Thus, it makes sense that we are limited by the serialization – if we are only examining a few nodes each level of the graph, then the other threads within a warp will be forced to execute separately. Additionally, because only a few threads are performing useful work, we can't easily hide the latency of main memory access and we don't have much parallelism within a warp to exploit.

The key takeaway here is that we are able to classify the bottlenecks for these applications based on simulation studies. This should help guide future GPU architectural research, by exposing the bottlenecks causing degraded performance.

5.2.4 Limitations

Our technique's limitations include our use of freely available CUDA benchmark implementations, potentially unoptimized algorithms, and simulators. For many statistics, `computeprof` only has counters on a single SM. Kernels with limited parallelism or non-steady state behavior are not well profiled with counts from a single SM, and so using a simulator allows us to collect a richer and more representative set of data. From a workload perspective, we acknowledge that the benchmarks could potentially be rewritten to be less challenging in the future, especially if algorithms are designed specifically to exploit the GPUs architectural features.

5.3 Summary

In this chapter, we have identified a suite of benchmarks, including some of the CMP benchmarks from the previous chapter, that represent new uses for GPUs, but which perform poorly on current GPUs. We also characterize these challenge benchmarks and find their performance

bottlenecks. The performance bottlenecks for these benchmarks are dispersed across memory, control flow, and parallelism limitations. The need for higher performance exists for at least half of the benchmarks in common suites, but there is no single architectural feature to focus on for that improvement. This means that there are multiple avenues GPU architects must explore to allow these challenge benchmarks to execute well on future GPUs.

Chapter 6

Conclusion

As GPU architectures become more programmable, new uses for the GPU within non-graphics applications are becoming possible. Previous work has highlighted how many general-purpose, non-graphic applications, especially scientific applications, have been ported to GPUs. These applications are usually embarrassingly parallel and utilize GPU architectural features such as shared memory and fast hardware transcendentals. In this thesis, we've highlighted how GPUs can be used in several new ways that they have not been previously used for. For example, previous work largely ignores the texture unit, especially its internal interpolation feature. Previous work has also not looked at how general-purpose CMP benchmarks might perform on GPUs. If GPUs are truly going to become a general-purpose architecture, they will need to be able to execute programs like CMP benchmarks with high performance.

There are some limitations to our studies in this thesis, including potentially unoptimized algorithms and simulators. For example, in Chapter 4, we prioritized the use of the current algorithms in the PARSEC benchmarks over wholesale algorithmic changes. We opted to incrementally change the algorithms to execute on GPUs instead of making complete overhauls of the algorithms. While this strategy is better from an implementation perspective, we may be missing the potential of those benchmarks to provide high performance on GPUs with different algorithms that are better suited to the GPU architecture. Additionally, we did not retest or re-implement our algorithms when newer versions of CUDA became available; it is possible that our results would be different if they used these updated versions.

By examining the new applications we have implemented on GPUs, especially the applications that did not perform well, we were able to identify what bottlenecks are preventing

these applications from achieving high performance on GPUs. We hope that these findings will guide future GPU architectural research.

Overall, we demonstrate several new uses for the GPU. These uses are enabled by the increased general-purpose nature of the GPU. By finding new uses for the GPU, we demonstrate how some underutilized components of the GPU architecture, such as texture memory interpolation, can be used to dramatically improve performance for non-graphics applications. By identifying sources of bottlenecks that prevent programmers from obtaining high performance for general-purpose, non-graphics applications on GPUs, we help direct future directions for GPU architectural research and enable more new uses for GPUs.

LIST OF REFERENCES

[1] Nvidia sdk 2.3. `http://developer.nvidia.com/object/cuda_2_3_downloads.html`.

[2] Nvidia sdk 3.1. `http://developer.nvidia.com/object/cuda_3_1_downloads.html`.

[3] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS 2009.*, pages 163 –174, April 2009.

[4] Christoph Bennemann, Mark W. Beinker, Daniel Egloff, and Michael Gauckler. Teraflops for games and derivatives pricing. *WILMOTT Magazine*, June 2008.

[5] A. Bernemann, R. Schreyer, and K. Spanderen. Pricing structured equity products on gpus. In *2010 IEEE Workshop on High Performance Computational Finance (WHPCF)*, pages 1 –7, November 2010.

[6] A. Bernemann, R. Schreyer, and K. Spanderen. Accelerating exotic option pricing and model calibration using gpus. In *Social Science Research Network*, pages 1–19, February 2011.

[7] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM New York, NY, USA, 2008.

[8] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[9] Christian Bienia, Sanjeev Kumar, and Kai Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Proceedings of the 2008 International Symposium on Workload Characterization*, September 2008.

[10] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[11] F. Black and M. Scholes. The pricing of options and corporate liabilities, 1973.

[12] P.D. Bryan, J. Beu, T. Conte, P. Faraboschi, and D. Ortega. Our many-core benchmarks do not use that many cores. In *In Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, pages 16–23, July 2009.

[13] I. Buck, K. Fatahalian, and M. Houston. Gpubench. `http://graphics.stanford.edu/projects/gpubench/`.

[14] S. Che, M. Boyer, M. anoyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, October 2009.

[15] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC 2009.*, pages 44 –54, 2009.

[16] C.J. Choi, G.H. Park, J.H. Lee, W.C. Park, and T.D. Han. Performance comparison of various cache systems for texture mapping. In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, volume 1, pages 374 –379 vol.1, may 2000.

[17] J Christensen-Dalsgaard, T Arentoft, T M Brown, R L Gilliland, H Kjeldsen, W J Borucki, and D Koch. Asteroseismology with the kepler mission. Technical Report arXiv:astro-ph/0701323, Nov 2009. Comments: Proc. Vienna Workshop on the Future of Asteroseismology, 20 - 22 September 2006. Comm. in Asteroseismology, Vol. 150, in the press.

[18] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 63–74, 2010.

[19] Wilson W.L. Fung and Tor M. Aadmodt. Thread block compaction for efficient simt control flow. In *HPCA-17*, 2011.

[20] Luigi Genovese. Graphic processing units: a possible answer to high performance computing? In *4th ABINIT Developer Workshop*, 2009.

[21] Abdullah Gharaibeh and Matei Ripeanu. Size matters: Space/time tradeoffs to improve gpgpu applications performance. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, 2010.

[22] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[23] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[24] Naga K. Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 611–622, New York, NY, USA, 2005. ACM.

[25] Ziyad S. Hakura and Anoop Gupta. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 108–120, New York, NY, USA, 1997. ACM.

[26] T. Hamada and T. Iitaka. The Chamomile Scheme: An Optimized Algorithm for N-body simulations on Programmable Graphics Processing Units. *ArXiv Astrophysics e-prints*, March 2007.

[27] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[28] G. Handler. β Cephei and Slowly Pulsating B stars as targets for BRITE- Constellation. *Communications in Asteroseismology*, 152:160–165, January 2008.

[29] D. Heath, R. Jarrow, and A. Morton. Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation. *Econometrica: Journal of the Econometric Society*, pages 77–105, 1992.

[30] J.C. Hull. *Options, futures, and other derivatives*. Pearson Education India, 2008.

[31] Homan Igehy, Matthew Eldridge, and Pat Hanrahan. Parallel texture caching. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '99, pages 95–106, New York, NY, USA, 1999. ACM.

[32] Craig Kolb and Matt Pharr. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 45: Options Pricing on the GPU, pages 719–731. Addison Wesley, December 2006.

[33] Thierry Lanz and Ivan Hubeny. A grid of non-lte line-blanketed model atmospheres of o-type stars. *The Astrophysical Journal Supplement Series*, 146(2):417, 2003.

[34] Thierry Lanz and Ivan Hubeny. A grid of nlte line-blanketed model atmospheres of early b-type stars. *The Astrophysical Journal Supplement Series*, 169(1):83, 2007.

[35] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[36] Wei Li, Xiaoming Wei, and Arie Kaufman. Implementing lattice boltzmann computation on graphics hardware, 2003. 10.1007/s00371-003-0210-6.

[37] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, march-april 2008.

[38] David Luebke and Greg Humphreys. How gpus work. *Computer*, 40:96–100, 2007.

[39] M. Maintz, T. Rivinius, S. Štefl, D. Baade, B. Wolf, and R. H. D. Townsend. Stellar and circumstellar activity of the Be star omega CMa. III. Multiline non-radial pulsation modeling. *Astronomy and Astrophysics*, 411:181–191, November 2003.

[40] Hctor Antonio Villa Martnez. Texture caching. `http://www.csl.mtu.edu/~havillam/AC/TextureCache.pdf`.

[41] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[42] Fredrik Nord. Monte carlo option pricing with graphics processing units. Master's thesis, KTH Royal Institute of Technology, October 2010.

[43] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. Streaming-data algorithms for high-quality clustering. In *Proceedings of 18th International Conference on Data Engineering*, pages 685–694, 2002.

[44] I. Roxburgh and C. Catala. The plator consortium. *Communications in Asteroseismology*, 150, 2007.

[45] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008.

[46] Karthikeyan Sankaralingam, Richard Townsend, and Matthew D. Sinclair. Grassy: Leveraging gpu texture units for asteroseismic data analysis. In *Proceedings of the 2010 GPU Technology Conference (GTC 2010)*, September 2010.

[47] Rahul Sathe and Adam Lake. Rigid body collision detection on the gpu. In *ACM SIGGRAPH 2006 Research posters*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[48] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D. Owens. Efficient computation of sum-products on gpus through software-managed cache. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 309–318, New York, NY, USA, 2008. ACM.

[49] Randy Smith, Neelam Goyal, Justin Ormont, Karthikeyan Sankaralingam, and Cristian Estan. Evaluating GPUs for Network Packet Signature Matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2009.

[50] R. H. D. Townsend. Spectroscopic modelling of non-radial pulsation in rotating early-type stars. *Monthly Notices of the Royal Astronomical Society*, 284:839–858, February 1997.

[51] Richard Townsend. Cu-lsp: Gpu-based spectral analysis of unevenly sampled data. In *Proceedings of the 2010 GPU Technology Conference (GTC 2010)*, September 2010.

[52] Richard Townsend, Karthikeyan Sankaralingam, and Matthew D. Sinclair. *Leveraging the untapped computation power of GPUs: fast spectral synthesis using texture interpolation*. Addison-Wesley, 2010.

[53] J. Tlke and M. Krafczyk. Teraflop computing on a desktop pc with gpus for 3d cfd. *International Journal of Computational Fluid Dynamics*, 22:443–456, May 2008.

[54] JA van Meela, A. Arnolda, D. Frenkelb, S.F.P. Zwartc, and RG Bellemand. Harvesting graphics power for MD simulations. *Molecular Simulation*, 34(3):259–266, 2008.

[55] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2010)*, White Plains, NY, USA, March 2010.

[56] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd annual International Symposium on Computer Architecture*, pages 24–36. ACM, 1995.

[57] Wei Xu and Klaus Mueller. A performance-driven study of regularization methods for gpu-accelerated iterative ct, September 2009.

[58] Zhiyi Yang, Yating Zhu, and Yong Pu. Parallel image processing based on cuda. In *International Conference on Computer Science and Software Engineering*, volume 3, pages 198–201, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[59] Simon F. Portegies Zwart, Robert G. Belleman, and Peter M. Geldof. High-performance direct gravitational n-body simulations on graphics processing units. *New Astronomy*, 12(8):641 – 650, 2007.