Porting CMP Benchmarks to GPUs

Matthew Sinclair
Henry Duwe
Karthikeyan Sankaralingam

Technical Report #1693

June 2011

Porting CMP Benchmarks to GPUs

Matthew D. Sinclair Henry Duwe[*] Karthikeyan Sankaralingam
The University of Wisconsin-Madison
Department of Computer Sciences
Vertical Research Group
Contact email:{sinclair}@cs.wisc.edu

Abstract

GPUs have become increasingly popular in recent years, in large part due to their potential to offer a large amount of computational power at low prices. They offer massive potential speedups in program performance, but only if an application maps well to its data parallel programming model. However, it is unclear how to effectively port programs that do not map well onto the GPU programming model. The amount of performance these programs will have on GPUs is also unclear. If GPUs can be shown to execute general-purpose programs with high performance, then it is possible that a GPU-like, many-core architecture could provide the next big increase in general-purpose program performance. In this project, we implemented four benchmarks from the PARSEC CMP benchmarks suite on GPUs – streamcluster, blackscholes, fluidanimate, and swaptions – then analyzed their performance and compared their performance to that of the PARSEC serial and pthreads versions of the same programs. We also investigated what general-purpose programming techniques worked well when mapped to a GPU, what techniques did not work well, and where bottlenecks occurred. We observed that general-purpose programs neither mapped uniformly easily nor well to GPUs in our implementations.

[*]Work performed while author was a student at the University of Wisconsin-Madison.

1 Introduction

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. However, it is neither clear nor obvious if other workloads and applications that aren't data parallel also map well to the GPU.

This paper reports on our work in porting four general-purpose CMP benchmarks from the PARSEC benchmark suite [2, 3] to GPUs using CUDA SDK 2.3 [1] and evaluated their performance. We acknowledge that more current CUDA SDK releases enable some new features for GPUs and note where these new features may improve performance. The benchmarks we implemented were streamcluster, blackscholes, fluidanimate, and swaptions. More details on these benchmarks and their GPU implementations are presented in Sections 2 and 3. 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 paper is as follows. In Section 2 we provide background on the PARSEC benchmark suite, the benchmarks we implemented, and related work in the area. In Section 3 we discuss our GPU implementations of the benchmarks. In Section 4 we outline our testing methodology and system information. In Section 5 we present and analyze our results. Finally, in Section 6 we conclude.

2 Background and Related Work

We chose to implement benchmarks from the PARSEC suite over other CMP benchmark suites like SPLASH-2 [18] because recent work has shown that PARSEC scales significantly better than SPLASH-2 [4, 8]. 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 perform well on GPUs, whereas we wanted to explore both programs that perform well and those which might pose problems for a GPU implementation. Table 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 [14]. 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 [6, 11]. 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 [13]. 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 of 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. Like blackscholes, swaptions also has a PDE that must be solved numerically. Further information on the algorithm can be found elsewhere [10]. 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 [9]. 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 [12]. 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.

3 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 5. 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.

3.1 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 operates 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 [17][1]. This required a significant amount of time and testing. A second issue we encountered was CUDA's lack of kernel support for C++[2]. To solve

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

[2]Nvidia has also improved this over time but it's still an issue.

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

Table 1: Background information on implemented PARSEC CMP benchmarks.

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.

3.2 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 memory, it significantly decreased memory access time of our GPU code because we perform caching on the GPU. As mentioned in Section 2, we also incorporated Kolb and Pharr's optimized mathematical operations to further improve performance by performing more fused multiply-adds and fewer total mathematical operations [12].

3.3 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. 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.[3]

3.4 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 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. 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 Methodology

4.1 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 and accurate comparisons with the PARSEC results, we replaced the PARSEC timing metric with the same metric we used in our GPU implementations.

4.2 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 the systems in Table 2. 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[4]. 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.

5 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)

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

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

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

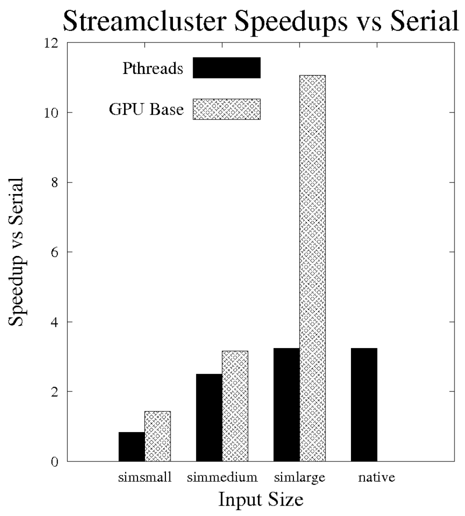
Table 2: Systems Used Specifications

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 and system. 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 [2, 5]. In the next four subsections, we discuss the results for each benchmark.

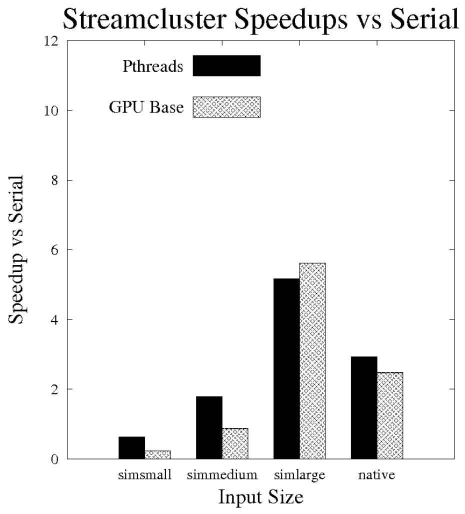
5.1 Streamcluster

Figure 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 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



(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 1: Streamcluster GPU speedups over serial CPU implementation.

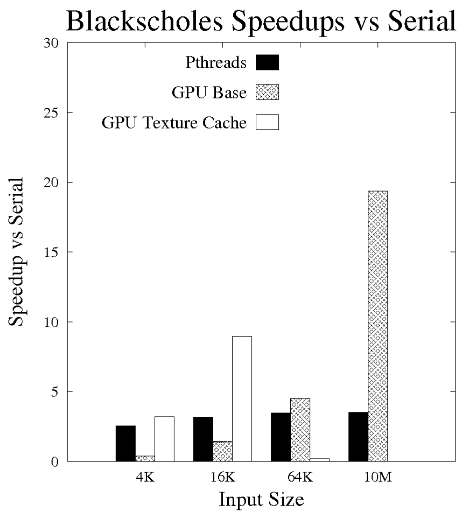
even provide a performance improvement over the serial implementation. These results are more in-line with the results we were expecting as compared to those obtained on the FX 580, because these small test sizes do not perform enough work per synchronization point per thread. However, for the simlarge 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.

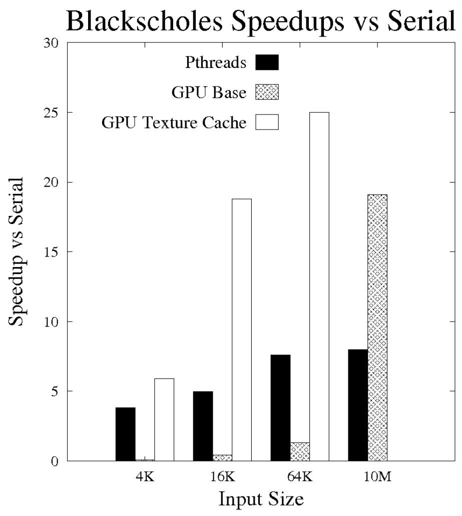
5.2 Blackscholes

Figure 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 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,



(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 2: Blackscholes GPU speedups over serial CPU implementation.

pthreads, and unoptimized GPU implementations immediately. This is because we have significantly decreased our latency to access memory on the GPU 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 use of texture caches that has been explored somewhat previously [16, 19]. 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.

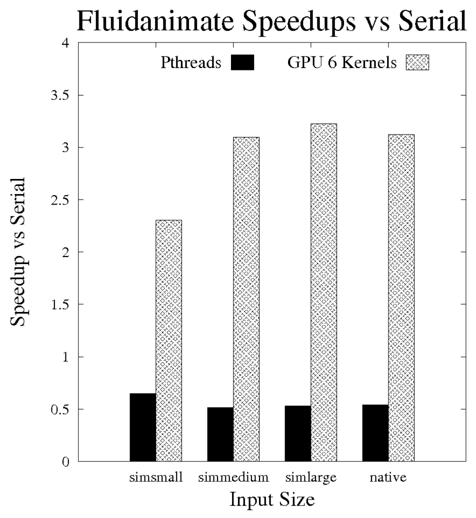
Figure 2b contains the results for blackscholes when it was run on the Tesla C1060 GPU and Xeon 8-core system. The results for the unoptimized version closely mirror those of the unoptimized version on the other system. As the number of threads increase, the performance of the unoptimized version also 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 cache the data and brings it closer to the SPs, which further improves performance for smaller input sizes.

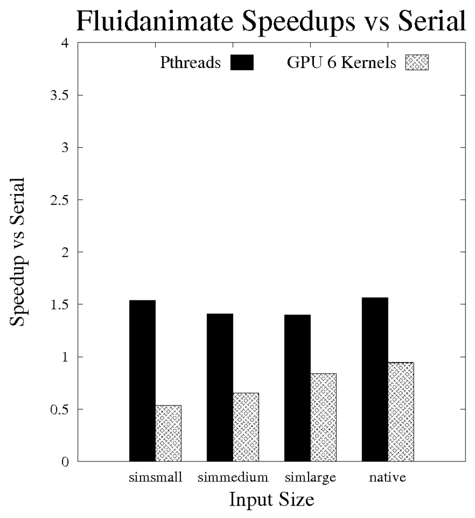
5.3 Fluidanimate

The fluidanimate results in Figures 3a and 3b only include the results for pthreads and the GPU implementation with three to six kernels. We found that the six kernel implementation had the best performance of all of the GPU implementations[5]. 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.

Figure 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 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.



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

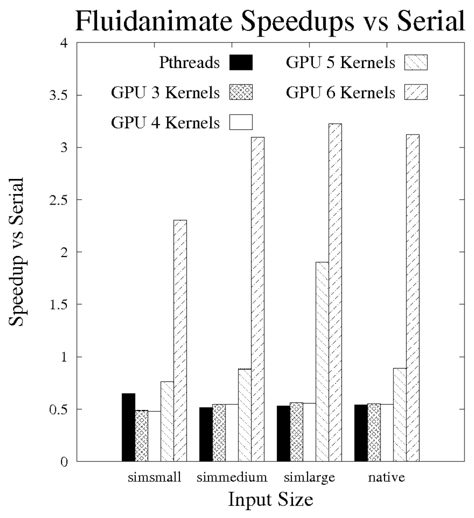


(b) On C1060 GPU and 8-core CPU. The pthreads results use 8 threads. The GPU results use 16K threads.

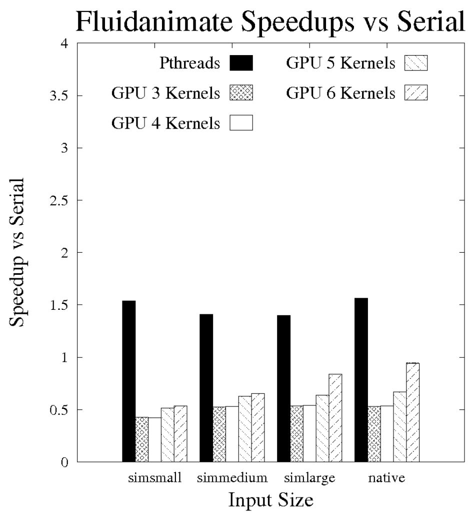
Figure 3: Fluidanimate GPU speedups over serial CPU implementation.

Figure 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

[5]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.

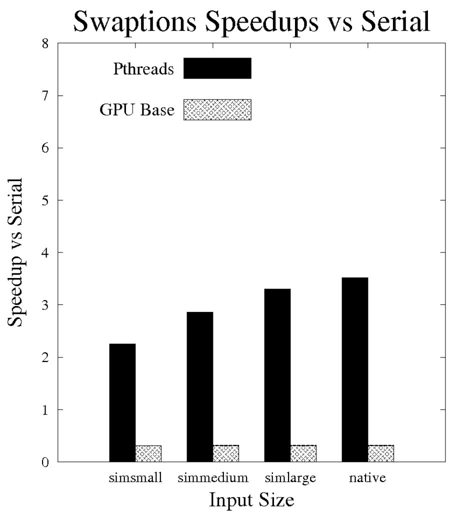
Figure 4: Fluidanimate GPU speedups versus serial CPU implementation for a varying number of GPU kernels.

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.

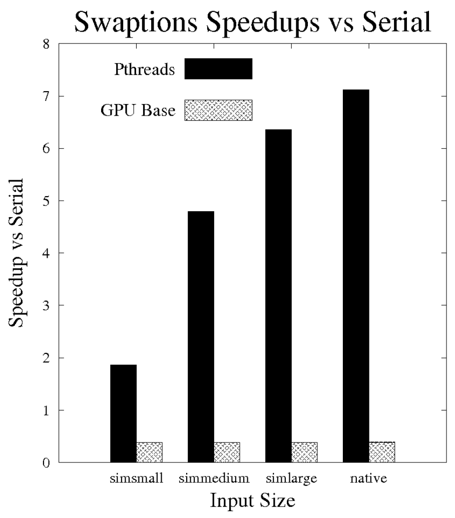
There are several likely causes for fluidanimate's poor performance. GPU optimizations are often specific to a GPU, and may not perform as well on another GPU, this may be a possible cause here. Additionally, this implementation exhibits thread divergence, which significantly decreases performance. It 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.

5.4 Swaptions

Figure 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



(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 5: Swaptions GPU speedups versus serial CPU implementation.

performance further, so they have been omitted.

Figure 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. 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.

6 Conclusion

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 [15]. 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, we can make changes to the GPU architecture to help alleviate the bottlenecks of these applications. Blem et al. have used these benchmarks to help identify what features of the GPU need to change in order to enable GPUs to execute CMP benchmarks with high performance [7]. Second, significant algorithmic changes can help execute applications like this on GPUs with high performance.

Acknowledgements

We thank Mark Hill for suggestions on the implementations of the benchmarks in CUDA during the work done as part of the CS/ECE 757 project (Spring 2010). We thank Richard Townsend for allowing us access to his GPU hardware platforms.

References

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

[2] 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.

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

[4] 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.

[5] 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.

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

[7] Emily Blem, Matthew Sinclair, and Karthikeyan Sankaralingam. Challenge

benchmarks that must be conquered to sustain the gpu revolution. In *Proceedings of 4th Workshop on Emerging Applications and Manycore Architectures (EAMA)*, June 2011.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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