Parallel Algorithm Design and Implementation of Regular/Irregular Problems: An In-depth Performance Study on Graphics Processing Units

by

Steven Solomon

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada

Thesis advisor

Parimala Thulasiraman

Author

Steven Solomon

Parallel Algorithm Design and Implementation of Regular/Irregular Problems: An In-depth Performance Study on Graphics Processing Units

Abstract

Recently, interest in the Graphics Processing Unit (GPU) for general purpose parallel applications development and research has grown. Much of the current research on the GPU focuses on the acceleration of regular problems, as irregular problems typically do not provide the same level of performance on the hardware. We explore the potential of the GPU by investigating four problems on the GPU with regular and/or irregular properties: lookback option pricing (regular), single-source shortest path (irregular), maximum flow (irregular), and the task matching problem using multi-swarm particle swarm optimization (regular with elements of irregularity). We investigate the design, implementation, optimization, and performance of these algorithms on the GPU, and compare the results. Our results show that the regular problem achieves greater performance and requires less development effort than the irregular problems. However, we find the GPU to still be capable of providing high levels of acceleration for irregular problems.

Acknowledgments

I would like to thank my advisor, Dr. Parimala Thulasiraman, for all the guidance and support she has provided throughout my research. I extend these thanks to Dr. Ruppa K. Thulasiram as well, for his added support and encouragement. I am also grateful to my examining committee, Dr. Michael Domaratzki and Dr. Shawn Liu, for providing useful feedback on my thesis and helping me to improve it further. I extend my gratitude to the National Science and Engineering Research Council of Canada (NSERC) for their financial support of my research in the form of an Alexander Graham Bell Canadian Graduate Scholarship award.

Finally, I would like to thank my family for supporting me throughout my pursuit of a Masters degree.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

For decades, the gaming and graphics industries have driven the evolution of the Graphics Processing Unit (GPU). Essentially the only users of GPUs, these industries pushed the performance and parallelism of the GPU to greater heights, year by year. With the introduction of the programmable shader, however, the GPU became open to General Purpose GPU (GPGPU) applications. Now it was possible to execute algorithms that did not belong to the gaming or graphics world on a GPU. With the more recent addition of frameworks such as CUDA and OpenCL, from Nvidia [36] and Khronos Group [14] respectively, developing GPGPU applications became even easier. The result was a surge of interest in GPGPU applications, and an explosion of research into the capabilities and potential of the GPU for general purpose applications.

We originally considered the GPU not only because it was a new architecture for parallel computing, but also because it provides us with a tremendous level of exploitable parallelism and computational performance. Nvidia [33], for example, measures the peak performance of a modern, high-end GPU (GTX 580) at slightly

over 1.5 TFLOPS (Trillions of FLoating Point Operations per Second), whereas a modern high-end multi-core CPU from Intel ranks in at under 250 GFLOPS (Millions of FLOPS). To lend further credence to this performance gap, Suda et al. [50] compared a high-end (at the time) GPU to a single node of the T2K Todai supercomputer (containing four quad-core processors). They concluded that the GPU offers power savings, improved single-precision floating point performance, and ten times lower cost compared to the supercomputer node.

Unfortunately, not all types of algorithms are guaranteed to fully exploit the GPU and harness this tremendous power. We note here two classifications of problems: regular and irregular. Regular problems feature a very structured, predictable nature in terms of both program flow and memory accesses. Consider our matrix-vector multiplication example in Figure 1.1. We know beforehand exactly what computations are required in order to generate the output vector. We know the computational steps of the basic tasks (computing an element in the output vector), and we know exactly how many of these tasks we will require. As a result, we can generally optimize an algorithm for solving this problem for a given architecture easily. In a multi-processor environment, for example, we may simply have each processor compute the resulting elements for a given number of unique rows in the matrix.

In our regular problem example, we note that the operations and data access patterns required to compute any given element in the result vector are exactly the same. We compute each task, or computation of an element in the result vector, in the exact same way, only changing the data we access. As a result, we develop an algorithm with *data-parallel* properties. Data-parallel implies that we can execute the

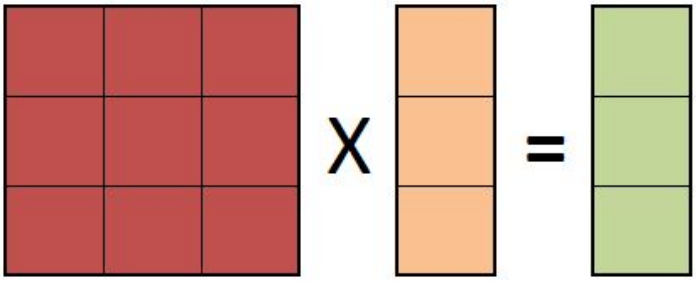

Figure 1.1: An example of a regular problem: matrix-vector multiplication

same instruction across multiple pieces of data in parallel. We provide an example of data-parallel processing in Figure 1.2. Note how each threads "feeds" a unique piece of data through the same instruction in parallel.

In the literature, a large amount of research has been dedicated to studying data-parallel algorithms. These types of algorithms are well suited for the GPU, due to the very data-parallel nature of the architecture which we will discuss further in Chapter 2. Regular, data-parallel algorithms allow one to exploit the thread-level parallelism of the GPU very easily and observe significant performance improvements.

Irregular problems, on the other hand, feature very unpredictable and unstructured properties for their program flow and data access patterns. They feature dynamic changes at run-time. Such problems typically (but not always) use pointer-based data structures such as graphs. With irregular problems, we do not know the interactions between tasks, or even the number of total tasks, ahead of time. Along with these unpredictable computations, these problems further features very unpredictable and unstructured memory accesses.

We use the breadth first search, shown in Figure 1.3, to illustrate the features of an algorithm with irregular properties. Until we reach a vertex and prepare to

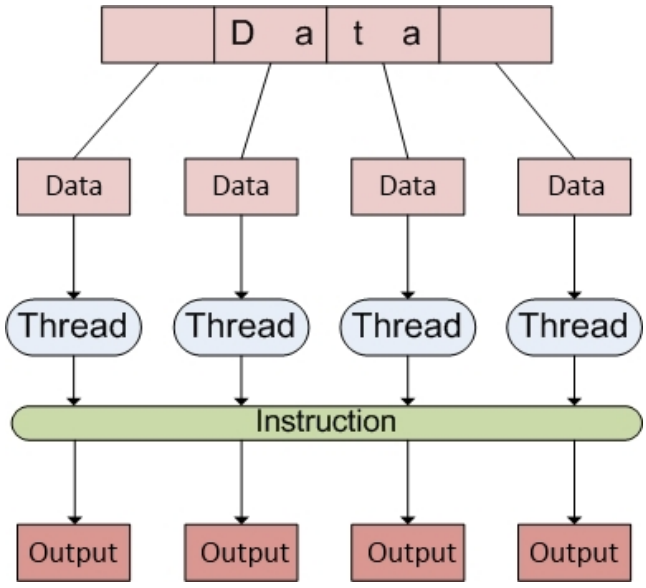

Figure 1.2: An example of data-parallel computations

traverse its edges, we do not know what data we will need next. We do not know how many edges we will traverse until we have completed a full breadth first traversal of the graph, and we do not know the general structure of the vertices and edges at compile time (given an unstructured graph). Furthermore, the path(s) we traverse through the graph may differ depending on which vertex we choose to start with. The data itself defines the structure of the computations and, as a result, we cannot easily optimize this algorithm for the particular nuances of an architecture, especially when it comes to memory accesses and load balancing. This holds true for the GPU, where we see reduced performance with irregular problems due to their unstructured and unpredictable nature.

Irregular problems have been studied extensively for CPU-based architectures. The burden of parallelism, however, has generally been left to smart, automatic compilers rather than the developer. The Galois system from Kulkarni et al. [26], for

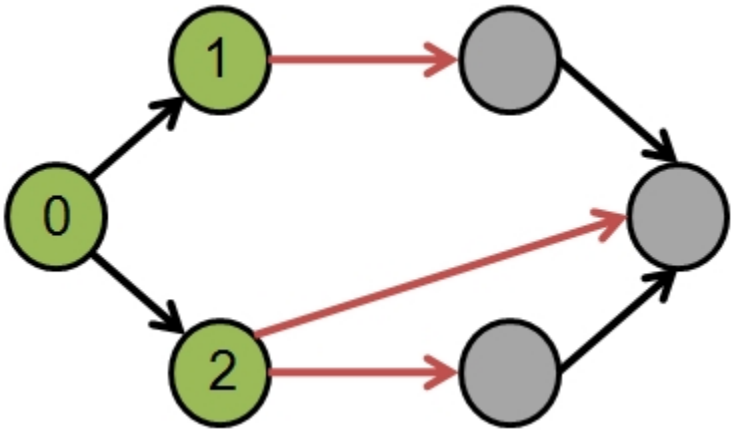

Figure 1.3: An example of an irregular problem: breadth first search

example, investigated the potential of automatic, dynamic (or optimistic, as they refer to it) parallelization of irregular problems. The system uses a set of parallel primitives which encapsulate parallel operations in order to invoke parallelism. The authors test their method against two irregular problems and show promising results. They further mention that the traditional static methods of automatic parallelization are ideal for regular problems, but are entirely unsuitable for irregular problems.

Kulkarni et al. [25] later investigated a number of irregular parallel algorithms using a tool they developed called "ParaMeter", which searches a given algorithm for parallelism and generates parallelism profiles of the algorithm. They then use these profiles to determine where the available parallelism lies in an algorithm, or, perhaps, to investigate if an algorithm contains a reasonable level of exploitable parallelism to begin with. These examples show that as of two years ago, there has been much ongoing investigation into the parallelization of irregular algorithms.

The study of irregular problems on the GPU, however, remains in its infancy. Due to the optimal mapping between regular problems and the GPU, the majority

of existing GPU research focuses on these types of problems. In general, the GPU provides significant performance gains on regular problems when compared to traditional CPU implementations. For example, Preis et al. [39] investigate a few financial algorithms on the GPU that contain regular properties. The authors measure an approximate 80 times performance improvement on the GPU when compared to a sequential CPU implementation for the highest performing algorithm. An earlier paper by Fujimoto [11] investigated dense matrix-vector computations on the GPU and saw up to 32 times better performance on the GPU compared to a sequential CPU implementation.

In this work, we investigate the design, implementation, and performance of algorithms for both regular and irregular problems on the GPU. In order to accomplish this we closely investigated four different problems: lookback option pricing, layered-network construction, the maximum flow problem, and the task matching problem. Within this set of problems we have a regular problem (lookback option pricing), a simple irregular problem (single-source shortest path), a complex irregular problem (the maximum flow problem), and a problem that is computationally difficult and requires heuristics to solve (task matching). The algorithms we choose to solve these problems maintain the same regular or irregular properties of their respective problem. In this respect, we choose the binomial lattice method for pricing lookback options, a breadth-first search to solve the single-source shortest path problem, the MPM maximum flow algorithm for solving the maximum flow problem, and a multi-swarm particle-swarm optimization algorithm for solving the task-matching problem. We believe this diverse choice of problems and algorithms will help to show the dif-

ferences in strategy and effort required to optimize regular versus irregular problems for the GPU, as well as the general performance possible on the architecture.

This, then, is where we make our major contribution with this work. By describing the design and extensively investigating the performance of a variety of problems on the GPU, we build a solid foundation of knowledge regarding what works well on the GPU, and the general capabilities of the architecture. We present the issues associated with regular and irregular problems, discover methods for improving the performance, and provide an in-depth examination of the performance each implemented algorithm achieves on the GPU. By experimenting with a number of algorithms we build a comparison of the major differences between algorithms with regular and irregular properties, such as the effort required for optimization, the performance achieved, and even the general structure of the GPU algorithms. We chose to investigate the GPU not only for its power, but also because we believe that the results from our work will provide greater insight and understanding for the relatively young GPGPU area, an area of parallel computing that continues to grow.

We organize this thesis as follows: Chapter 2 starts with a brief introduction into parallel systems, describes the GPU architecture and CUDA framework, and concludes with a discussion on some parallel computing primitives and speedup. Chapter 3 describes the first problem we investigate, the lookback option pricing problem, using the binomial lattice algorithm. Chapter 4 contains our investigation of the breadth first search algorithm for solving the single-source shortest path problem, and Chapter 5 discusses the MPM algorithm. We complete the chapters dedicated to the problems we investigated with Chapter 6, which provides our work on a multi-

swarm particle swarm optimization algorithm for solving the task matching problem. In Chapter 7 we compare the algorithms we looked at, and discuss what similarities and differences we observed in terms of design, implementation, and performance. Finally, we discuss our conclusions and routes for future work in Chapter 8.

Chapter 2

Parallel Computing and the GPU

In this chapter, we discuss the relevant details of the GPU architecture, the CUDA framework, and two basic parallel algorithms that we use in our work. As we implemented and tested all of our work on an Nvidia GTX 260 GPU (based on the GT200 architecture), all information (and hard numbers) in this chapter pertains to this particular model. Because the GPU is a relatively new architecture in the general purpose parallel algorithms arena, we start with a brief discussion on the two general categories of parallel systems and where the GPU fits in.

2.1 Parallel Systems

We divide the parallel systems used in parallel computing into two major camps, homogeneous and heterogeneous, based on the processing elements contained within the system. Homogeneous systems are the most common systems, and are used very widely to this day. These include hardware like the traditional multicore processor

which contains a number of equivalent, or symmetrical, cores. The system is *homogeneous*, it does not contain any processing elements that differ. Heterogeneous systems, on the other hand, contain processing elements that differ from others within the same system. They are not composed of the same hardware throughout, but contain other processing elements that may be more suited to specialized tasks.

The GPU falls under the heterogeneous systems category. While the GPU itself is composed of a number of identical processing elements, it, alone, does not compose the entirety of the system. The GPU requires a traditional CPU to drive the computational processes we want to execute on it. In effect, the GPU is an accelerator: when present in a system, we design algorithms that the main CPU will schedule for execution on the GPU in order to accelerate tasks. We need both a CPU and a GPU within a system in order to execute algorithms/code on the GPU, creating our heterogeneous system.

Outside of definitions based on processing element composition, Flynn [8], with his taxonomy, splits up parallel computing systems into three categories [1] based on their execution capabilities. The two we are concerned about here are Multiple Instruction Multiple Data (MIMD) and Single Instruction Multiple Data (SIMD). With MIMD, each processing element executes independently of one another. In essence, MIMD allows for each processing element to execute different instructions on different data from one another.

SIMD, on the other hand, involves each processing element executing in lock-step with one another. That is, every processing element executes the same instruction at

[1] Flynn describes four total categories of computing systems, however the Single Instruction Single Data category is a uniprocessing system, rather than a parallel system

the same time as one-another, but executes this instruction on different data. MIMD exploits task-level parallelism (achieving parallelism by executing multiple tasks at one time), while SIMD exploits data-level parallelism (achieving parallelism by taking advantage of repetitive tasks applied to different pieces of data). As we discuss in the next two sections, the MIMD style of system is very different from that of the GPU, which follows the SIMD paradigm.

2.2 CUDA Framework

Prior to discussing the GPU architecture itself, we will cover some details of the CUDA framework. Nvidia developed CUDA [36], or the Compute Unified Device Architecture, in order to provide a more developer-friendly environment for GPU application development. CUDA acts as an extension to the C language, providing access to all of the threading, memory, and helper functions that a developer requires when working with the GPU.

The GPU hardware provides us with a tremendous level of exploitable parallelism on a single chip. Not only does a standard mid-to-high end GPU contain hundreds of processing cores, but the hardware is designed to support thousands, hundreds of thousands, even *millions* of threads being scheduled for execution. CUDA provides a number of levels of thread organization in order to make the management of all these threads simpler. At the top level of the thread organization we have the thread grid. The *thread grid* encompasses all threads that will execute our GPU kernel (the application we run on the GPU). To get to the next level down, the *thread block*, we split up the threads in the thread grid into multiple, equal-sized blocks. The user

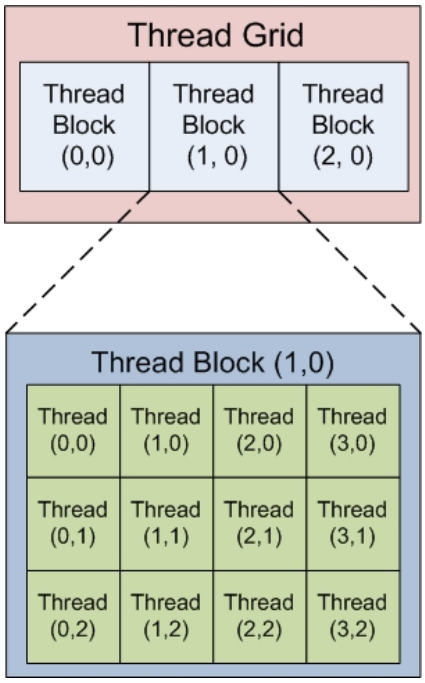

Figure 2.1: 2D Organization of thread grid and thread blocks. [33]

specifies the organization of threads within a thread block and thread blocks within a grid. What this means for a thread block is that we may organize and address threads in a one, two, or three-dimensional fashion. The same holds true for thread blocks within a grid, the user specifies one, two, or three-dimensional organization of the blocks composing the thread grid. Figure 2.1 provides an example of two-dimensional organization of a thread grid and thread blocks.

At the lowest level of thread organization we have the *thread warp*. Equal sized chunks of threads from a thread block form the thread warps for that block. Unlike the size or dimensions of a block/grid, the hardware specifications predetermine the size of a warp, and the threads are ordered in a one-dimensional fashion. For the GT200 (and earlier) architecture, 32 threads form a warp. The hardware issues each

thread within a warp the same instruction to execute, regardless of whether or not all 32 threads have to execute it (we discuss this concept further in Section 2.3). When branching occurs, threads which have diverged are marked as inactive and do not execute until instructions from their path of the branch are issued to the warp. Algorithms for GPUs should therefore reduce branching, or ensure that all threads in a warp will take the same path in a branch in order to maximize performance — a difficult task when working with the unpredictability of irregular algorithms.

Typically, a parallel application will involve some degree of synchronization. Synchronization is the act of setting a barrier in place until some (or perhaps all) threads reach the barrier. In essence, this ensures that the threads in question will all be at the same step in the algorithm immediately after the synchronization point. CUDA provides a few mechanisms for synchronization based around the thread warp, block, and grid. First, each thread in a warp is always synchronized with all the other threads in that same warp as they all receive the same instruction to execute. Secondly, CUDA provides block-level synchronization in the form of an instruction. By using the `__syncthreads()` instruction, threads reaching the instruction will wait until all threads in the thread block have also hit that point.

Unfortunately, CUDA does not provide any mechanisms within a kernel to synchronize all threads in a grid. As a result, we must complete execution of the kernel and rely on the CPU to perform the synchronization. CUDA provides two methods for accomplishing this:

1. Launching another kernel — After invoking one kernel, attempting to launch another will result in the CPU application halting until the previous kernel has

completed execution (effectively "synchronizing" all threads in the thread grid, as they must all have completed execution of the first kernel).

2. Using the `cudaThreadSynchronize()` instruction in the CPU application — Essentially the same as the above, but explicitly controlled by the user. Again, the CPU application will halt here until the previous kernel has completed execution.

To conclude this section we make note of a few memory instructions that CUDA provides, as we use them in our work in order to avoid read-after-write and write-after-write memory hazards. These hazards occurs when multiple threads attempt to perform some computations and store the result into the same location in memory. A second thread may read outdated data while the first thread has not completed writing the *updated* data to the memory location. This second thread then uses the outdated data for its computations, rather than the updated data it should have read (from the first thread). In order to remove these potential errors, we use CUDA's `AtomicX` operations, where `X` represents the actual operation we wish to perform on the data (such as `OR` or `AND`). This atomic memory operation forces each thread executing it to perform their accesses/transactions in a serial fashion, ensuring that each subsequent thread works with the most recent value in that memory location.

2.3 GPU Architecture

In this section we move on to a description of the GPU architecture itself. Further, we connect the architectural details of the GPU with the CUDA framework informa-

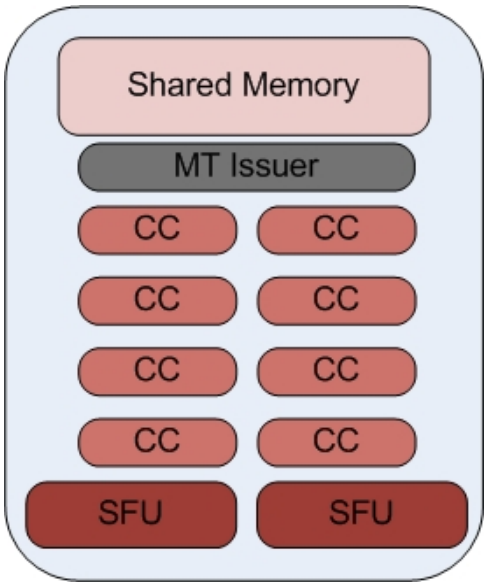

Figure 2.2: General Layout of a Streaming Multiprocessor

tion we discussed in the previous section. We begin with the GPU as a whole, which is composed of two separate units: the core and the off-chip memory, connected by a proprietary and undisclosed interconnection network. At this moment in time we are most interested in the units that compose the core of the GPU, the Streaming Multiprocessors or SMs. As pictured in Figure 2.2, each SM contains eight CUDA Cores, or CCs. These CCs are the computational cores of the GPU, and handle the execution of instructions for the threads executing within the SM. SMs also contain a multi-threaded instruction dispatcher, and two Special Function Units (SFUs) that provide extra transcedental mathematic capabilities.

Execution of instructions on each SM follows a model similar to SIMD, which Nvidia [33] refers to as SIMT, or Single Instruction Multiple Threads. In SIMT, the hardware scheduler first schedules a warp for execution on the CCs of an SM. The hardware then assigns the same instruction for execution across all threads in the

chosen warp — only the data each instruction acts on is changed. This threading model implies that all threads in a warp are issued the same instruction, regardless of whether or not every thread needs to execute that instruction. Consider the case where the threads encounter a branch: half of the threads in a warp take path A in the branch, the other take path B. With SIMT, the hardware will issue instructions for path A to all threads in the warp, even those that took path B. This represents an important concept as threads in a warp *diverging* across different paths in a branch results in a loss of parallelism — each branch is essentially executed serially, rather than in parallel. That is to say, rather than having 32 threads performing useful work, only a subset of the threads do work for path i, while the remaining threads idle, waiting for instructions from their own path.

The importance of the thread block becomes apparent when we discuss them in context of the SM. All of the threads within a thread block must execute entirely within a single SM. This means that we (or the hardware) cannot split up the threads in a thread block between multiple SMs. Multiple thread blocks, however, may execute on a single SM if that SM has enough resources to support the requirements of more than one thread block.

Each of the SMs further contains 16 kilobytes of shared memory. This shared memory essentially acts as a developer-controlled cache for data required during kernel execution. As a result, the responsibility is on the developer to place data into this memory space — there does not exist any automatic hardware caching of data (Nvidia changed this in their Fermi [33] architecture, which introduced a hardware-controlled cache at each SM). Nvidia [35] claims that accesses to shared memory are up to 100

times faster than global memory, given no bank conflicts. As shared memory is split into 16 32-bit wide banks, multiple requests for data from the same bank arriving at the same time cause bank conflicts and, as a result, are serialized. In effect, bank conflicts reduce the overall throughput of shared memory, as some threads must wait for their requested data until the shared memory has serviced the requests from previous threads. Shared memory is exclusive to each thread block executing on a given SM. That is to say, a thread block cannot access the shared memory data from another thread block, even if it is executing on the same SM.

Moving on to the other memory systems present within the GPU we have the global memory. Global memory is the largest memory space available on the GPU and is read/write accessible to all threads. Unfortunately, a significant latency, measured by Nvidia [33] at approximately 400 to 800 cycles, occurs for each access to global memory. Global memory accesses are not cached at any level, which serves to further compound this latency issue. Thus, every access to global memory will result in the same latency hit. The GPU contains, however, some auxiliary memory systems that *are* cached at the SM level. Each SM has access to caches for the constant and texture memory of the GPU. While these two memories are still technically part of global memory (that is, data stored in these memories are stored in the global memory space) their caches help to reduce the latency penalty by exploiting data locality.

Based on what we have learned about the memory systems within the GPU, we clearly want to place an emphasis on exploiting shared memory as much as possible. With comparatively fast access speed and no dependencies on data locality to mitigate high latencies, shared memory represents the most optimal location for storage.

Unfortunately, we run into many situations where the small size of shared memory results in insufficient storage space for the data required at a given moment in time. Due to the stringent size constraints, along with the unpredictable nature of irregular algorithms, our irregular algorithms in particular rely heavily on global memory, while still attempting to exploit shared memory where possible.

While global memory clearly represents a major area of performance loss due to latency, there is one important technique we can use to mitigate the damage: *global memory coalescing*. In order to understand how coalescing works, we must first revisit the idea of a warp. As we described earlier, a thread warp is composed of 32 threads, all of which are given the same instruction to execute. In the worst case, we would expect there to be 32 individual requests to global memory if the instruction in question requires data from global memory. With coalescing, however, we have the ability to reduce the total number of requests down to only two requests in the best case. The reason for this lies with how memory requests are handled at the warp level: they are performed in a *half*-warp fashion. That is, 16 threads request data from memory first, followed by the remaining 16 shortly thereafter. As a result, the best scenario for coalescing combines all memory requests from each half-warp.

In the GT200 architecture, coalescing occurs when at least two threads in one half of a warp are accessing from the same memory segment in global memory. This technique is very powerful and leads to tremendous improvements in the performance of global memory. Unfortunately, data access patterns must be very structured in order to ensure threads will access data from the same memory segment as one another, something that is not guaranteed when working with irregular algorithms. The easi-

est way to achieve this result is ensuring that each thread accesses data from global memory that is one element over from the previous thread's access location. As we will show, we make use of coalescing as much as possible in our algorithms in order to achieve greater memory performance.

We close this chapter off with a brief discussion on the isolation between thread blocks enforced by CUDA. Recall that shared memory is exclusive to a thread block — other threads in other blocks cannot access the shared memory allocated by the block. Furthermore, there are no built-in mechanisms for communication or synchronization between thread blocks [2]. All of these items combine to show one of the main tenets of CUDA: thread blocks are isolated units of computation. Threads within a thread block communicate with one another, but they cannot readily communicate with other thread blocks. Due to this lack of thread-grid-wide communcation, we will show that our algorithm design changes from what it would otherwise be with full communication functionality. In essence, we rely on multiple kernels for the processing of an algorithm, rather than packing all program code into a single kernel.

2.4 Parallel Techniques and Performance Metrics

In this section we describe two of the basic parallel techniques that we use a number of times throughout our GPU implementations. In essense, these techniques form a base for many algorithms in parallel computing, and, true to this, we make use of them in almost every algorithm we investigate. Following a description of these

[2]Of course, the availability of global memory means that there will always exist a method for communication if desired. The latency of global memory coupled with the overhead of potentially thousands (if not more) of threads accessing a single data element (say, as a synchronization flag) results in an entirely unacceptable solution with a tremendous degree of performance degradation.

techniques, we describe the *speedup* performance metric.

2.4.1 Parallel Reduction

A parallel reduction involves reducing a set of values into a single result, in parallel. More formally, given a set of values, $v_1, v_2, ..., v_n$, we apply some associative operator, $\oplus$, to the elements: $v_1 \oplus v_2 \oplus ... \oplus v_n$, resulting in a single value, w. We consider the structure of a parallel reduction to be that of a binary tree. At the leaf nodes, we have the original set of values. We apply $\oplus$ to each pair of leaf nodes stemming from a parent node one layer up the tree (closer to the root node), giving us the value for that parent node. We repeat this process until we reach the root node, providing us with the final result, w.

When we want to parallelize this reduction technique, we first note that layer i of the tree (where the root node is layer 0, and the leaves layer $\log n$) requires the accumulated partial solution values from layer $i + 1$. This requirement results in synchronization; we can compute one layer of the tree in parallel, but we must wait for all threads to complete their processing of nodes in that layer before moving on to the next. Figure 2.3a provides an example of performing an addition reduction (that is, $\oplus = +$) in parallel. Each subsequent array represents the next layer of computations. In the first layer, we have the initial array. In parallel, we add up each pair of elements (four parallel operations in total) and place the results back into the array. The next layer we have two remaining parallel operations which we use to add up the partial sums from the previous layer. Finally, we add the remaining two partial sums together and end up with the final result in element zero of the array

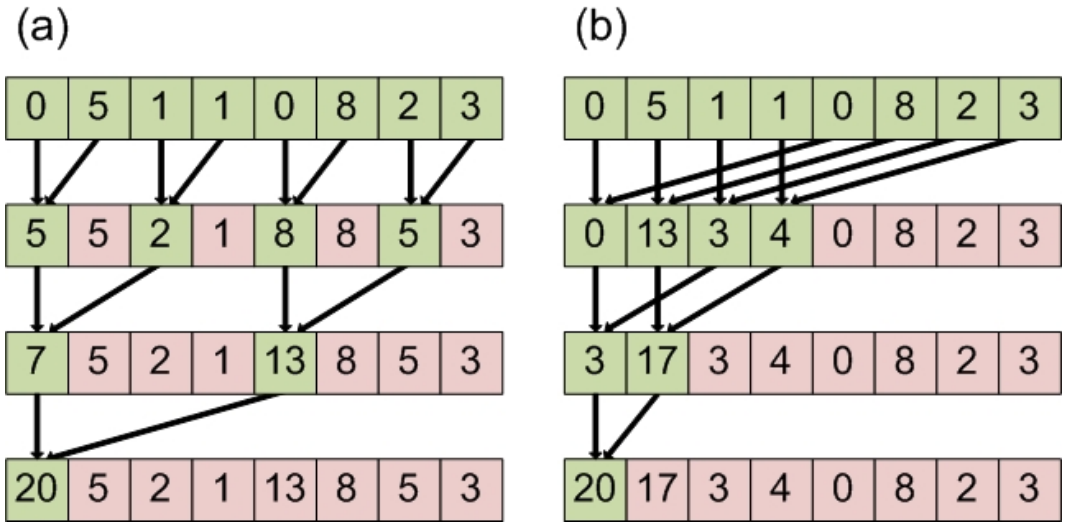

Figure 2.3: Two styles of parallel add reduction on an array of elements.

which contains the total sum of all initial values.

On the GPU, we typically perform a parallel reduction within a single thread block if possible. This allows us to use thread block level synchronization rather than the more costly thread grid level synchronization. As parallelism is plentiful on the GPU, we assign one thread per node in the current layer of the tree. To further optimize this algorithm for the GPU, we do not use the interleaving method shown in Figure 2.3a but, rather, split the layer into halves, and work on one side (pulling data from the other). We show this technique in Figure 2.3b. By working on contiguous areas/halves we ensure coalescing takes place as we read data from global memory, and bank conflicts do not occur as we read data from shared memory.

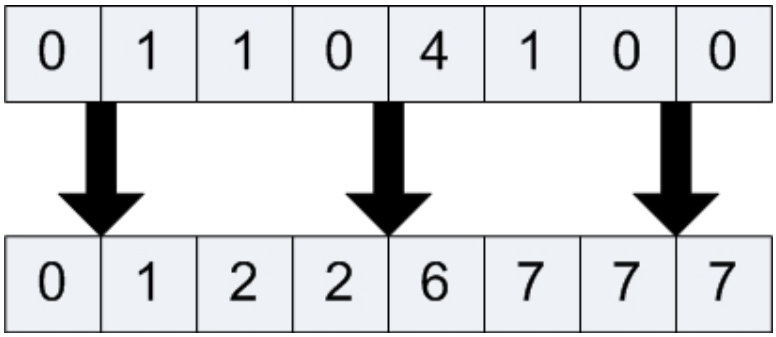

Figure 2.4: Example of an additive scan initial value and result.

2.4.2 Scan

The scan (sometimes referred to as a "parallel prefix sum") operation takes a set of values, $v_1, v_2, ..., v_n$ and an associative operator, $\oplus$, and returns another set of values, $v_1^{'}, v_2^{'}, ..., v_n^{'}$ such that $v_1^{'} = v_1, v_2^{'} = (v_1 \oplus v_2), v_3^{'} = (v_1 \oplus v_2 \oplus v_3), ..., v_n^{'} = (v_1 \oplus v_2 \oplus ... \oplus v_n)$. We provide an example of the resulting data from a scan operation where $\oplus = +$ in Figure 2.4.

While we implement the parallel reduction algorithm ourselves, we use the high performance scan operation provided in the CUDA Data Parallel Primitives (CUDPP) Library [43, 42] in cases where we require it. Sengupta et al. [41] thoroughly describe and explain the design and optimization of the scan algorithm present within CUDPP. The authors describe the build up from a warp-level scan, to a block-level scan, to a grid/global-level scan. They emphasize further optimizations such as the increased workload provided to each thread to better hide global memory latency, and the focus on reducing per-thread register use to ensure that a large number of threads may be active at one time within a block (which serves to further help hide the latency of global memory accesses). We chose this library as it has been extensively optimized

for the GPU and offers all the functionality we require.

2.4.3 Speedup

Speedup is a frequently used performance metric for parallel applications. Basic speedup measures the difference in execution time for the same application when executed on one processing element versus n processing elements. Equation 2.1 provides the formula for speedup, where T_1 is the execution time on one processing element, and T_n the execution time on n processing elements.

$$S(n) = \frac{T_1}{T_n} \tag{2.1}$$

We believe that the standard speedup metric as shown in Equation 2.1 is unfair not only for the GPU, but for parallel systems in general. When developing an application for parallel execution, the developer generally ensures the application is optimized for parallel execution. That is, we assume we have a parallel computing environment and develop the application accordingly. Thus, executing such an application on a single processing element and comparing its execution time against the same application running on n processing elements does not provide very useful results. Instead, we turn to *absolute* speedup, which we believe creates a more suitable environment for comparison.

Unlike speedup, absolute speedup measures the difference in execution time between the parallel application running on n processing elements, and an optimal sequential application. Thus, two applications must be created in order to measure absolute speedup: the parallel application, and a sequential application. In this case, both implementations are designed around the type of system (parallel or sequential)

they will execute on, resulting in a fairer measurement. The formula for absolute speedup replaces T_1 from Equation 2.1 with T_s, where T_s is the execution time of the sequential application.

Throughout this work we use the absolute speedup metric when measuring performance, but refer to it as "speedup" for simplicity.

Chapter 3

Binomial Lattice for Pricing Lookback Options

In this chapter we discuss our work on lookback option pricing for GPUs. The pricing of options represents a fundamental problem in finance and is useful for a variety of purposes from risk analysis to portfolio management. Due to the dynamic and volatile nature of the market, pricing options quickly represents an advantage for the players in the market. As a result, advanced algorithms and hardware platforms are always in demand for real-time pricing.

We chose to investigate this problem on the GPU not only for the reasons described above, but also because of the structured nature of a particular solution to this problem. As we will explain, we investigate the binomial lattice method of pricing options due to its very regular, very structured nature. This binomial lattice algorithm serves as a helpful benchmark to determine the level of effort required for the optimization of a regular problem, as well as a base measure of performance/speedup.

We first lead in with a description of option pricing, followed by the related work in this area, and conclude with details on the mapping and implementation of our algorithm itself.

3.1 Introduction to Option Pricing

Financial options form a contract between two parties. The holder of an option gains the right to exercise the option up to the end of the contract period, otherwise referred to as the maturity date of the option. On the opposite side, we have the writer of an option. While an option grants its holder the right to exercise, an option obligates its writer to follow the whims of the holder. If the holder decides to exercise the option, the writer must comply, and if the holder decides not to exercise the option, the writer cannot force the holder to exercise.

We categorize options into two main types: call options and put options. A call option gives the holder the right to buy an underlying asset (such as a stock) at a price specified at the time of writing the option (the strike price). A put option provides the opposite scenario; it gives the holder the right to sell an underlying asset at a pre-specified price.

Alongside these two types of options are a variety of different styles as well. European and American represent the two "vanilla" (simple) option styles. A European option grants the holder the right to exercise the option only at the maturity date, while the holder of an American option may exercise at any point in time during the contract period. There exist other styles of options as well, generally referred to as exotic options, which include Asian options, chooser options, and the particular

exotic option style we focus on in this Chapter: lookback options.

In terms of exercise style, lookback options may be treated as either a European or American option. For this work, we consider the American-style put lookback option. The payoff of a lookback option is based on the maximum (for a put) or minimum (for a call) value of the underlying asset between the purchase date and the maturity date of the option.

We are, of course, interested in the pricing of these lookback options. We define an option price as the premium associated with the option — the price the holder must pay to the writer in order to purchase the option. In other words, the option price represents the cost associated with the actual purchase of the option. A variety of methods exist for pricing options of various styles, and we explore one such technique in this work: the binomial lattice method for option pricing.

3.1.1 Binomial Lattice Method for Option Pricing

The binomial lattice, as described by Cox et al. [3], is a popular method for approximating the movement of an underlying asset's value, which enables the pricing of an option. A binomial lattice is essentially a binary tree, where the root represents time zero (of the option contact) and the leaf nodes represent the maturity date of the option. The binomial lattice method assumes the value of the underlying asset follows some random walk from the root node to a leaf node of the lattice.

As we move from one node in the lattice to the next, we have a choice of moving "up" or "down". An up movement corresponds to an increase in the value of the underlying asset, S and a downward movement a decrease in the value. We represent

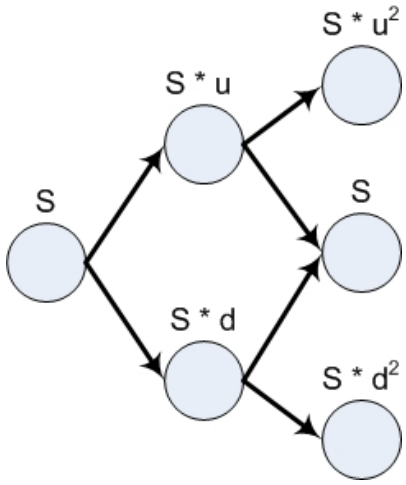

Figure 3.1: Structure of a three-layer binomial lattice.

the increased value via an upward movement with u, and the decreased value via a downward movement with d. The lattice also contains a probability (based on the volatility of the asset) of an upward movement, p, and a probability of a downward movement, $1-p$. We compute a value for p using the formula provided by Haug [16]:

$$p = \frac{e^{r*\delta t} - d}{u - d}$$

We provide a simple example of a three-layer binomial lattice structure in Figure 3.1. We note that this figure does not exactly resemble a binary tree, as we combine nodes at each layer, forming a recombining binomial lattice. This combining of nodes helps to reduce redundant computations as we move through the lattice. The binomial lattice supports this recombining strategy as an upward movement followed by a downward movement in the lattice is equivalent to a downward movement followed by an upward movement. For this reason, we combine these equal nodes at each level in the lattice, resulting in the structure shown in Figure 3.1.

The binomial lattice method offers us a very iterative, structured algorithm to

work with. We start at the leaves of the lattice, and work backwards, layer-by-layer, until we reach the root node (at which point we compute the option price). We label the root node as layer zero, and the leaf nodes as layer $L-1$, where L equals the total number of layers in the lattice. Hence, any node in layer $i+1$ represents one time step in the future from any node in layer i. As we reach a node at each layer we need to compute the option payoff at that node. As an example, for a European-style option we compute the option payoff of a node in layer i of the lattice based on the option payoff value of the two adjacent nodes in layer $i+1$ using the formula provided by Haug [16]:

$$C_{i,j} = e^{-r*\delta t} * (pC_{i+1,j+1} + (1-p)C_{i+1,j-1}) \tag{3.1}$$

where r is the risk-free interest rate, δt is the change in time between each level in the lattice, p is the probability of an up movement in the lattice, and $C_{i,j}$ represents the option payoff for node j at level i in the lattice. Essentially, the option payoff at $C_{i,j}$ is influenced by the up and down movements of the underlying asset at a future time (level $i+1$) as well as the probability of either movement. Of course, we use an alternative formula for the leaf nodes, as they do not have any adjacent nodes at a higher level to use in the option payoff calculation. For leaf nodes, the payoff equation (for a put option) becomes:

$$C_{i,j} = max(S-K, 0)$$

where S is, again, the price of the underlying asset and K is the strike price of the option.

3.1.2 Pricing Lookback Options using the Binomial Lattice

While the basic binomial lattice works for vanilla American and European options, we cannot use it as-is for American-style lookback options. Hull [17] (Technical Note 13) describes how to modify the binomial lattice method in order to support lookback option pricing. Hull suggests a new parameter, $Y(t)$. $Y(t)$ takes the place of the asset price at each node in the lattice, and the author defines it with the following equation: $Y(t) = G(t)/S(t)$ where $G(t)$ represents the maximum asset price achieved up to time t, and $S(t)$ the current asset price at t. We compute the value for $Y(t)$ at each node in the lattice using the following rules:

1. Y is equal to 1 at the root node of the lattice.

2. When we move to the next time step from a node where $Y = 1$, $Y = u$ if the move is an upward move, or 1 if the move is a downward move.

3. When we move to the next time step from a node where $Y = u^m$ for $m \geq 1$, $Y = u^{m+1}$ for an upward move, or $Y = u^{m-1}$ for a downward move.

We provide an example of the modified binomial lattice in Figure 3.2, complete with the associated $Y(t)$, $G(t)$, and $S(t)$ values for each node, given that $u = 1.12$. We orient the lattice in the traditional manner in Figure 3.2, where increasing values at each node (represented by $Y(t)$ instead of the asset value in this case) are preceded by an upward movement in the lattice. However, we note a few key differences from the original binomial lattice. First, we stress that while an upward movement in the original binomial lattice was paired with an upward movement in the asset price, an upward movement in this modification of the binomial lattice represents a *downward*

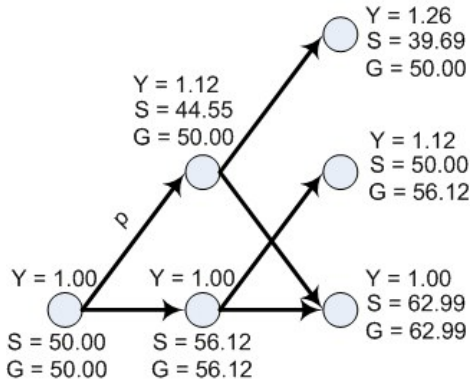

Figure 3.2: An example of the binomial lattice for an American put lookback option and the associated $Y(t)$, $G(t)$, and $S(t)$ values for each node.

movement in the asset price, but an upward movement in $Y(t)$, our new "parameter of importance". This reversal results from how we construct $Y(t)$. In order for $Y(t)$ to increase, $S(t)$ must decrease, as $G(t)$ cannot, by definition, decrease (recall that $G(t)$ represents the maximum asset price observed thus far, hence it can only increase). As a result, a downward movement in the lattice corresponds to an increase in the value of $S(t)$.

As a further example, consider the first choice of movement from the root node, with values $S(0)$ and $G(0)$. Clearly, $G(0) = S(0)$ as the highest asset price we have observed thus far can only be the initial asset price. If we take a downward movement in the lattice (in Figure 3.2 this is simply a horizontal movement as $Y(t)$ will not change), then $S(i)$ increases. This increase implies that $S(i) > S(0)$, and $G(i) = S(i)$, and, therefore $Y(i) = 1$.

The modifications to the binomial lattice do not end with this new parameter, however. We must also modify the option payoff formula for each node to correspond with the formula required for lookback option payoff/pricing. We replace the original

formula defined in Equation 3.1 with one defined in Hull [17], resulting in the following formula:

$$C_{i,j} = \texttt{max}(Y_{i,j} - 1, e^{-r\delta t} * ((1-p)C_{i+1,j+1}d + pC_{i+1,j-1}u)) \tag{3.2}$$

for $j \geq 1$, and

$$C_{i,j} = \texttt{max}(Y_{i,j} - 1, e^{-r\delta t} * ((1-p)C_{i+1,j+1}d + pC_{i+1,j}u)) \tag{3.3}$$

for $j = 1$. All parameters remain the same as in the original binomial approach, and i represents the current time step (which, to reiterate, is further represented by a vertical layer in the lattice). The introduction of u and d represent the proportional movement of the asset price, up or down respectively, at each time step (layer in the lattice).

The first argument inside the `max` function on the right-hand side of each equation represents the local payoff, or option value at that particular node, in terms of Y. The second argument provides the computed option value for node $C_{i,j}$ based on its children nodes at $C_{i+1,j+1}$ and $C_{i+1,j-1}$. Put another way, we weight the local payoffs at nodes $C_{i+1,j+1}$ and $C_{i+1,j-1}$ by their probabilities of achieving these payoffs. As these children nodes represent a time in the future (from $C_{i,j}$), we discount this weighted sum in order to simulate its value at the current layer, i, in the lattice. We then compare the resulting discounted option value with the local payoff at that node, and denote the maximum of these two values as the option value $C_{i,j}$.

Due to the very structured, regular nature of the binomial lattice, we believe it represents a "good fit" for the GPU architecture. We compute each time step/layer in the lattice synchronously, and we know ahead of time what data will be required at

what step in the computations, hence the algorithm contains very regular, predictable properties. This well-suited nature of the algorithm motivated us to choose to work with the binomial lattice method on the GPU. We extended this choice to cover lookback options, as they had not been previously studied on the GPU hardware. Further, due to the ever-present need for faster and faster algorithms in the finance industry, a comparison to sequential CPU will help to show how beneficial a GPU may or may not be for financial applications.

3.2 Related Work

The traditional binomial lattice method for vanilla options has been studied on the GPU in the past. Podlozhnyuk [38] provided one such example of the binomial lattice method on GPUs using CUDA. The author describes a novel technique where they divide the lattice into sub-trees. A thread block then processes each sub-tree independently. Podlozhnyuk explains that this method allows for computations to take place solely within shared memory. The author admits, however, that redundant computations take place as a side effect of splitting the lattice into sub-trees. The improved speed of shared memory, however, may more than compensate for the redundant computations required. Unlike the work of Podlozhnyuk, we target an alternative method for the binomial lattice method that exploits global memory coalescing and removes all redundant computations.

Jauvion and Nguyen [23] further extended the work of Podlozhnyuk to the trinomial lattice, an incremental improvement over the binomial lattice. In their case, however, the authors investigate the potential for pricing one option per thread block,

effectively pricing multiple options in parallel. The authors describe the GPU's precision as being sufficient, and claim that they achieve four decimal points of precision with 1000 time steps. Jauvion and Nguyen further show a speedup of up to 31.6 when parallelizing the pricing of 64 options over 1024 time steps.

As discussed earlier, we base our work around the lookback option pricing method described by Hull [17] (Technical Note 13). To the best of our knowledge, outside of our own work by Solomon et al. [46], there does not exist any other existing work studying the design, implementation, and performance of the binomial lattice method for pricing lookback options on the GPU.

3.3 Option Pricing on the GPU

In this section, we discuss the actual design and implementation of our GPU algorithm. As mentioned before, we do not follow the work of Podlozhnyuk [38] and, instead, choose a design technique that exploits global memory coalescing instead of shared memory and removes the redundant computations associated with dividing the lattice up into sub-trees.

Furthermore, we exploit the synchronous, iterative nature of the binomial lattice in our implementation. Computations start at the leaf nodes of the lattice and progress backwards, layer-by-layer (synchronously), until we reach the root node and compute the final option price. Due to this synchronous, layer-by-layer nature, each invocation of our GPU kernel handles one layer of the lattice. In other words, an invocation of our GPU kernel processes all nodes within a given layer, and, as a result, we require N invocations of the GPU kernel in order to complete the processing of a lattice with

N time steps/layers.

Although we are aware of the general structure of the computations, we have to map these computations to threads on the GPU. As the GPU is a very highly parallel architecture we take a fine-grained approach: we assign one thread to each node in the current layer of the lattice being processed. As we cover a single layer with each kernel invocation, this implies that we require only as many threads as there are nodes within the current active layer.

Within a given lattice layer t, we have $t + 1$ nodes that require processing [1], and, thus, we must launch that exact number of threads for each layer. We note that this implies a reduction in the number of threads launched as the algorithm progresses. Clearly, we will have the most threads, and thus the most parallelism, at the first phase of the algorithm (covering the leaf-node layer of the lattice). Each subsequent phase of the algorithm reduces the number of threads we require by one.

In our algorithm, we store the data in global memory. Since global memory access carries a high latency penalty, we try to avoid as many accesses to global memory as possible. We also reduce the memory footprint of the algorithm by taking into account the structure of the computations. For example, if our problem instance has N time steps, and thus, N layers in the lattice, we initially require a maximum of $N + 1$ nodes allocated in the GPU's global memory. Note that we do not store the nodes for every layer in the lattice into memory. Consider that we work layer-by-layer through the lattice; clearly, by Equations 3.2 and 3.3 we only need data from the previous layer in order to generate the new layer.

[1] Layer t does not contain $2 * t + 1$ nodes, as one might expect, due to the recombining nature of the lattice.

Therefore, we require at least $N + 1$ nodes of the lattice stored in global memory in the worst case. Our actual memory requirements double, however, due to not knowing which threads will execute when during each phase. We cannot simply write over the current node data in global memory as one thread may still require the data from the previous layer when another thread is attempting to write new data into that location for the node in the current layer. As a result, our memory requirements increase to $2 * (N + 1)$ in order to maintain two sets of lattice node data in global memory and double-buffer the data. At each phase, we swap the lattice node data set that we update with the new option payoffs for the given layer.

Finally, we make use of a small amount of shared memory. As we have shown in Figure 3.2, a node requires multiple values from the previous layer, and multiple nodes in the current layer may require data from the same node in the previous layer. In order to avoid redundant accesses to global memory from threads requiring the same data to update their respective nodes, we perform an initial load of the required data for all nodes in a thread block into shared memory. When the algorithm begins, each thread accesses the required data from shared memory, rather than global memory. In this way, we eke out a small amount of extra performance by reducing the overall global memory accesses.

Furthermore, our reading of data in global memory (in order to populate shared memory) makes full use of global memory coalescing. We have each thread store a single value from global memory into the shared memory space. The other piece of data a thread requires is written to shared memory by the thread's neighbor. Figure 3.3 provides an example of this data access pattern — each thread pushes one

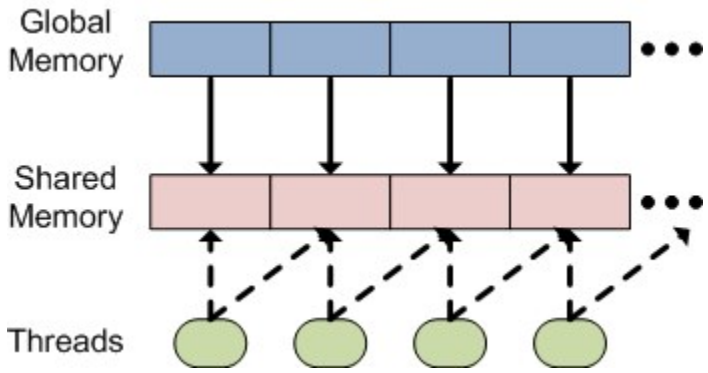

Figure 3.3: Thread-level data access patterns in the lookback GPU kernel.

value from global memory to shared memory, but reads two from shared memory. In this way, we exploit coalescing and improve the overall performance of all but one (per thread block) of our global memory reads [2] due to the structured, regular nature of the binomial lattice algorithm. Writing data back to global memory exploits coalescing as well, as each thread writes its value at the same time as its neighbors.

Our completed algorithm consists of two parts: the GPU kernel and the CPU control loop. The GPU kernel is shown in pseudocode in Algorithm 1. The `OptionVals[]` and `TempOptionVals[]` arrays are the two lattice node data sets. `TempOptionVals[]` contains the data that will be replaced with the current nodes data that we compute using the previous layer's values found in the `OptionVals[]` array. The first major if block (line 3) handles the loading of required data into shared memory (from global memory), and the second major if block (line 11) performs the actual computations to update a node. The CPU control loop manages the kernel invocations, the double-buffering of the node arrays, and decides when to terminate the algorithm (which

[2] As shown in Algorithm 1 the last thread in a block must perform one extra global memory read to finish populating the shared memory buffer. This single read is uncoalesced as there are no other threads available in the warp to combine memory requests.

occurs when the algorithm completes processing of the root node).

Algorithm 1 Lookback GPU Kernel

Require: u, d, pu, pd, disc, OptionVals[], TempOptionVals[], timeStepNodeCount

```
1: threadID ← blockIdx.x ∗ blockDim.x + threadIdx.x
2: sVal[] //Location in shared memory
3: if threadID < timeStepNodeCount then
4:    sVal[threadIdx.x] ← OptionVals[threadID]
5:    if This thread is the last in the block, or the last node in the lattice then
6:       sVal[threadIdx.x +1] ← OptionVals[threadID +1]
7:    end if
8:    yValue ← u^threadID //Compute y-value for this node
9: end if
10: __syncthreads()
11: if threadID < timeStepNodeCount then
12:    if threadID == 0 then
13:       optionValue ← max(yValue - 1, (pu * sVal[1] * d) + (pd * sVal[0] * u)
14:    else
15:       optionValue ← max(yValue - 1, (pu * sVal[threadIdx.x + 1] * d) + (pd *
          sVal[threadIdx.x] * u)
16:    end if
17:    TempOptionVals[threadID] ← optionValue
18: end if
```

3.3.1 Hybrid Computations

Before closing this section, we want to bring attention back to the reduction in parallelism throughout the algorithm. Recall that each subsequent phase of the algorithm (covering a subsequent layer in the lattice) requires one less thread than the previous phase. With the GPU, we have an architecture that places a strong emphasis on parallel execution. Due to the structure of the lattice, we are reduced to throwing only a few threads at a time at the hardware by the close of the algorithm. In order to try to combat the reduction in performance this may imply, we investigated the potential of a hybrid algorithm that moves the computations over to the CPU when the number of threads required for an iteration has dipped below a certain threshold. The threshold value defines how few iterations must remain before switching to CPU computations. For example, a threshold value of 128 implies that when there are 128 iterations remaining (that is: 128 layers left to be processed in the lattice), we cease computations on the GPU and shift to the CPU.

When we shift the computations to the CPU we must first copy the current state of the lattice from the GPU's global memory over to the CPU's main memory. The amount of data required for the transfer is small, however. As we will only be transitioning to CPU computations when the number of remaining levels are small, the number of lattice vertices required in the transfer will also be small. As a result, we do not expect the memory transfer overhead to be of a concern. We further hypothesized that the CPU will prove to be an optimal choice for executing the remaining iterations of the computations. Due to the lack of parallelism present in the later layers of the lattice, a comparatively powerful CPU may very well prove to outper-

form the lower-powered GPU cores that rely on high degrees of parallel throughput. As we show in the following section, however, this modification did not provide the performance improvement expected.

3.4 Results

Before investigating the performance of our GPU algorithm, we first tested it for correctness. In order to determine whether or not the GPU algorithm provided correct pricing results, we first ran a sequential CPU algorithm against a known test case provided in Hull [17] (Technical Note 13). Once we confirmed the sequential CPU algorithm was producing correct answers, we ran a more thorough set of tests involving unknown problem instances. We ran each instance on the sequential CPU algorithm and the GPU algorithm and compared the results. So long as the results were within the third order of accuracy (0.001) we accepted them as correct. Table 3.1 provides the results of some of these tests. In the case of smaller time steps, the GPU and Hybrid implementations provided results that fell within the acceptable range of error. However, as we increased the number of time steps above 10,000 steps, the level of precision began to drop and the GPU and Hybrid implementations no longer provide acceptable solutions. Our tests ended at 30,000 time steps, as single-precision floating-point data could no longer cope at higher steps. Higher time step counts resulted in overflow due to an exponentiation step in the algorithm.

We believe that there are two reasons for why the result on the GPU begins to lose precision (when compared to a reference CPU solution). First, the GPU we used (a GTX 260) does not have ECC-enabled memory on board, and second, the

S - 50, T - 0.35, $\sigma - 0.3$, r - 0.1, N - 1000		
CPU	GPU	Hybrid
6.676507	6.676391	6.676435
S - 50, T - 1.5, $\sigma - 0.3$, r - 0.15, N - 5000		
CPU	GPU	Hybrid
12.671818	12.670992	12.671094
S - 25, T - 0.75, $\sigma - 0.3$, r - 0.1, N - 10000		
CPU	GPU	Hybrid
4.895824	4.895118	4.895140
S - 25, T - 0.75, $\sigma - 0.3$, r - 0.1, N - 15000		
CPU	GPU	Hybrid
4.901764	4.900720	4.900743
S - 25, T - 0.75, $\sigma - 0.3$, r - 0.1, N - 20000		
CPU	GPU	Hybrid
4.913733	4.912446	4.912470
S - 25, T - 0.75, $\sigma - 0.3$, r - 0.1, N - 30000		
CPU	GPU	Hybrid
4.914032	4.911983	4.912004

Table 3.1: Correctness tests for GPU and Hybrid implementations on a GTX 260. S is the price of the underlying asset, T is the time (in years) until the maturity date of the option, σ is the volatility, r is the risk-free interest rate and N is the number of time-steps.

GPU handles single-precision floating-point mathematics in a slightly different manner when compared to the CPU. Both of these combined likely caused the increased error rate as the number of time steps (and, thus, floating-point operations) increases substantially.

In order to test our hypothesis we investigated the solution quality of the GPU and Hybrid implementations on a new model of Nvidia GPU, a GTX 570. This GPU supports the CUDA 2.0 specification, and, as claimed by Nvidia [34], more closely follows the IEEE-754-2008 floating point standards. Table 3.2 provides our results for very large time steps. Clearly, the results are much improved over the GTX 260,

S - 25, T - 0.75, $\sigma - 0.3$, r - 0.1, N - 10000		
CPU	GPU	Hybrid
4.895786	4.895808	4.895805
S - 25, T - 0.75, $\sigma - 0.3$, r - 0.1, N - 15000		
CPU	GPU	Hybrid
4.901701	4.901758	4.901771
S - 25, T - 0.75, $\sigma - 0.3$, r - 0.1, N - 20000		
CPU	GPU	Hybrid
4.913624	4.913828	4.913815
S - 25, T - 0.75, $\sigma - 0.3$, r - 0.1, N - 30000		
CPU	GPU	Hybrid
4.913840	4.914063	4.914051

Table 3.2: Correctness tests for GPU and Hybrid implementations on a GTX 570. S is the price of the underlying asset, T is the time (in years) until the maturity date of the option, σ is the volatility, r is the risk-free interest rate and N is the number of time-steps.

and allow us to run up to the maximum of 30,000 iterations while still retaining the required precision [3].

In all cases, the GPU algorithm computes a result with acceptable precision. The GPU does not generate a result exactly equal to the CPU result due to the differences between how the two architectures handle single-precision floating-point numbers and arithmetic. This inconsistency explains the slightly differing results between the sequential CPU and parallel GPU code.

With the correctness tests completed successfully, we move on to the actual performance tests. For the first set of performance tests we look exclusively at the Hybrid algorithm. Before comparing the performance results of all three algorithms (CPU, GPU, and Hybrid), we wanted to determine the optimal threshold value for the hy-

[3]The slightly differing CPU values from Table 3.1 to Table 3.2 are due to the different hardware systems used for each test. We ran the GTX 260 tests on AMD CPU running CentOS Linux, and the GTX 570 tests on an Intel CPU running Windows 7.

brid algorithm. The threshold value determines at what iteration/time step we shift the computations from the GPU to the CPU. For example, a threshold value of 128 enforces a switch in execution to the CPU when there are 128 time steps left for the algorithm to compute (before reaching the root node in the lattice).

We investigated a number of the thresholds, and we have provided a graph of the performance results in Figure 3.4. As we have shown, lower threshold values typically exhibit an improved level of performance over higher values. These results fall in line with our expectations. Prior to these tests, we hypothesized that there were two problems with a higher threshold value: increased emphasis on sequential code for completing a greater amount of overall work as well as higher data transfer costs as the GPU must transfer the current state of the lattice from its global memory to the CPU's main memory. With a low threshold value of 128 or 256, we delay sequential execution on the CPU until we reach levels of significantly reduced parallelism.

Despite running the hybrid threshold tests over a large number of runs (with the results in Figure 3.4 an average), a threshold value of 256 slightly edged out that of 128. We believe this is due to a slightly more optimal balance between taking the load off the GPU when parallelism is lower, and communication costs compared to thresholds of 128 or 384. As this threshold value was optimal, we use it in the remaining performance tests that include the Hybrid algorithm.

Following the analysis of the Hybrid algorithm, we investigated the performance of all three algorithms: sequential GPU, parallel GPU, and Hybrid. We immediately observe, from the results pictured in Figure 3.5, that the GPU and Hybrid implementations vastly outperform the CPU implementation at large time step counts.

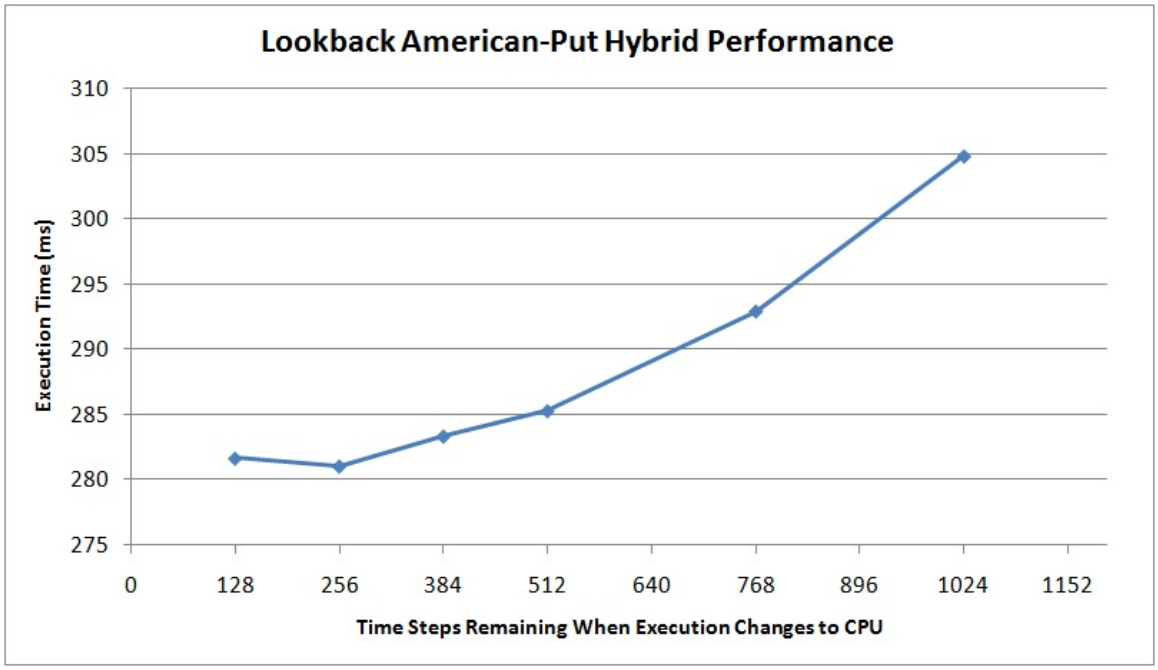

Figure 3.4: Performance of the Hybrid lookback algorithm with varying threshold values.

Hull [17] describes this particular lookback pricing technique as being slow to converge, hence performance results for large time steps are of immediate interest due to the inherent, improved accuracy of the result.

We also observe in Figure 3.5 a slightly disappointing trend: despite our best efforts to optimize the Hybrid implementation, it does not perform noticeably faster than the standard GPU algorithm. We see a performance improvement of only a few milliseconds for the Hybrid implementation at best. We initially believed this lack of improved performance results from the time required to transfer data from the GPU to the CPU. Using the CUDA profiling tool, however, showed us that we were mistaken. Unfortunately, the time required to transfer the small amount of data to the CPU, only 0.02% of the total run time, was miniscule. Our only remaining explanation is

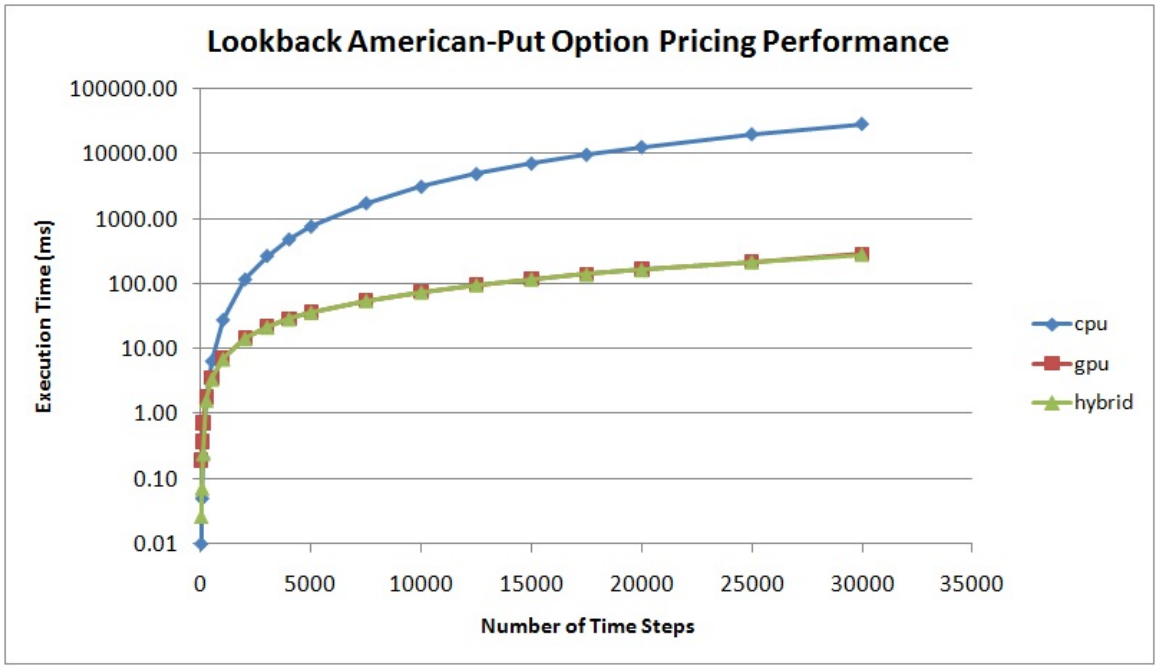

Figure 3.5: Performance of the CPU, GPU, and Hybrid lookback algorithm implementations.

that, while we see a clear reduction in parallelism as we approach the root node/layer of the lattice, there still exists some degree of exploitable parallelism throughout (even at layer one we can execute the computations for two nodes in parallel). In the end, it appears that the GPU still provides competition against sequential CPU code while there still exists the capability to execute this very regular code in parallel.

Finally, we show these results from another angle in Figure 3.6, where we measure the speedup of the Hybrid implementation compared to the sequential CPU implementation. We observe a roughly linear increase in speedup up to a time step count of 30,000. After this point, the speedup continues to increase as the time step count increases, but at a reduced rate. Our results show that the GPU is clearly able to dramatically outperform the CPU for large time step counts. We see a maximum of

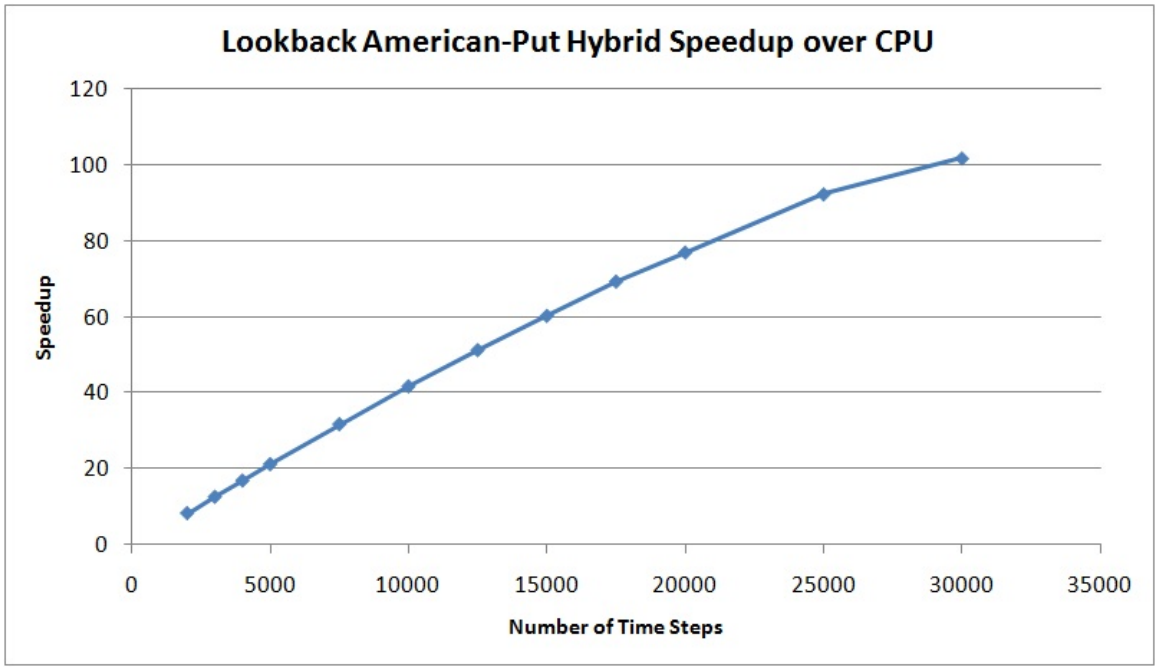

Figure 3.6: Speedup of the Hybrid implementation relative to a sequential CPU implemenation.

101 times speedup with the largest time step count tested (30,000 steps).

3.5 Summary

In this chapter we have described a high performance GPU algorithm for the pricing of lookback options using the binomial lattice technique. We originally chose this problem because it provided us with a very regular, data-parallel algorithm to experiment with on the GPU, and had not been done in the past. Due to the regular nature of this problem we were able to develop a very efficient algorithm that exploits both the high performance shared memory as well as the global memory coalescing capabilities of the GPU. We have shown tremendous speedup values, up to 101 times faster than a sequential CPU implementation, as the time step count grows large.

These results help to show that the structured, predictable nature of a regular problem allows for the realization of very high performance algorithms on the GPU.

In the next chapter we move on to an investigation of the Breadth First Search for solving the single-source shortest path problem. While the option pricing problem we just examined was very regular, the single-source shortest path problem provides us with the chance to experiment with a simple *irregular* problem. We expect that our results will provide some contrast between the performance of parallel algorithms with regular and irregular properties on the GPU.

Chapter 4

Breadth First Search

In this chapter we discuss our use of the Breadth-First Search algorithm for solving the single-source shortest path problem. We chose to investigate the BFS algorithm for two reasons: (1) BFS is an example of an irregular algorithm, and, (2) BFS is a relatively simple algorithm, that, unlike the binomial lattice for option pricing algorithm, contains irregular properties. As a result, we believe looking at the BFS algorithm serves as an ideal benchmark for the last two algorithms we cover in this thesis, which contain irregular properties but are significantly more complex.

4.1 Introduction to BFS

In a typical breadth-first traversal of a directed graph $G = (V, E)$, where V represents the set of vertices, and E the set of edges in the graph, we start from some vertex, $s \in V$, designated as the source vertex. In the first iteration we traverse all outgoing edges from s in order to reach all vertices neighboring s. We then repeat

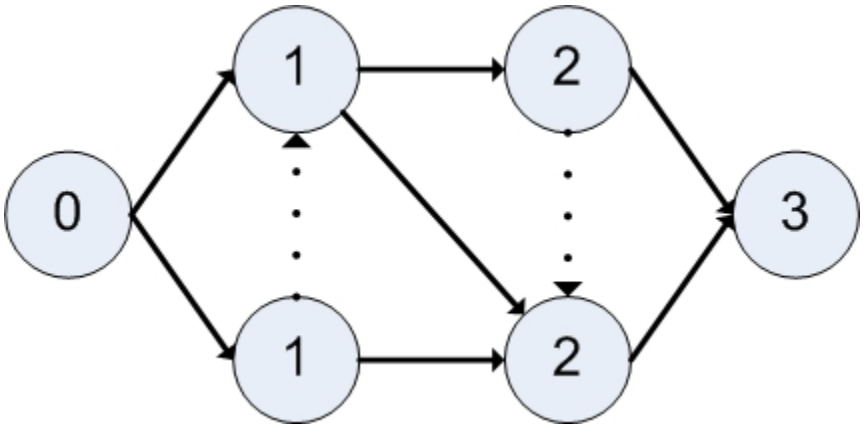

Figure 4.1: Layered network of a graph.

this process for each neighbouring vertex that we reached in the previous iteration (if and only if it had not already been visited in an even earlier iteration) until either all possible nodes in the graph have been visited or we have found the vertex we are searching for.

In order to test the worst-case of the BFS, and better connect it to the maximum flow algorithm that we investigate in the next chapter, we use the BFS to construct a layered network, which will allow us to solve the single-source shortest path problem. In this case, the BFS does not terminate until it has visited every vertex possible in the graph, assuming we are starting from some source vertex. In essence, we use the BFS to solve a simplified version of the single-source shortest path problem where all edge weights are assumed to be equal to one.

Our simplified version of the single-source shortest path problem involves finding a label for every vertex connected to s. The label on some vertex v represents the minimum number of edge traversals required to reach v from s. In effect, we construct a layered network of the graph, relative to s. A layered network of a graph labels

the vertices based on the minimum number of edge traversals required to reach them from the source vertex and, further, only retains edges in G such that the edge moves from layer i to layer $i + 1$. Figure 4.1 provides an example of a layered network with four layers, where the number inside a node represents the layer it belongs to. Layer 0 contains only the source vertex, and layer 1 contains all vertices directly reachable from the source. The vertices in layer 3 contain no further outgoing edges to "new" vertices, and, thus, layer 3 is the final layer in the network. The dotted edges represent edges in the original graph that no longer exist in the layered network (as they do not connect to a vertex in the next layer).

The layered network we construct using BFS allows us to then determine the shortest path to any vertex from a single source, s. We assign vertices to a layer based on the minimum number of edge traversals required to reach them from the source, and any edge from any vertex always moves to the next layer. As a result, we have a new graph structure that contains a guaranteed shortest path route from the source to any other vertex in the graph.

We chose to investigate this particular solution for the single-source shortest path problem not only because it allows us to test an algorithm with irregular properties on the GPU, but also because of the importance of a layered network for the maximum flow algorithm we discuss in Chapter 5.

4.2 Related Work

Unlike lookback option pricing, which was a new work for the GPU, there does exist some prior work on the BFS for GPUs. Hussein et al. [18] investigated a par-

allel BFS algorithm to traverse a graph layer-by-layer on the GPU. In their implementation, each invocation of a kernel handles the construction of one layer, with synchronization occurring between layers. The authors use a flag array to mark active vertices, and a parallel prefix sum (also known as a scan) operation to compact the list. They further optimize the BFS for traversal of grid-graphs by introducing lockstep traversal. This traversal method takes advantage of the structured nature of a grid-graph and activates vertices along each cardinal direction one at a time per active vertex. Hussein at al. further discuss that the version of the BFS not optimized for grid-graphs may not be able to provide improved performance over a CPU implementation. They discuss this in terms of computational complexity, however, and do not provide any actual performance results.

Following the work of Hussein et al. [18], Harish et al. [15] developed an alternative BFS implementation on the GPU a few years later. The authors' followed Hussein et al.'s work, as they move through the graph synchronously, layer-by-layer, and use an array of flags to identify the layer each vertex belongs to. Unlike Hussein et al. however, Harish et al. use two flag arrays. One stores a boolean value indicating whether or not a vertex has been visited, and the other stores a boolean value indicating whether or not a vertex needs to be visited in the next iteration (the "frontier" vertices). The frontier array requires a secondary kernel to copy values between a dummy array, in a sort of double-buffering strategy. Furthermore, they use a third array to store the layer information for each vertex. In effect, their solution requires a large number of memory operations, space, and even an auxiliary kernel invocation for copying the current state of the frontier array. The authors show

that their algorithm achieves up to 15 times speed up compared to a sequential CPU implementation when working with random graphs.

Martín et al. [30] implemented Dijkstra's algorithm on the GPU in order to solve the single-source shortest path problem. The authors focus on the parallelization of the edge-relaxation step of Dijkstra's algorithm. They further experiment with a number of hybrid implementations that offload some of the inherently sequential processing to the CPU. Finally, Martín et al. develop two GPU kernels: one that makes use of atomic memory operations, and another that does not. The authors show that their GPU implementations perform better than their CPU implementations. They further describe how their kernel that uses atomic memory operations outperforms the non-atomic kernels due to the changes in logic required to strip away the atomic memory operations. However, the authors claim that one of the reasons why the performance penalty is not high for the kernel that uses atomic memory operations is due to the low-degree graphs that they experiment with. Martín et al. show that for high-degree graphs the performance of the atomic memory kernel dips below that of the others.

Finally, Solomon and Thulasiraman [47] investigated the performance of the BFS on grid graphs as a comparison to another algorithm with similar iterative properties, matrix parenthesization. Our results on small grid graphs showed that the GPU implementation could not provide improved performance over a sequential CPU implementation. We further described, however, that the performance disparity between the GPU and CPU shrank as the graph size (in terms of vertex count) increased. Due to the simple and rudimentary nature of the work, we significantly expand on it here,

and focus on large, random graphs for our testing.

4.3 BFS on the GPU

The work that we present here marks an improvement on the work we previously described in Solomon and Thulasiraman [47]. We extend the work to cover random graphs, and, more importantly, test with very large random graphs in order to get a better idea of the GPU performance.

Our work, as with the previous work by Solomon and Thulasiraman [47], loosely follows the design of Harish et al.'s [15] BFS. We invoke our GPU kernel, which performs the actual work associated with the BFS, once per layer in the graph. In other words, one invocation of our GPU kernel constructs one layer of the graph by following edges from each vertex in the current layer and performing the appropriate labeling operation.

In terms of thread responsibilities, we map one thread to one vertex in the graph. Thread i covers vertex i, thread $i + 1$ covers vertex $i + 1$ and so on. Initially, we do not consider maintaining a record of only those vertices that will be active in the current iteration and assigning only enough threads to cover *those* vertices. Instead, our first design of this algorithm for the GPU simply launches all threads covering all vertices in the graph, regardless of whether or not that vertex belongs in the current layer (and, thus, will be active and performing some useful work).

In order to store the graph data itself, we use the same two one-dimensional arrays, described by Harish et al. [15], which represent an adjacency matrix. Elements of the first array, of length $|V|$, store the offset of the starting edge for the vertex

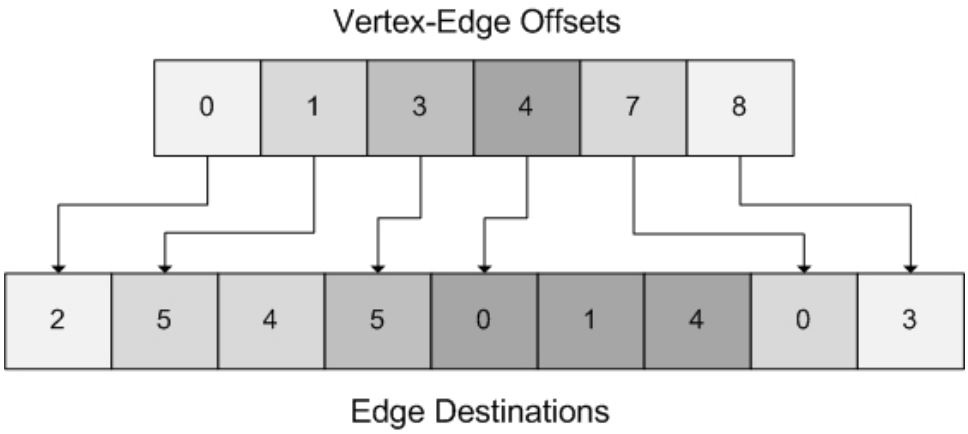

Figure 4.2: Structure of two-array adjacency list for storing graph data on the GPU.

corresponding to that element's index. The second array, of length $|E|$, stores the actual edges themselves; elements in the first array provide indices into this array.

Figure 4.2 provides one example of the two-array adjacency matrix we use to store graph data. We base this example on a graph that contains six vertices and nine edges. As an example of how we find the index of a vertex's starting edge, consider element three in the vertex-edge offsets array (assuming a zero-based indexing). Element three corresponds to the fourth vertex in the graph, and has a value of 4. This 4 tells us that the fourth element in the edge destinations array contains the starting edge for vertex four. We then peek at the next element in the vertex-edge offsets array, which contains a value of 7. This 7 tells us that all edges from elements 4 to 6 in the edge destinations array are edges with their source at vertex four.

Having described the data structure we use to store the graph data we can now begin our description of the algorithm. At the start of an invocation of the GPU kernel, each thread first determines if the vertex it covers is active or not. We make this determination by reading the thread's assigned layer. Consider iteration i of the

kernel. In this iteration we search for all vertices that belong to layer $i + 1$. In order for a vertex to belong to layer $i + 1$ it must be connected by an edge to a vertex in layer i. Thus, any threads assigned to a layer equal to the current iteration number are "active". Any threads that are not active simply halt their processing and do not participate further in the current iteration of the kernel.

Following the determination of which threads are active, each active thread then iterates through the edges of its vertex and sets the level of each unlabeled neighboring vertex to the appropriate value. In order to iterate through the edges of a vertex, however, each active thread must first read in the corresponding data from the adjacency matrix. The way in which we store the graph data allows us to exploit global memory coalescing for the reading of the vertex-edge offset array. As thread i covers vertex i, we guarantee that each thread reads from a location in memory offset by one element from its neighboring threads. Of course, inactive threads do not participate, but this lack of participation does not break the coalescing effect. Unfortunately, we cannot guarantee any level of coalescing while each thread iterates through the edges of its vertex, as we have no guarantee that each vertex contains the same number of edges. Thus, some threads may either read from dramatically different segments of memory, or may simply have more work to cover than the majority of threads (depending on how many edges each vertex contains).

As we know from Figure 4.2 and our corresponding example, we need two values for each thread from the vertex-edge offset array — the starting element/offset for that vertex's edges, and the starting element/offset for the next vertex's edges (from which we compute how many edges the vertex contains). When we load this data, we

have each thread read a *single* value from global memory, rather than the two values that are required. This is possible as we store these read values in shared memory. As a result, thread i reads the value it requires for a starting offset (value i in the vertex-edge offsets array), and thread $i + 1$ reads value $i + 1$, which both thread i and $i + 1$ require. This allows us to cut down the global memory reads required when loading in graph data at the start of the kernel.

One of the benefits of our method compared to the work of Harish et al. [15] is that we do not require multiple arrays to store the status of our vertices. In their work, they maintained a frontier array, a visited array, and a vertex-label array. This required more global memory operations to keep these arrays updated, potentially reducing the performance. Instead, we manage with a single array: the vertex-label array. This array defines which vertices are active in a given phase based around what layer the algorithm assigned them to. If a vertex belongs to a layer equal to the current iteration it forms a part of the frontier, any vertex with a layer number assigned to it at all has been visited, and any vertex with no layer assigned simply waits to be assigned a label.

We guarantee that an unlabeled vertex will always receive the most optimal label the moment a thread encounters it due to the synchronous nature of the algorithm. In the first iteration, only the source vertex is active, and from there, we visit every neighboring node and assign them to level 1, thus activating them for the next iteration. All edges leading out from every active vertex are investigated at each iteration, and each thread assigns any unlabeled vertex a label equal to the current level. Hence, we visit each vertex as early as possible and with the minimum number

of hops required to reach it from the source vertex.

As was the case with the option pricing algorithm, the CPU drives the iterations, and invokes the GPU kernel each iteration. The CPU terminates the algorithm once an iteration has passed and the kernel has assigned no new vertices to a layer. Algorithm 2 provides the pseudo-code for the GPU kernel. Lines 5 – 7 perform the loading of the edge offset data. We require a block-level synchronization instruction immediately thereafter to ensure that all required data has been loaded in to global memory in case a thread moves on and tries to read data another thread has not loaded yet. Lines 9 through 19 perform the actual BFS algorithm we have discussed in this section. Following the BFS work itself, we perform a parallel OR reduction on the shared memory flag array which will tell us whether any thread in the thread block has added at least one vertex to the current level. Finally, the first thread in all thread blocks write this reduced value into the same location in global memory. We use an *AtomicOR* operation to ensure consistency. This single value in global memory will contain either a 1 (at least one vertex was added to the level) or a 0 (no vertices were added to the level) which the CPU control loop uses to determine whether or not to terminate the algorithm. While the CPU control loop is simple, we provide the pseudo-code in Algorithm 3 for completeness sake.

4.3.1 Active Vertices Modification

With the original BFS algorithm completed for the GPU, we noted one primary concern. The fact that we simply launched all threads covering all vertices in the graph with every iteration of the algorithm was a clear issue in the original im-

Algorithm 2 Breadth First Search GPU Kernel

Require: $G = V, E$, Levels[], VertexEdgeOffsets[], EdgeDests[], CurrLevel

```
1: threadID ← blockIdx.x * blockDim.x + threadIdx.x
2: sEdgeOffsets[] //Location in shared memory
3: sNewVertex[] //Location in shared memory
4: //Load in the edge offsets for the graph data.
5: if threadID < |V| then
6:    Read a single edge offset value equal to thread ID, store in sEdgeOffsets[]
7: end if
8: __syncthreads() //Synchronize threads in block.
9: if threadID < |V| then
10:   if Levels[threadID] = CurrLevel −1 then
11:      Load startEdge and endEdge offset data from shared memory.
12:      for each edge e = (u, v) in EdgeDests[] from startEdge to endEdge do
13:         if Levels[v] has no level then
14:            Levels[v] ← CurrLevel
15:            Set this thread's sNewVertex[] entry equal to 1.
16:         end if
17:      end for
18:   end if
19: end if
20: __syncthreads()
21: //Determine if any thread in the block updated a vertex label
22: Perform parallel OR reduction on sNewVertex[]
23: Thread 0 (per block) AtomicOR's sNewVertex[0] to static global memory location
```

Algorithm 3 Breadth First Search CPU Control Loop

```
1: NewVertices ← 1
2: level ← 1
3: while NewVertices > 0 do
4:     Invoke BFS GPU kernel for the current level
5:     NewVertices ← active vertex count from GPU Global Memory
6:     level ← level +1
7: end while
```

plementation. In an effort to improve on our design, we implemented the "active vertices" modification.

Our modification follows a similar idea as that of Hussein et al. [18]. In their work, they describe the use of a parallel prefix sum, or scan, operation to determine the active vertices for a given iteration. We, too, use a scan operation in order to determine the active vertices for the next iteration. To accomplish this, we generate a vertex flag array that defines which vertices will be active in the next iteration of the BFS. A 1 value for vertex i tells us that i has been assigned a level in the previous iteration, and will be active in the next, while a 0 tells us the opposite, that i will not be active in the next iteration. In order to generate this we require an extra write to global memory in our BFS algorithm — each active thread must mark each vertex it encounters with a 1 in the appropriate spot in this array.

Of course, we require more than just the vertex flag array, as the end result that we desired was to have zero "useless" threads (that is, threads covering inactive vertices) launched with each iteration of the BFS kernel. Having a linear array of boolean

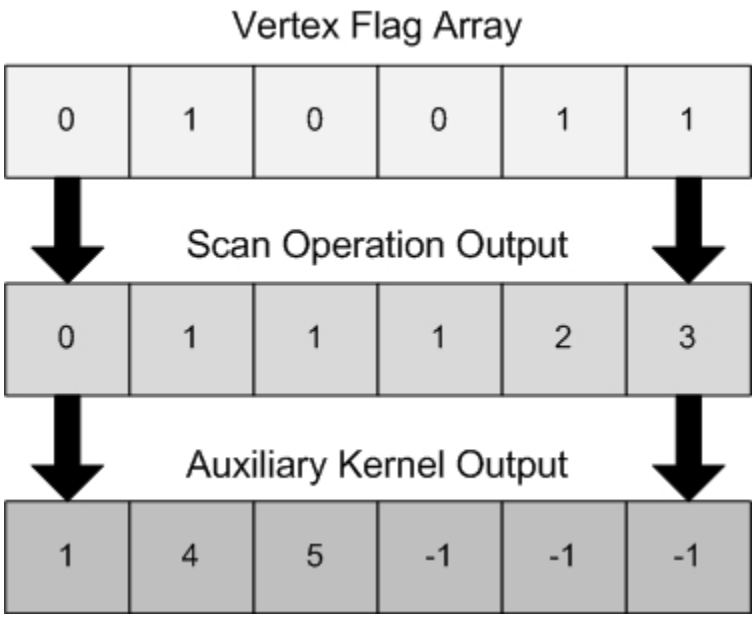

Figure 4.3: Process of recording active vertices for the next iteration.

flags does not help us achieve this goal as we would still have to launch the kernel with all threads and have each one read its element in this array. At this point, we introduce the scan operation. We perform, using the CUDPP Library's [43, 42] high performance GPU algorithm, a scan operation on the vertex flag array. This gives us a new array where each change in value from one element to the next corresponds to the index of an active vertex. Consider the example in Figure 4.3. The scan operation consumes the vertex flag array and outputs an array where element i equals the total sum of elements 0 through $i-1$ in the vertex flag array. We see that each *index* in the scan operation output array where the value changes from that of the previous element represents the index of an active vertex. Using a lightweight auxiliary kernel, we read this scan array and write out, to global memory, an exact listing of the active vertices along with the total number of active vertices (so we know how many threads

to launch in the next iteration).

We did not perform any major additions to the existing GPU kernel in order to introduce the active vertices strategy. In fact, we were able to simplify the kernel as we longer needed to maintain a record of whether or not each thread block updated at least one vertex. With the active vertices modification we moved these checks to the auxiliary kernel that generates the flag array used for the scan operation.

We hypothesized that both the simplification of the kernel, as well as the fact that we were no longer launching many useless threads with every iteration may outweigh the overhead created by the scan and auxiliary kernel operations. We believed the end result would improve our performance over the standard BFS algorithm. Unfortunately, as we show in the following section, that was not necessarily the case.

4.4 Results

Prior to discussing our results, we must first describe how we generate our random graphs. In order to accomplish random graph generation we use the graph generation tool described by Viger and Latapy [54, 55]. We chose this particular tool as it generates simple random graphs and supports the ability to define the range of degrees for vertices. By being able to adjust the average degree of the vertices in a graph, we can test performance across a wide range of scenarios. For all of our performance tests we generate at least five different graphs for each vertex count, with slightly different vertex degrees. This allows us to average the performance results for a certain graph size across a few unique graphs. We further average the performance results for each individual graph by taking the average execution time of 50 runs per individual graph.

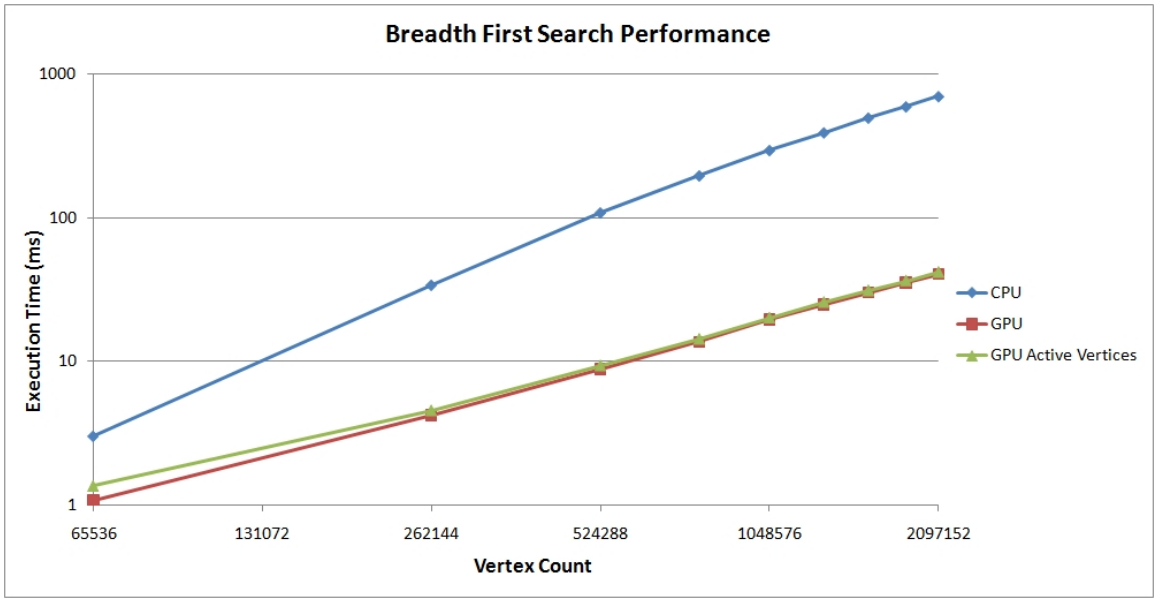

Figure 4.4: Performance of the CPU, GPU, and active vertices GPU implementations.

We test the performance of the BFS for both the standard GPU implementation as well as the active vertices implementation, and compare the results against a sequential CPU implementation. We provide the comparison results for all three implementations in Figure 4.4. For these performance tests we set the average vertex degree at 12 in order to test scenarios where the BFS will likely have to run for more than a few iterations in order to encounter all the vertices possible. As Figure 4.4 shows, both GPU implementations outperform the sequential CPU implementation. Our results contradict Hussein et al.'s [18] hypothesis, as both GPU implementations, including the active vertices implementation (which relies on a scan operation) outperform the CPU.

Unfortunately, we do not see the performance improvements we had hoped for in the active vertices implementation. In fact, this version of the GPU algorithm consistently performs worse than the basic implementation! As we show in Figure 4.5,

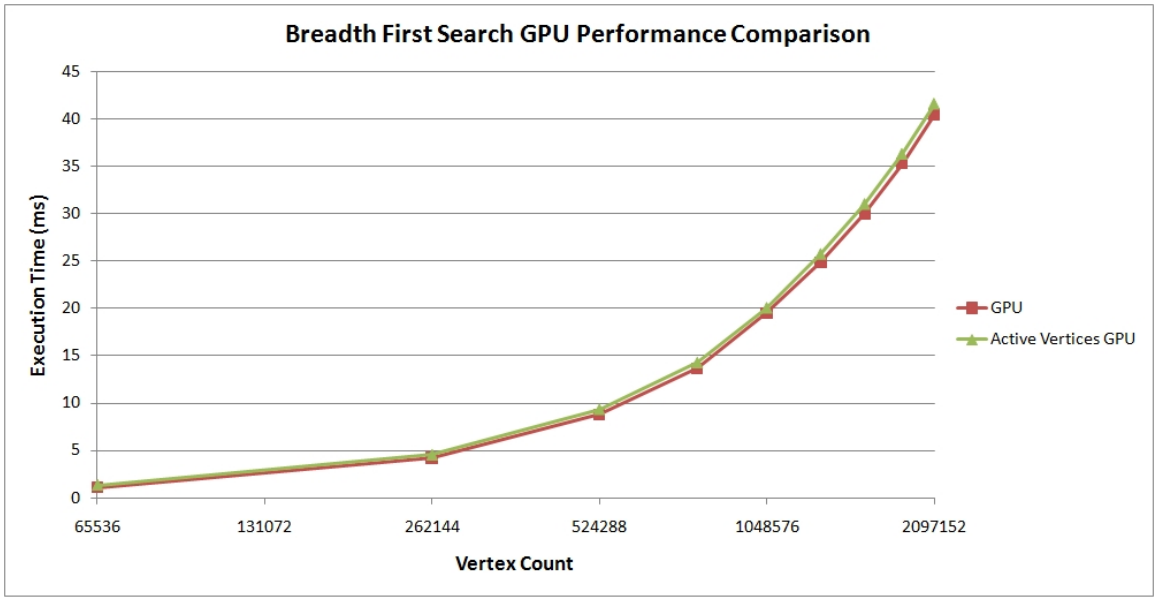

Figure 4.5: Performance comparison of GPU implementations

which removes the CPU performance results in order to get a closer look at the GPU results, both implementations perform very similarly, but the active vertices implementation does not provide superior performance at any time.

To show these results from another angle, we compute the speedup of both GPU implementations relative to the sequential CPU code and provide a graph of the results in Figure 4.6. From the speedup results, we see that the GPU increases its performance relative to the CPU as the problem size increases. This falls in line with our expectations as more vertices potentially equals more exploitable parallelism at any given iteration. The GPU hardware, being a very highly parallel architecture, can handle the extra parallelism with ease. We see up to approximately 17 times speedup for the largest graph sizes tested, and, on average, a speedup of 13. As we expect, the speedup results for the active vertices implementation do not match those

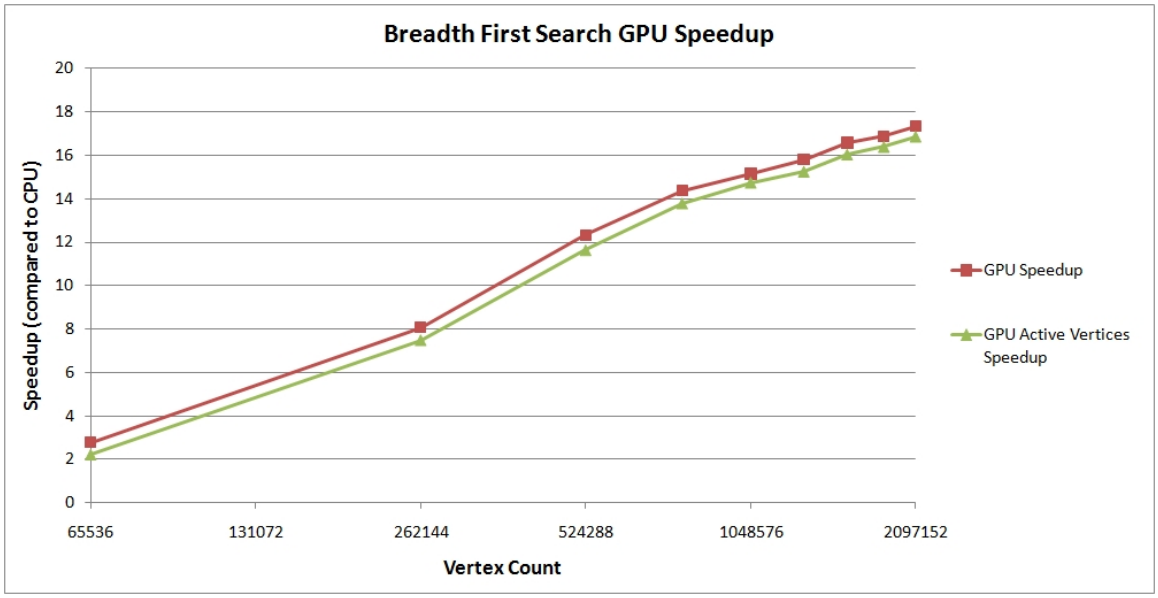

Figure 4.6: Speedup of GPU BFS implementations.

of the basic GPU implementation, but they come close (in general, the speedup was only 0.5 units lower).

	BFS kernel time (μs)	Scan & auxilary kernel (μs)	Total (μs)
Basic BFS	47135.71	—	47135.71
Active vertices BFS	40696.96	6889.21	47586.18

Table 4.1: Profiling data for GPU BFS kernels

The lack of improved performance inherent to the active vertices implementation left us with the question of *why*. We found that the overhead resulting from the scan operation (and the auxiliary kernel required to prepare the data) outweighs the performance savings in the BFS kernel. To provide an example of this we profile one execution run of both GPU implementations using the CUDA Profiling Tool. Table 4.1 provides our results. From the table, we see that the BFS kernel itself

performs much better in the active vertices implementation. We expect this, as we are launching only the threads necessary to complete the iteration, rather than *all* threads covering *all* vertices in the graph (despite many of them likely being inactive). Unfortunately, the overhead of the scan operation outweighs these performance benefits. The end result, as Table 4.1 shows, is that the basic BFS implementation manages to provide a greater level of performance.

4.5 Summary

In this chapter we investigated an algorithm with irregular properties: the BFS algorithm for solving the single-source shortest path problem. While we observed a high level of speedup with the BFS, the performance paled in comparison to the results of our option pricing algorithm. The disparity in performance between these two algorithms is the result of the irregularity present in the BFS. Unlike the option pricing algorithm we were unable to guarantee global memory coalescing. Further, threads had workloads of varying sizes, and not all threads were guaranteed to be active without extra effort made to cull inactive threads from execution. Unfortunately, further issues occurred as the effort made to cull inactive threads resulted in even worse performance. Despite these problems, however, we still measured up to 17 times speedup with our GPU implementation, showing that the GPU still offers a high level of potential performance for a simple irregular algorithm. In the next chapter, we build on this simple irregular problem and introduce a more complex irregular problem and associated algorithm: the MPM maximum flow algorithm for solving the maximum flow problem.

Chapter 5

The MPM Maximum Flow Algorithm

In this chapter we investigate the use of the MPM algorithm for solving the irregular maximum flow problem. Unlike the BFS, which contained only a single phase, the MPM algorithm contains a number of phases which we parallelize. These phases add complexity, and require optimizations of their own in order to prepare them for efficient GPU implementation. We lead this chapter with an introduction to the maximum flow problem before moving on to describe the MPM algorithm itself.

5.1 Introduction to Maximum Flow

Before describing the MPM algorithm, and how it solves the maximum flow problem, we must first define the maximum flow problem. Given a directed graph $G = (V, E)$ where V is the set of vertices, and E is the set of edges, G must satisfy

the following conditions:

1. There exists a single vertex with zero indegree. By indegree, we refer to the number of directed edges leading into a vertex. Hence, a vertex v has zero indegree if there are no edges in E of the form (i, v), where $i \in V$. We refer to this vertex as the *source*, or s.

2. There exists a single vertex with zero outdegree. Opposite to indegree, outdegree refers to the number of directed edges leading out of a vertex. Hence, a vertex v has zero outdegree if there are no edges in E of the form (v, j), where $j \in V$. We refer to this vertex as the *sink*, or t.

3. Every directed edge $e = (i, j)$ is assigned a non-negative real number $c(i, j)$. This value represents the capacity for flow of e, or how many units of flow we can send across e. $c(i, j) = 0$ if there is no edge directed from i to j.

A flow, f, through G is an assignment of positive real numbers, $f(i, j)$, to the various edges $e = (i, j)$ such that the following conditions are satisfied:

1. *Capacity constraint*: $0 \leq f(i, j) \leq c(i, j)$, for all $e = (i, j)$.

2. *Conservation constraint*:

$$\sum_j f(i,j) - \sum_j f(j,i) = \begin{cases} \sum_j f(s,j) & \text{if } i = s, \\ -\sum_j f(j,t) & \text{if } i = t, \\ 0 & \text{otherwise} \end{cases}$$

The capacity constraint ensures that the flow across an edge does not exceed the capacity for flow of that edge. The conservation constraint ensures that a vertex does

not trap flow: all flow entering a vertex via incoming edges must be able to be pushed out of that vertex via the outgoing edges while still satisfying the capacity constraint. In other words: the *inflow* of a vertex (all flow entering the vertex) must be equal to the *outflow* of the same vertex (all flow exiting the vertex). In the case of s and t, we can, of course, have no inflow or outflow respectively. The conservation constraint captures these special cases as well.

The value of a flow f, denoted by $val(f)$, is defined as $val(f) = \sum_j f(s, j) = \sum_j f(j, t)$, or the total sum of all flow across all edges in G. Due to the conservation constraint, the total amount of flow pushed out of the source is equal to the total amount of flow pushed into the sink, which is equal to the total flow pushed through all other edges not connected to s or t in G. We have found a *maximum* flow, $f^\star$, in G if there exists no other flow f in G such that $val(f) > val(f^\star)$. Therefore, the maximum flow problem is to find the maximum flow, $f^\star$, in a given graph, $G = (V, E)$.

5.2 The MPM Algorithm

Malhotra, Pramodh Kumar, and Maheshwari [29] ("MPM") based their maximum flow algorithm around the layered network concept proposed by Dinic [6]. The authors designed their algorithm around a number of phases, where each phase performs a distinct task in the algorithm. This algorithm exhibits very synchronous, iterative properties between phases and, as a result, we hypothesized it to be a good fit for the GPU. We came to this conclusion by considering the SIMD nature of the GPU along with the synchronicity inherent between kernel executions in the GPU. In essence, the algorithm iterates through the various phases, with synchronization between each

phase. Hence, we hypothesized that phases would readily map to kernels and retain the original nature of the algorithm.

We first break the MPM algorithm down into its constituent phases before describing the algorithm in full. Each phase covers some unique function in the algorithm, and the next phase cannot start until the current phase has completed [1]. In total, the MPM algorithm contains six phases: layered network construction, construction of blocking flow, pushing of flow, pulling of flow, updating edge capacities, and pruning. We describe each phase in more detail in the subsections below.

5.2.1 Construction of layered network

In this phase, we construct a layered network, G_L, from G. This phase considers only *useful* edges of G, or any edge that has not been fully saturated with flow. Put more formally, we call an edge $e = (u, v)$ useful if $f(e) < c(e)$. The first layer L_0 contains the source. The next layer L_1 consists of those vertices v of G such that there is an useful edge from s to v. We add an edge, $e = (u, v)$, to the layered network if and only if e is useful, u belongs to layer L_{i-1}, and v has not been assigned to any of the previous layers. This process continues until the sink vertex, t has been assigned to some layer, or no new layers can be constructed (in which case the algorithm terminates). Finally, edges that lead to "dead ends" (that is, they reach a vertex with no useful outgoing edges and, thus, do not form a path from s to t in G_L) are pruned from G_L. In essence, to construct a layered network we perform a breadth-first traversal of a graph.

[1] The one exception, as we will discuss in Section 5.4, is the pushing and pulling of flow.

As we traverse the edges in this phase, we assign each edge $e \in G_L$ a residual capacity value. The residual capacity of an edge, $\tilde{c}(e) = c(e) - f(e)$, represents the remaining units of flow this edge can support. For example, if an edge, e, already has 20 units of flow ($f(e) = 20$), and a capacity of 30 ($c(e) = 30$) its residual capacity becomes $\tilde{c}(e) = 10$. The residual capacity of an edge becomes important when we investigate the method by which we push and pull flow throughout G_L.

5.2.2 Construction of blocking flow

In this phase we search for a candidate vertex in G_L which will act as the starting point for the pushing and pulling of flow. We begin with a few important defitions. Let v be a vertex of the layered network G_L. Then the *in-potential* of v is the sum of all residual capacities of the edges directed into v. The *out-potential* of v is the sum of all residual capacities of all edges directed out of v. The *potential* of v is then the minimum of the in-potential and the out-potential of v. Each of the vertices in G_L calculates their potential and we choose the vertex with the minimum potential greater than zero, v_m, as the candidate vertex for initiating the pushing and pulling of flow through G_L.

5.2.3 Push flow

Let candidate vertex v_m of smallest potential, p, be in layer L_i. Consider the edges out of v_m. Clearly these edges belong in layer L_{i+1}. Examine all these edges and try to saturate each edge with flow until all required flow is pushed (where the required flow is equal to p). Next, consider all vertices of L_{i+1} that received some flow from

the previous layer. Perform the push operation as explained above from this layer to layer L_{i+2}. Repeat this process until the flow reaches t. Note that at each layer we push exactly p units of flow. As the candidate vertex v_m was of minimum potential in G_L we are guaranteed to always be able to push p units of flow through even just a single vertex in any layer in G_L.

5.2.4 Pull flow

Similar to the pushing of flow, we must also pull p units of flow from v_m to all layers lower than L_i. Consider all the incoming edges of v_m in L_i and saturate as many as possible until we have pulled p units of flow out of v_m. Next, consider all vertices of L_{i-1} that received some flow from the previous layer. We perform the pull operation again from this layer to layer L_{i-2}. Repeat this process until the layer containing source s (layer L_0) is reached.

5.2.5 Pruning

Once we have pushed the flow to t and pulled flow from s, we prune G_L. This pruning step removes saturated edges as well as vertices with no incoming and/or outgoing edges. We are guaranteed to remove at least one vertex initially, as the vertex v_m that had minimum potential must have had either its incoming or its outgoing edges (or both) saturated. When pruning a vertex, we completely remove all of its incoming and outgoing edges. Next, there may be other vertices that we are now able to remove. This pruning effect potentially cascades to the neighboring vertices of a removed vertex and continues until we can prune no further vertices from

G_L.

5.2.6 Update capacities

We increased the flow in various edges, e, of G_L by pushing/pulling flow through them and, as a result, the residual capacities of each edge decreased. We now need to update the capacities of each edge in G from the corresponding edges in G_L. The edges in G_L contain the updated residual capacities and, by updating G with these values, the flow in G reflects the latest amount of flow in G_L.

5.2.7 The complete algorithm

Once the algorithm completes the pruning phase, we iterate through the construction of blocking flow, pushing flow, pulling flow, and pruning phases again if there still exists some path from s to t in G_L. In the event that no vertices remain in G_L, or there is no path from the source to the sink, the total flow sent through the edges of G_L are merged with the current data stored in G (phase six). A pruning step takes place in G and the algorithm restarts by constructing a new layered network (phase one) from the updated state of G. The algorithm terminates when we can no longer create a layered network from G with a path between s and t.

We provide an example of an iteration of the MPM algorithm in Figure 5.1. Consider part (a) in Figure 5.1, which shows a graph, G, in its initial starting state. Numbers above (to the left, in the case of edge $(4, t)$) represent $c(e)$, while numbers below represent $f(e)$. In the initial state of G, $f(e) = 0$ for all edges, as we have not pushed any flow through G. In part (b), we construct the layered network, G_L from

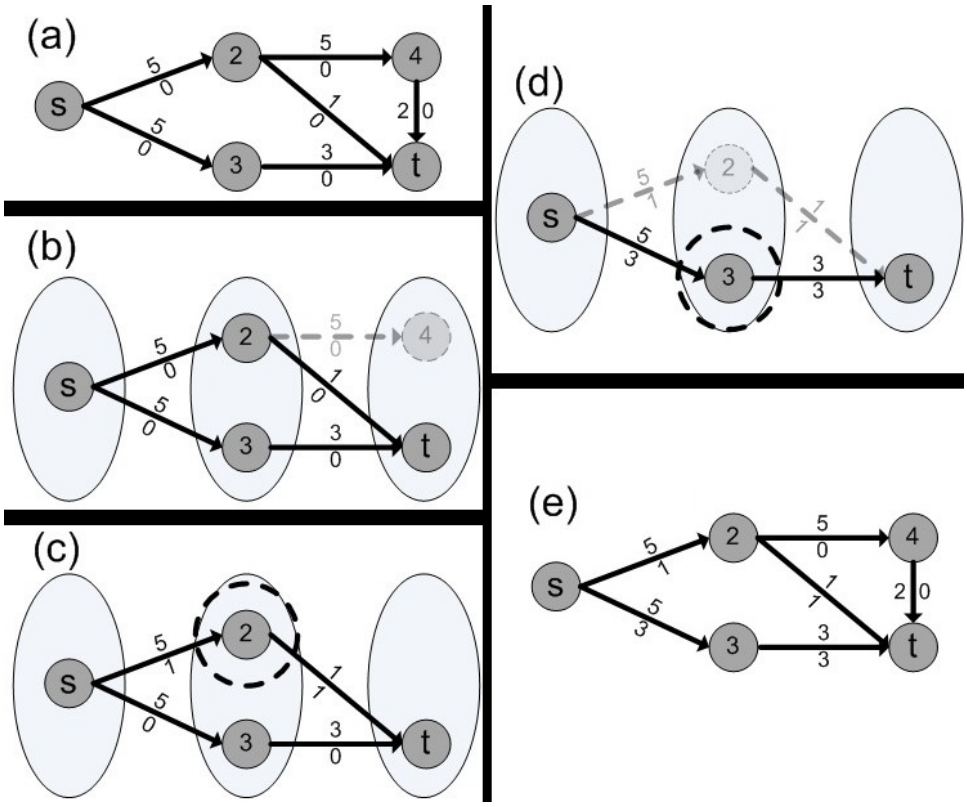

Figure 5.1: An example iteration of the MPM algorithm.

G. Note that edge $(2, 4)$ and vertex 4 are pruned from G_L as they are not a part of any valid path from s to t in the layered network. We then search for v_m in G_L in part (c). We choose vertex 2 as its potential equals 1, which is smaller or equal to the other possible vertex in G_L, 1 (potential of 3). Part (c) further involves pushing and pulling from from vertex 2. As the potential of this vertex equals 1, we push 1 unit of flow across edge $(2, t)$ and pull one unit of flow across edge $(s, 2)$, updating the $f(e)$ values in G_L accordingly. Notice that edge $(2, t)$ becomes fully saturated at this point — as a result, we prune it from G_L, as shown in part (d) of Figure 5.1. As vertex 2 no longer forms any part of a path from s to t we prune it as well, with

further cascades in to pruning edge $(s, 2)$. We still have a path from s to t in G_L, however, so we search for the next v_m. We choose vertex 3, with a potential of 3, and push/pull 3 units of flow to the sink and source respectively. As a result, we end up pruning vertex 3 and all of its incoming and outgoing edges. With no further paths from s to t in G_L we terminate the iteration and merge the flow results into G. From here, we start another iteration by constructing a new layered network and repeating the process.

5.3 Related Work

While we focus on the MPM algorithm in particular, there are many known algorithms for solving the maximum-flow problem in the literature. Ford and Fulkerson [9] described the first original *flow augmenting path* algorithm. The authors, using their algorithm, proved the *max-flow-min-cut* [37] theorem. This theorem proposes that the maximum amount of flow possible from the source (s) to the sink (t) of some graph equals the minimum capacity removed from the graph when enough edges are removed such that no path exists from s to t. Later, Edmonds and Karp [7] proposed an improvement to this algorithm. While Ford and Fulkerson's [9] algorithm was unbounded and exponential in complexity, Edmonds and Karp [7] improved this to $O(|E|^2|V|)$ (bounded by vertices, V, and edges, E, in the graph).

Shortly thereafter, Dinic [6] described an $O(|V|^2|E|)$ algorithm for solving the max flow problem. Dinic's algorithm introduced the concept of a layered network. Constructed from some graph, G, the layered network contained only those edges of G such that each edge moves from a layer i to layer $i + 1$, where each layer represents

one step closer to t. Dinic's algorithm then constructs a *blocking flow* on this layered network such that at least one edge along all the possible paths from s to t becomes saturated with flow.

Malhotra, Pramodh Kumar and Maheshwari's [29] "MPM" algorithm provides a more efficient, potentially parallelizable method of finding blocking flows on the layered network with a complexity of $O(V^3)$. The algorithm makes use of several phases which we previously explained in Section 5.2. The authors essentially built on top of the previous work by Dinic, Edmonds and Karp, and Ford and Fulkerson.

Unlike the previous authors, Goldberg and Tarjan [12, 13] did not extend the layered network and flow augmenting path ideas, and instead designed a *push-relabel* method for solving the maximum flow algorithm. The push-relabel algorithm, with a complexity of $O(EV log(|V|^2/|E|))$, is unlike all other algorithms mentioned in this section in that it violates the conservation constraint. At times during the algorithm, edges may be over-saturated with flow. The idea here is that oversaturated edges will push back their flow in order to get back to a valid state. Goldberg and Tarjan's push-relabel algorithm exhibits asynchronous properties and is well suited for parallelization. These asynchronous properties, however, may result in performance issues on the very synchronous hardware of the GPU.

In the general parallel computing literature, Shiloach and Vishkin [45] developed a parallel maximum flow algorithm based on Dinic's [6] algorithm. Their algorithm also shares some concepts with the later max flow algorithm from Goldberg and Tarjan [12, 13], as vertices may be oversaturated with flow that must be pushed back at some point in time. Shiloach and Vishkin subdivide the graph amongst the

available processors and use a parallel BFS to generate a layered network and initiate the pushing of flow.

Anderson and Setubal [1] developed a parallel push-relabel algorithm on shared memory machines. Each processor works on subsets of the vertices in the graph, retrieving more from a shared queue when they complete their current workload. They measure up to an approximate 7 times speedup when running their algorithm on 20 processors.

Bader and Sachdeva [2] extended the work of Anderson and Setubal [1] and developed a cache aware implementation of the push-relabel algorithm on symmetric multiprocessor architectures. The authors focused on cache awareness in order to mitigate the issue of memory latency and speed and try to improve the overall speedup of the application. They further modify the algorithm to better support random graphs — Bader and Sachdeva note that the basic cache aware implementation performs well on dense graphs, but required more work to improve the performance on random level graphs. Random level graphs are graphs where vertices are organized into rows (forming an overall rectangular set of vertices) and each vertex on a row connects with three random vertices on an adjacent row. They do not provide any speedup results compared to sequential CPU code, but show that they achieve nearly four times speedup when executing their algorithm across eight processors when compared to execution on two processors.

On the GPU/CUDA side of the literature there exists a number of works investigating the design and performance of the push-relabel algorithm for solving the maximum flow problem. Hussein et al. [18] described one such GPU implementation

of Goldberg and Tarjan's [13] push-relabel algorithm. In their work, the authors implemented a series of optimizations that target grid graphs. One such example is the authors' development of a cache emulation system. With cache emulation, Hussein et al. enforce cooperation amongst threads when accessing data from global memory. In effect, the authors exploit global memory coalescing by taking advantage of the structured nature of a grid graph, and loading in tiles of vertex data from global memory at a time. The authors show that their algorithm achieves up to 4.5 times speedup compared to the fastest known sequential CPU implementation.

Vineet and Narayanan [56] later implemented a push-relabel max flow algorithm on the GPU that attempts to provide better load balancing between active and inactive vertices in a graph. In order to accomplish this, the authors developed an optimization that skips inactive nodes from processing. Vineet and Narayanan further investigated the impact compacting data in order to reduce global memory accesses had on the performance. Interestingly, they experimentally showed that the compaction schemes resulted in worse performance of the overall application. Further, the authors' described, implemented and tested two versions of the algorithm: one that uses atomic memory operations and one that does not. Like the work of Hussein et al., Vineet and Narayanan's algorithm is limited in scope to grid graphs.

Most recently, Harish et al. [15] described their implementation of a series of graph algorithms on the GPU. As was the case with the previous GPU work, Harish et al. also focused on the push-relabel algorithm for solving the maximum flow problem. The authors used a scan operation to identify vertices that will be active in a given phase and only assign threads to those vertices. Harish et al. concluded that high

degree graphs perform significantly better than graphs with a low degree. The authors explained this result by describing the lack of parallelism present in graphs with lower vertex degrees (that is, more linear graphs).

From the following works, we note that research in parallel computing for the maximum flow problem on the GPU has been focused on the push-relabel algorithm. In the current work, we consider the MPM algorithm on the GPU as our choice of max flow algorithm, as it contains properties we believe make it more suited for the GPU architecture. The MPM algorithm is synchronous and can be more readily mapped to an SIMD architecture unlike the push-relabel method, whose asynchronicity may not make it an ideal match for the GPU. We believe the design and implementation of the MPM algorithm will make a nice comparison piece to the existing push-relabel work.

5.4 MPM on the GPU

We begin the discussion on our GPU implementation of the MPM algorithm with a description of how we modified the phases. In Section 5.2 we described the six main phases of the MPM algorithm: construction of the layered network, construction of the blocking flow, pushing flow, pulling flow, pruning, and updating capacities in G. These phases work well in serial, as we can only complete one task at a time. In parallel, however, we have the added benefit of being able to perform operations at the same time, assuming that they do not require any consistency or synchronization between them. In the MPM algorithm, the pushing of flow and pulling of flow phases meet these requirements. Consider which layers of G_L the pushing and pulling of

flow phases work on. When pushing flow, we work only with layers L_i through L_t (where L_i contains v_m, and L_t contains t). Conversely, when pulling flow, we work only with layers L_0 through L_i (which contain s and v_m respectively). As a result, both phases never update or interact with the same edges (or vertices, outside of v_m) at any point in time, allowing us to execute them in parallel without requiring performance-degrading consistency checks.

Due to this lack of interaction, we merge the pushing of flow and pulling of flow phases into a single phase. Effectively, this merger allows us to work on both sides of the layered network (relative to v_m) in parallel, removing the synchronization barrier that would otherwise be in place between these two phases, and improving performance. Furthermore, due to the simplicity of the construction of blocking flow phase, which involves only a search for v_m, we merge it with the pushing and pulling of flow phase as well. As the search for v_m is very simple, and involves only a single search for a minimum potential value across all edges in G_L, we feel that it holds little interest and do not treat it as a separate phase in our discussion.

In our GPU implementation, we store only the residual capacity, $\tilde{c}(e)$, for each edge's flow data. That is, each edge stored on the GPU maintains only its residual capacity value, rather than $c(e)$ and $f(e)$. With this approach, we reduce the number of memory accesses required, as well as the memory storage requirements and computations required for an edge. In the same manner as our BFS algorithm, we use the two-array adjacency list structure for storing the vertices and edges of G (see Figure 4.2 in Section 4.3). As we have previously discussed the set up and use of this structure in Section 4.3 we do not repeat the material here. Finally, we use a number

of one-dimensional arrays to store the state of flow in the graph:

1. `Layers` — Stores the layer each vertex in G_L belongs to, if any.
2. `EdgeState` — The state of an edge in G_L, either ACTIVE or INACTIVE.
3. `EdgeFlow` — The residual capacity along each edge in G.
4. `InPotential` — The current in-potential of each vertex in G_L.
5. `OutPotential` — The current out-potential of each vertex in G_L.

Note that EdgeFlow stores the residual capacity of edges in G, not in G_L. By doing this, we avoid the phase requiring us to merge the flow results from G_L into G. Instead, we simply work on G's flow data directly.

In the end, we compress the original six phases of the MPM algorithm into three: layered network construction, pushing and pulling of flow and pruning. A description of our design and implementation for each of these phases follows.

Layered Network Construction

We use a breadth-first traversal of G in order to construct the layered network, G_L. The breadth-first traversal we use for this phase differs only slightly from that we described in Chapter 4. As was the case with our BFS algorithm, we run multiple iterations of the kernel in order to construct the layered network. Each iteration corresponds to the construction of a single layer in G_L. A CPU control loop manages the GPU phases, continually launching new phases for each layer until the sink node, t, has been assigned to a layer. At this point, we no longer need to continue the traversal of G as at least one path now exists between s and t in G_L.

In terms of thread responsibilities, we assign a single thread to cover a single vertex in G. With each iteration of the kernel, each thread assigns adjacent vertices to the next layer following the rules for layered network construction described in Section 5.2. We further mark the edges connecting this thread's vertex with the vertices added to the new layer as "ACTIVE". We need to know ahead of time which edges we can traverse when pushing and pulling flow in G_L, hence we mark edges accordingly in the `EdgeState` array.

Another minor modification we make to our BFS algorithm updates the in-potential and out-potential of each vertex. We perform this step in order to reflect the activation of an edge in G_L — the in-potential and out-potential of vertices in G_L rely on which edges are marked as "ACTIVE". We use a simple auxiliary kernel to update the in and out-potential data. Within this kernel, each thread simply loops through its "ACTIVE" edges and updates the in and out-potential accordingly. Rather than threads potentially updating this data at any time during the execution of the more complex BFS kernel, we enforce that they update this data all at the same time, within the same, simple kernel. As a result, we exploit global memory coalescing and improve the overall performance of this necessary operation.

As the layered network construction kernel does not differ significantly from our BFS kernel, we do not provide the pseudocode.

Pruning of G_L

As we described earlier, the pruning phase involves removing useless vertices (and edges) from G_L. As was the case with the layered network construction phase, we

map one thread to one vertex (this further implies that a thread is responsible for all of the edges attached to a vertex). If the potential of a vertex equals 0, we prune that vertex from G_L, as it clearly does not contain any useful outgoing and/or incoming edges (at least one of these sets must be empty, or have a total residual capacity of 0).

When pruning a vertex, however, we need to check whether the in- or out-potential was 0. If the in-potential of a given vertex is 0 then we remove the outgoing edges of this vertex, whereas if the out-potential is 0 we remove incoming edges. In order to reduce thread divergence in the kernel, we assign temporary pointers pointing to the in or out-potential arrays, ensuring threads performing work do not diverge when the actual pruning steps occur (and, instead, push the divergence to the simple task of setting a pointer). As the removal of incoming edges requires "backwards" edges (that is, rather than $e = (u, v)$ we need $e' = (v, u)$ which does not exist in G), we also assign a pointer to the graph to use (containing forward or backward edges). In this respect, we store both the original graph data, as well as the "backwards" graph, called G', which stores all edges in reverse. This increases the memory requirements of our algorithm, but allows us to quickly locate both the original (forward) edges as well as backward edges.

Algorithm 4 CPU Layer Network Pruning Loop

```
1: while At least one vertex was pruned in the previous phase do
2:     Call LayerNetworkPruningGPU
3: end while
```

Similar to the layered network construction above, the CPU loop controller invokes

multiple iterations of the pruning phase on the GPU. The CPU controller continually invokes the GPU pruning kernel until an iteration with no vertices pruned has been completed. In order to check for termination conditions, the CPU retrieves 4 bytes of data after each kernel invocation in order to determine if any vertices have been pruned in the last iteration. We provide the pseudocode for the CPU control loop in Algorithm 4, and the GPU kernel in Algorithm 5. Lines 5—15 handle the intermediary pointers, which allows us to control some of the easily guaranteed thread branching that occurs in lines 17—20 (as not all threads contains the same number of edges they consider in the `for` loop). The `AtomicSub` operation on line 18 ensures consistency in the event that multiple threads attempt to write to the same data element in `EdgeFlow` at the same time. This instruction reduces performance, as threads writing to the same location must proceed with their operations in serial, but is required in order to ensure write-after-write hazards do not occur.

Push/Pull Flow

As we described earlier, we concatenate the pushing and pulling of flow into a single GPU kernel in order to perform both of these tasks in parallel. In this kernel, each thread is, again, responsible for a single vertex in G. Pushing flow is handled by saturating as many edges as possible at each vertex from v_m to t using forward edges in G_L. Pulling flow, on the other hand, uses backward edges in G_L (as read from G') and "pushes" flow from v_m to s along the backward edges (thus, "pulling" flow from s).

Before the pushing or pulling of flow can occur, however, we must locate v_m. A

GPU kernel performs this search and executes a minimum-reduction on the vertex potential data in order to discover the position and potential of that minimum vertex. We provide the pseudocode of this kernel in Algorithm 6.

With v_m found we can now begin the pushing and pulling of flow, which we handle using another kernel. Each iteration of the GPU kernel for the pushing/pulling of flow pushes flow across a single layer and pulls flow from a single layer in G_L. Similar to the pruning phase, we use temporary/intermediary pointers to reduce the thread divergence, as a single kernel handles both the pushing and pulling of flow. This phase requires a state array (VertexState[]) to maintain a record of which vertices are pulling and which are pushing, and another state array to keep track of the total amount of flow waiting to be pushed/pulled at each vertex (FlowAtVertex[]). When a vertex pulls (pushes) flow, it sets the state array of the vertex receiving the flow to indicate this new vertex will pull (push) in the next iteration.

As expected, this phase requires the most memory operations out of all the phases. We must keep track of not only which vertices and edges are active, but also the various flow states, flow amounts, and other such data across the ACTIVE edges and vertices. Each thread must also update the residual capacity value for edges that it pushes flow across, as well as the in or out-potential for both its own vertex as well as the vertices receiving flow. Because we must invoke the kernel multiple times in order to complete the pushing/pulling, we cannot store this data in shared memory for fast retrieval. Due to the high memory complexity of this kernel, we expect that it will perform the worst, relative to a sequential CPU implementation, as the number of irregular reads and writes to memory are high. Unfortunately, such problems are

inherent to algorithms with irregular properties, and while we have made some efforts to reduce their performance-degrading effects (such as the use of intermediary pointers to avoid high thread divergence), many other occurrences, such as the irregular reads and writes as the various threads push/pull flow across edges that are unknown ahead of time, are simply unavoidable.

Algorithm 7 CPU Push/Pull Flow Control Loop

```
1: while at least one vertex needs to push/pull flow do
2:     Call Push/PullFlowGPUKernel
3: end while
```

As was the case with the other phases, a CPU control loop (shown in Algorithm 7) repeatedly invokes the Push/Pull kernel until it encounters some termination condition. In this case, we terminate the algorithm when no new vertices have had flow pushed or pulled to them in a given phase. As with the other two phases, this requires a negligible 4 bytes of data transferred from the GPU to the CPU.

The Complete MPM Algorithm

Having covered the individual phases of our MPM algorithm, we can now piece the parts together and describe the algorithm as a whole. Algorithm 8 shows the high-level CPU framework used to drive the MPM algorithm computations on the GPU. Currently, the CPU only controls the invocation of GPU kernels and the termination condition checks. The GPU performs all other computations. This allows for minimal memory transfers between the CPU and GPU, which are generally costly and time consuming. As discussed, in each phase the CPU requires only 4 bytes of data

following each GPU kernel invocation in order to check the termination condition for each individual phase as well as the MPM algorithm as a whole.

Algorithm 8 MPM Maximum Flow Algorithm Framework

Require: $G = (V, E, s, t)$

1: **while** A path exists from s to t in G **do**
2: $G_L \leftarrow G$ partitioned into a layered network
3: **while** A path exists from s to t in G_L **do**
4: Find vertex v with min. vertex potential
5: Push/pull flow from v to s, t
6: Prune G_L
7: **end while**
8: **end while**

5.4.1 Active Vertex Modification

We recognize the same critical flaw in our implementation of the MPM algorithm on the GPU that we noted with the BFS: at every iteration of every kernel for every phase all threads covering all vertices in G are launched. Clearly this is sub-optimal as: (1) the pruning and pushing/pulling of flow phases work on vertices in G_L and not G, and (2) not all vertices in G_L will be active during every iteration of every phase (consider the pushing/pulling of flow which works on a layer-by-layer basis — clearly there will be, at most, 2 layers of active vertices in G_L for a given iteration!).

In order to improve on our algorithm, we modify each phase such that they perform an extra operation that determines which vertices will be active in the next iteration,

and store the vertex numbers in an array. As we discussed in the BFS chapter, Harish et al. [15] describe this technique, and we modify it for use in our algorithm. Our hypothesis at the time was that the effect of this strategy should have a noticeable impact on performance, as we now have the entire thread grid for a kernel saturated only with threads that will perform useful work. As we originally observed with the BFS, however, this strategy did not result in improved performance overall. We believed, however, that the differences in the pruning and push/pull kernels from the BFS/layered network construction kernels will likely result in the active vertices modification improving performance.

While the underlying active vertices technique was tuned to the requirements of the kernel (and will be explained for each below), the common theme for each was the use of a scan operation to determine the active vertices. In essence, each kernel keeps track of vertices that have been activated (using a scan state array with one slot for each vertex where 0 is inactive and 1 is active), and we perform a scan operation on this state array to determine the actual vertex indices for each active vertex. Similar again to our work on the BFS, we use the high performance parallel scan operation that is a part of the CUDPP library [43, 42] in order to accomplish this.

We conclude this section with a description of the modifications made to each phase in order to support the active vertices strategy.

Modified Layered Network Construction Phase

Due to the reduction of performance we observed in the BFS active vertices modification, we neglected to investigate the potential of this strategy for layered network

construction. As this phase is nothing more than a breadth-first traversal of a graph with a few minor modifications, we believe that no performance improvement is possible via this strategy.

Modified Pruning Phase

In the pruning phase, we can only mark threads as active if we believe they *may* cover a vertex that requires pruning. Clearly, in the first iteration of this kernel all threads must be marked as active, as, potentially, any vertex could be pruned. We simply cannot have this information ahead of time. When a vertex is pruned, we mark all neighboring vertices (using backward edges in the case of a vertex with zero out-potential, and forward edges for vertices with zero in-potential) with a one value in the vertex flag array. The next iteration of the kernel will then launch with threads considering only these vertices for pruning. As was the case with the original pruning phase, we iterate until no further vertices are considered for pruning in a given iteration.

Modified Push/Pull Flow Phase

For the pushing/pulling of flow phase, we structure the modifications in a manner similar to the modified pruning phase. Rather than start with all vertices active, however, we start with only v_m active. As flow is pushed/pulled to neighboring vertices, we mark those vertices as active in the vertex flag array. The next iteration of the kernel will then launch threads covering only those vertices that must now push or pull flow. We iterate this phase until flow has reached both s and t — the point at which no vertices are active in a given iteration, as s contains no incoming edges,

and t no outgoing edges in G_L.

5.5 Results

We averaged the performance results for the MPM algorithm across at least 5 random graphs for each vertex count (with an average degree of 12). We ran the MPM algorithm against each of these graphs 25 times in order to gather averaged execution time results.

For our initial results, we compared the overall execution time between all three implementations (CPU, GPU, and active vertices modification). We show that our persistence with the active vertices strategy paid off in Figure 5.2. From the results, we can see that both GPU implementations perform significantly better than the sequential CPU implementation, but also that the active vertices modification resulted in further performance improvements over the base GPU version. Previous work by Solomon et al. [48] tested only a small number of graphs over a small number of iterations. By improving our tests with many more graphs and iterations for this work, the averaged speedup of the base GPU MPM implementation was reduced from 8 [48] to approximately 6.5. However, the averaged speedup of the active vertices version of the GPU MPM implementation was approximately 9.5, higher than our previously optimistic results in Solomon et al. [48] without the active vertices strategy.

We further investigated the performance of each phase of the MPM algorithm individually, in order to gain a greater understanding of which computations perform well on the GPU. Figure 5.3 provides a graph detailing the total execution time taken by the layered network construction phase in a sequential CPU implementation and

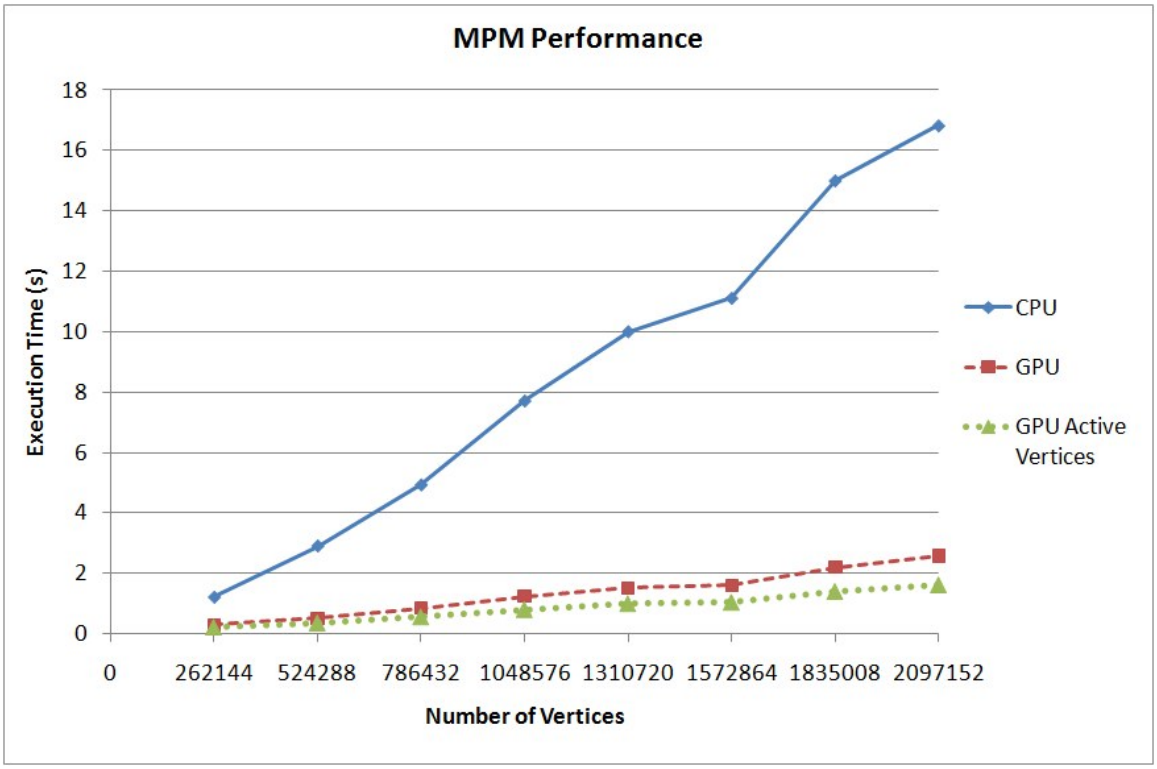

Figure 5.2: Performance of the MPM Algorithm.

a parallel GPU implementation. As we expected, given the performance of the BFS, the GPU implementation performs significantly better than the CPU implementation. Unfortunately, as was also the case with the BFS, the active vertices modification is unlikely to provide any performance improvements.

From Figures 5.4 and 5.5 we can see that the results for pruning and pushing/pulling of flow appear to be much better than that of the layered network construction phase, or the BFS, in terms of the active vertices modification. Table 5.1 shows that both of these phases see their execution times improved upon by over 50% on average when switching from the basic kernel to the active vertices kernel. We believe these phases see such a tremendous improvement while the BFS/layered

network construction did not, for two reasons:

1. Their kernels are more complex than the layered network construction kernel, with more computations and many more accesses to global memory.

2. The layered network construction phase contains different patterns of change in the active threads compared to the pruning and pushing/pulling of flow phases. For layered network construction the number of active vertices (in the best case) grows by a factor of $a * d$ where a is the number of active vertices in the previous iteration and d is the average vertex degree. That is to say, the active vertex count has the potential to grow very quickly. Thus, in later phases, the number of active vertices we write to the state array are large — in these cases the overhead of the scan operation becomes far greater than just letting a relative few inactive threads slip past. Pruning, however, starts with all vertices active and then rapidly diminishes based on the number of vertices pruned at each iteration. For pruning, the active vertex count decreases with each iteration. Finally, for pushing/pulling flow, we have a somewhat consistent active vertex count after the first iteration. This is simply because the vertices that have the potential to be active must be located in one of the two layers considered in the current iteration.

When we combine these reasons with the ability to fill warps with only those threads that are active, it begins to become apparent as to why we see such a large performance improvement in these two phases with the active vertices strategy. The high potential for growth of the (relatively) simple layered network construction kernel cannot take full advantage of the active vertices technique. In this kernel, we likely

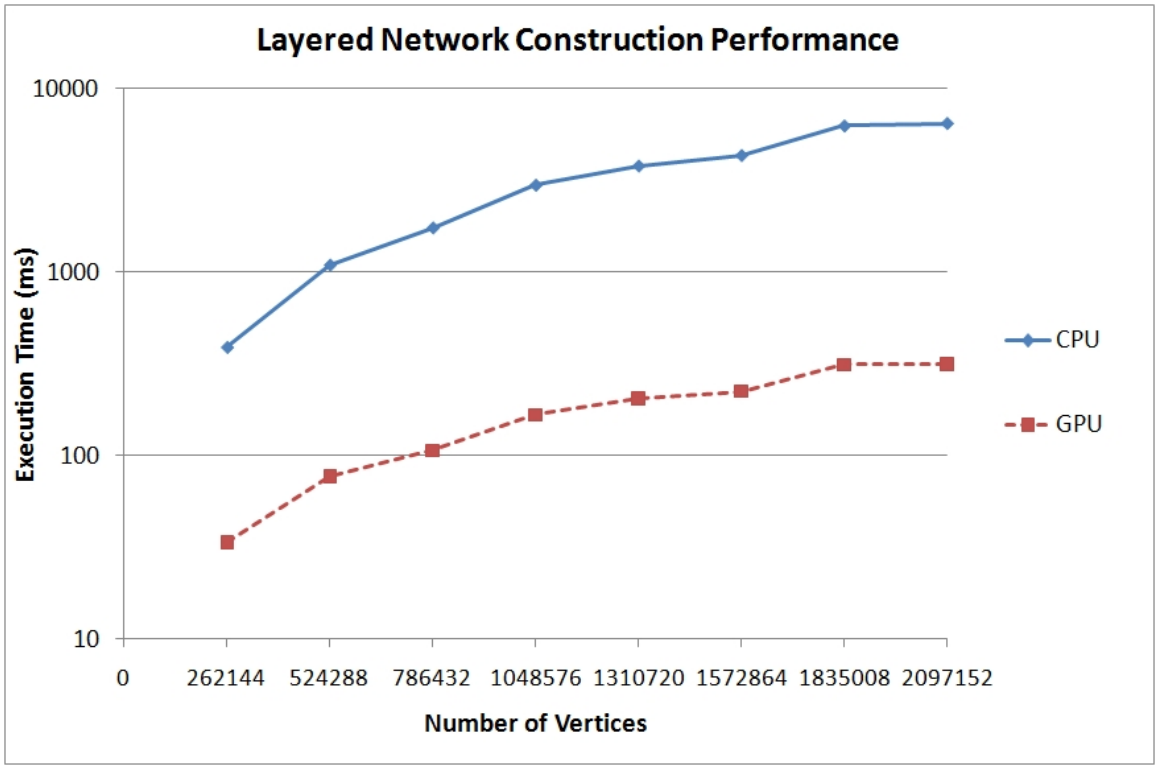

Figure 5.3: Total execution time taken by layered network construction phase.

	% Improved
Layered Network Con.	—
Pruning	61.0%
Push Pull Flow	56.5%

Table 5.1: Average Performance Improvement Between Normal GPU and Active Vertices Implementations

have a large number of threads active at each iteration, reducing what performance improvements we hoped to achieve by ignoring inactive threads. In contrast, the reduction or consistent nature of the pruning and push/pull flow phases respectively, coupled with their greater complexity, results in the active vertices strategy having a greater impact on performance improvement of the kernels.

Next, we measured the approximate percentage of the run time each phase con-

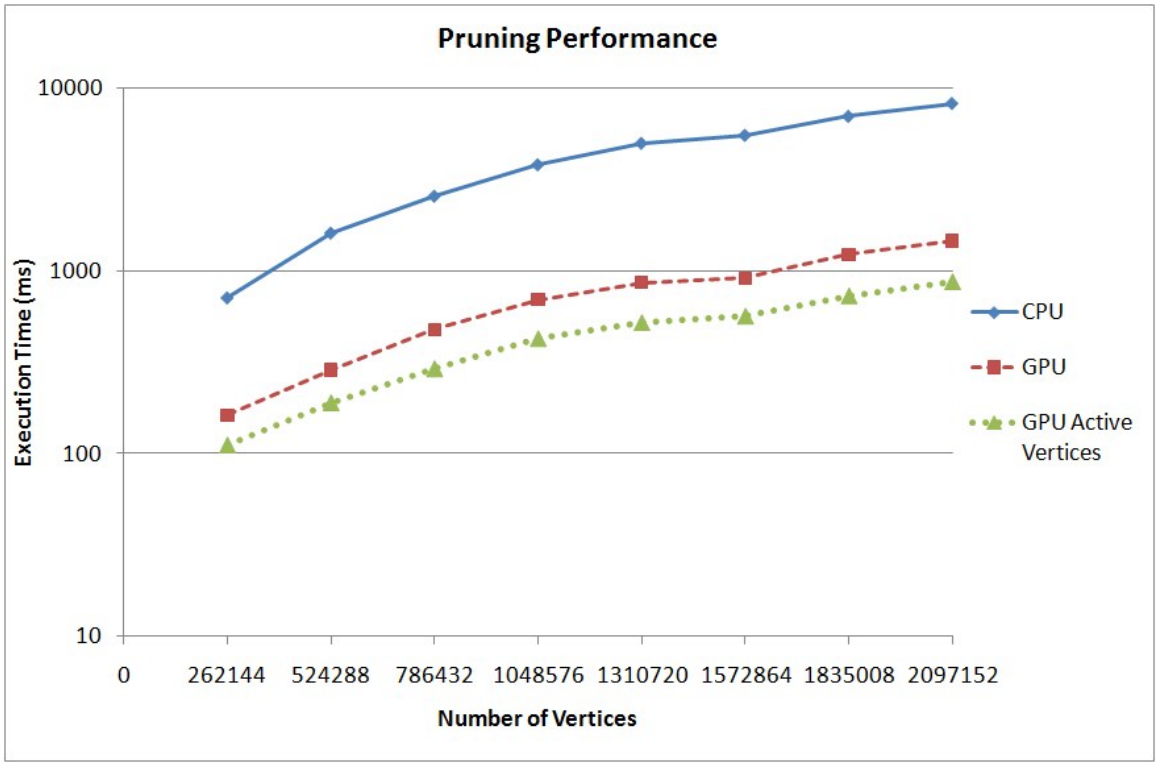

Figure 5.4: Total execution time taken by pruning phase.

sumes and provide the results in Table 5.2. For these results, we ignore miscellaneous computations outside of the phases. We also measured the average speedup between the GPU phases, using the best implementation of each kernel (active vertices strategy for all but the layered network construction) versus the sequential CPU implementation and place the results in Table 5.3. When we look at the data from Tables 5.1, 5.2, and 5.3, we observe some interesting results. Firstly, while the pruning phase consumes slightly over half of the phase execution time it saw the greatest improvement with the active vertices strategy, a 61% improvement. With the active vertices strategy we have substantially improved the most time consuming phase in the algorithm, thus providing us a significant performance improvement in the algorithm as a whole.

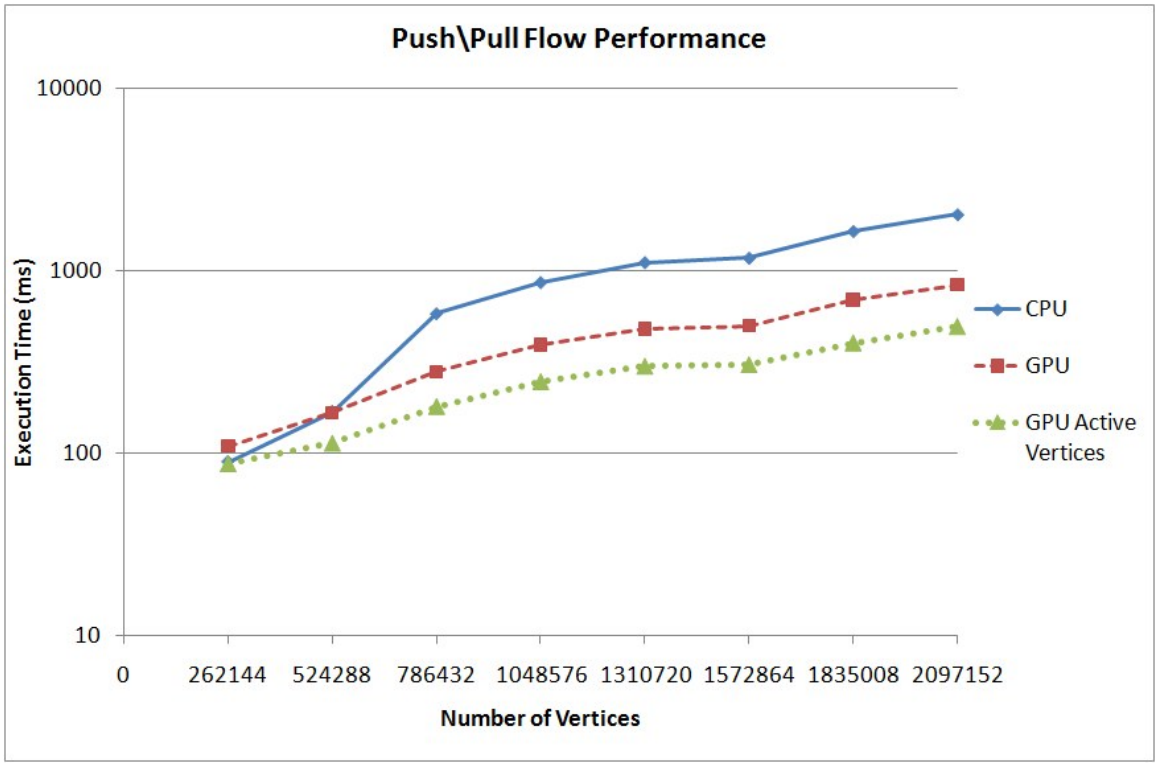

Figure 5.5: Total execution time taken by push/pull flow phase.

While the push/pull flow phase also saw a tremendous performance improvement, at 56.5%, the phase only makes up 19% of the phase execution time, resulting in a much smaller impact on the improvement of the overall performance. Finally, while the layered network construction phase displayed the best speedup out of the three, it does not gain any improved performance via the active vertices implementation. The layered network construction phase also represents a large portion of the execution time, at 30% of the share, which makes the inability to improve its performance an unfortunate situation.

Overall, however, we believe the final results are impressive. Not only does the active vertices version of the GPU code feature improved performance over the basic

	Execution Time %
Layered Network Con.	30%
Pruning	51%
Push Pull Flow	19%

Table 5.2: Average division in execution time between all three phases.

	Speedup
Layered Network Con.	18
Pruning	9
Push Pull Flow	3

Table 5.3: Average speedup of best implementation versus sequential CPU implementation

GPU code, but we also see a reasonable level of speedup compared to a sequential CPU implementation. While we never reach the speedup values we saw with the BFS, we expected this. The BFS represented a simple algorithm with irregular properties, whereas the MPM algorithm represents a more complex algorithm with irregular properties. The differences in memory and computational complexity have been well covered, and they result in the performance on the GPU suffering somewhat. We feel that the fact that we can still achieve nearly a ten times performance improvement over the CPU helps to show that the GPU still provides a reasonable level of performance when working with complex irregular problems.

5.5.1 Comparison to Existing Work

Finally, we attempt to compare our results to Harish et al.'s [15] GPU implementation of the push-relabel algorithm. Unfortunately, the author's paper does not provide any hard numbers for the majority of the performance result, nor do they

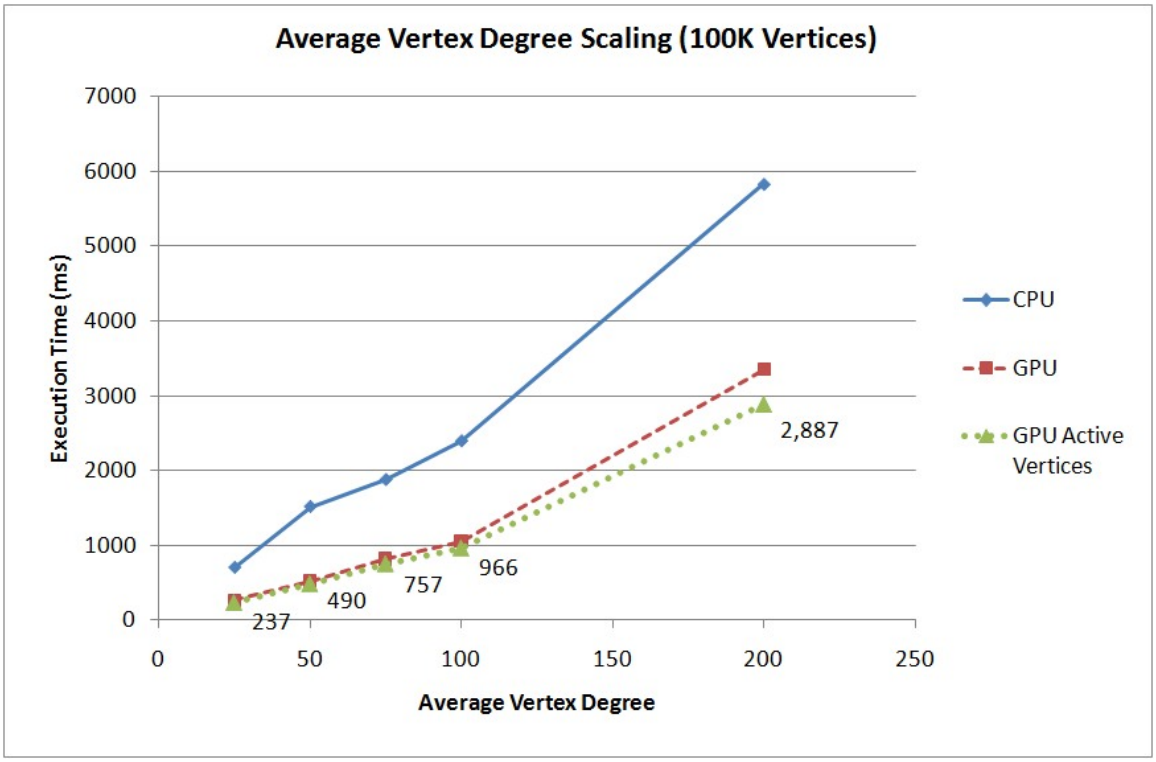

Figure 5.6: Performance scaling by average vertex degree.

publish the exact random graphs used for their main performance tests. Further, they use superior GPU hardware with a higher clock speed, more SMs, and more global memory compared to our testing (their hardware included a GTX 280 and Tesla S1070 compared to our GTX 260). As a result, the comparisons we make here are not concrete, but merely an interesting diversion. With that said, Harish et al. did publish some hard performance numbers when they tested the vertex degree scaling of their algorithm. We performed the same type of tests, with random graphs similar to theirs (100K vertices with varying average vertex degrees). Unfortunately, due to the lack of available memory on our GPU we could not test extremely high vertex degrees.

Figure 5.6 provides a graph with the vertex degree results (the numbers posted at each data point are the average execution times for the active vertices implementation). Comparing our results to those of Harish et al. [15] we see that our MPM implementation comes very close to their performance levels for average vertex degrees of 100 and 200 (808 and 2976 milliseconds [15] respectively). We want to reiterate that we do not believe we can draw a full comparison from these results as our works use different random graphs from one another. We do consider it of interest, however, to note that two very different maximum flow algorithms appear to perform similarly on the GPU.

Another observation we make from these results deals with the performance of the MPM algorithm degrading as the average vertex degree increases. While this may seem somewhat counter-intuitive — higher vertex degrees should potentially equal more parallelism — having a graph composed of vertices with very high degrees results in much larger data sizes, as the number of edges in the graph increases. This increase in the overall amount of data that may or may not require processing at some point in time during the algorithm explains the increase in execution time. To explain this further, consider that one thread covers one vertex in all phases of our MPM implementation. As the average degree increases, each thread must perform much more work as it has many more edges to cover in its vertex.

5.6 Summary

As expected, the level of speedup we observed for the MPM algorithm was lower than both the BFS and option pricing algorithms we previously investigated. Similar

to the BFS algorithm, we encountered issues with the inability to fully exploit global memory coalescing and were unable to ensure we provided each thread with equal workloads. Unlike the BFS, however, we observed a fairly significant performance improvement with the active vertices modification. Despite the overhead inherent to this technique, we increased the speedup of our implementation from 6.5 up to 9.5. While the speedup results were nearly halved from what we observed with the BFS, we feel that these results still help to show that the GPU allows for adequate performance when solving even a complex irregular problem. Outside of the rare occurrence of super-linear speedup, we expect to require at least 10 traditional CPU processing cores to achieve the same speedup results as we saw on a single mid-range GPU.

For the next chapter, we investigate a very different algorithm. Thus far, each algorithm contained either regular or irregular properties in their entirety. Particle swarm optimization for solving the task matching problem, however, features a number of very regular, data-parallel segments coupled with some areas of irregularity. Due to this mix, we expect that the performance will fall between the BFS and option pricing algorithm. We do not expect greater performance than the option pricing algorithm due to the irregular sections, but expect the performance to greatly exceed that of the BFS due to the large amount of regular areas. We expect that the particle swarm optimization algorithm will help us understand the design, optimization, and performance of algorithms on the GPU to an even greater degree due to the dramatic differences between it and the previous algorithms we have investigated.

Algorithm 5 GPU Layer Network Pruning Phase

Require: $G = \{V, E\}$, $G' = \{V, E'\}$, Layers[], EdgeState[], EdgeFlow[], InPotential[], OutPotential[]

```
1: tid = blockIdx.x * blockDim.x + threadIdx.x
2: if tid ≤ |V| then
3:   if Layers[tid] ≥ 0 AND Vertex Potential = 0 then
4:     Layers[tid] = −1
5:     if InPotential[tid] = 0 then
6:       //We will prune outgoing edges
7:       Graph = G
8:       MyVertexPotentialPtr = OutPotential
9:       AdjVertexPotentialPtr = InPotential
10:    else
11:      //We will prune incoming edges
12:      Graph = G'
13:      MyVertexPotentialPtr = OutPotential
14:      AdjVertexPotentialPtr = InPotential
15:    end if
16:    MyVertexPotentialPtr[tid] = 0
17:    for each e = (u, v) in G where u = tid do
18:      AtomicSub(AdjVertexPotentialPtr[v], EdgeFlow[e])
19:      EdgeState[e] = INACTIVE
20:    end for
21:  end if
22: end if
```

Algorithm 6 GPU Push/Pull Flow Kernel

Require: $G = \{V, E\}$, $G' = \{V, E'\}$, Layers[], VertexState[], EdgeState[], EdgeFlow[], FlowAtVertex[], InPotential[], OutPotential[]

```
1: tid = blockIdx.x * blockDim.x + threadIdx.x
2: if tid ≤ |V| then
3:    if NodeState[tid] = PUSH then
4:       MyState = PUSH, GraphPtr = G
5:       MyVertexPotentialPtr = OutPotential, AdjVertexPotentialPtr = InPotential
6:    end if
7:    if NodeState[tid] = PULL then
8:       MyState = PULL, GraphPtr = G'
9:       MyVertexPotentialPtr = InPotential, AdjVertexPotentialPtr = OutPotential
10:   end if
11:   if (MyState = PUSH OR PULL) AND (EdgeFlow[tid] > 0) then
12:      remainingFlow = FlowAtVertex[tid]
13:      for Each edge e = (u, v) in GraphPtr where u = tid do
14:         VertexState[v] = MyState
15:         Update EdgeFlow[e], remainingFlow, and potentials of u and v
16:         if EdgeFlow[e] = 0 then
17:            EdgeState[e] = INACTIVE
18:         end if
19:         Break from for loop if remainingFlow ≤ 0
20:      end for
21:   end if
22: end if
```

Chapter 6

Multi-Swarm Particle Swarm Optimization

Having completed our investigation of two irregular problems and one regular problem, we turn our attention towards the computationally difficult problem of task-matching. The final algorithm we investigate to solve this problem is a multi-swarm Particle Swarm Optimization (PSO) algorithm. We chose a multi-swarm PSO algorithm as it is an algorithm that contains significant degrees of parallelization and, thus, we found it interesting to experiment with on the GPU. We target this algorithm at solving the task matching problem not only because it is a real-world problem, but also because solving this problem with PSO results in some areas of irregularity in terms of data-access patterns. We had already investigated a thoroughly data-parallel algorithm with lookback option pricing, so we chose to move things along a different path here.

We start this chapter off with an introduction to the PSO and multi-swarm PSO

algorithms, a description of the task matching problem, and the related work for this area. From there, we move on to a description of our PSO algorithm for solving the task matching problem and then describe our GPU implementation, and its performance.

6.1 Introduction to PSO and Multi-Swarm PSO

The PSO algorithm, first described by Kennedy and Eberhart [22], is a bio-inspired or meta-heuristic algorithm that uses a *swarm* of *particles* which move throughout an area, searching for an optimal solution. In essence, the PSO algorithm was inspired by swarm activities in nature. For example, consider a flock of birds searching for food. As one bird finds food, others may move in towards the same general location as that area is known to have that desirable property (sustenance).

In PSO, our particles function like the aforementioned birds. Each particle searches for some optimal value, and collaborates with other particles in the swarm. We refer to the area that the particles search through as the solution space. Each point in the solution space (represented by a real number for every dimension) represents one solution to the optimization problem (not necessarily an *optimal* solution, however). The solution space itself may be composed of as many dimensions as required for the optimization problem at hand. For example, searching across a land mass for food requires only two or three dimensions in the solution space (representing positions along the x, y, and optionally, z, axes). Other problems, however, may require solution spaces of higher dimensionality. In other words, PSO works with an n dimensional solution space, where $0 < n < \infty$.

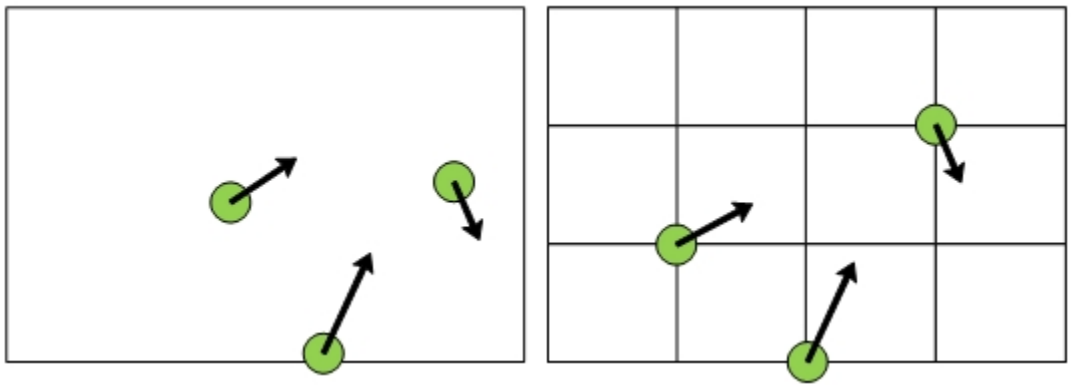

Figure 6.1: Continuous (left) and Discrete (right) solution spaces.

Furthermore, we can define a solution space as either continuous or discrete. In the case of a continuous solution space, each dimension contains an infinite number of *real* points. That is to say, there does not exist any discrete, exact points along the dimensions defining where particles may lie — in a continuous solution space particles may exist at any possible location. Conversely, a discrete solution space contains, along each dimension, a set of discrete points. Unlike the continuous space, which allows particles to fall wherever they please, the discrete space enforces that particles must be located on one of the points, and nowhere else. We provide a simple example of continuous and discrete 2-dimensional solution spaces in Figure 6.1. Note that the particles (the circles in Figure 6.1) are constrained to the gridlines (discrete points along the x and y dimensions) in the discrete solution space, while the continuous solution space allows the particles to be located at any point.

Particles move throughout the solution space over the course of the algorithm and, as a result, require a position and velocity. We represent the position of particle i at iteration j of the algorithm with X_i^j and the velocity with V_i^j. As we work with an

n-dimensional solution space, X_i^j and V_i^j contain n components (or elements) each — one for each dimension in the solution space.

Now that we have attached a position and velocity to our particles, we need some method of testing the optimality of their current location in the solution. Each particle tests the fitness (or optimality) of the solution at its current location and uses this information to decide on where it will move next. The fitness function uses a formula to determine the optimality of a point in the solution space (a given solution for the optimization problem). This function must be intrinsically tied to the specific optimization problem at hand, as every problem contains its own definition for what makes a solution optimal. For example, if we were working with a problem of finding an area with the greatest deposits of food, we would set the fitness function to provide a value tied to the amount of food in an area. The greater the total mass of food in an area, the greater the value the fitness function returns. More complex problems, such as the problem we investigate (the task matching problem) require more complex fitness functions.

Similar to the MPM algorithm, we will describe the actual PSO algorithm in piecemeal fashion by splitting it up into phases. In this case, we have four main phases to PSO: the initialization of the swarm, the updating of the velocity and position of each particle, updating fitness, and the updating of the (local and swarm) best positions.

6.1.1 Initialization

The initialization phase involves initializing the particles within the swarm. We provide each particle with a random starting position and a random starting velocity.

Recall that X_i^j (position) and V_i^j (velocity) contain n elements, corresponding to the n dimensions in the solution space. Thus, we must initialize all n elements for each. We set the random starting position along each dimension such that:

$$X_{\min,i} < X_i^0 < X_{\max,i}$$

where $X_{\min,\,\mathrm{i}}$ and $X_{\max,\,\mathrm{i}}$ represent the minimum and maximum positions possible in dimension i. We randomly assign the velocity for a particle as any value between 0 and $X_{\max,i}/2$.

6.1.2 Velocity and Position Update

In order to have these particles move throughout the solution space we must provide them with a velocity value. We follow the modified PSO algorithm as established by Shi and Eberhart [44]. These authors update the velocity of a particle, i, using Equation 6.1.

$$V_i^{j+1} = w * V_i^j + c_1 * \mathrm{rnd}() * (X_{\mathrm{Pbest}} - X_i^j) + c_2 * \mathrm{rnd}() * (X_{\mathrm{Gbest}} - X_i^j) \qquad (6.1)$$

where X_i^j is the particle's current location, X_{Pbest} is the particle's local best position, X_{Gbest} is the global/swarm best position, and rnd() generates a uniformly distributed random number between 0 and 1. w, the inertial weight factor, along with c_1 and c_2 provide some tuning of the impact the previous velocity, V_i^j, X_{Pbest}, and X_{Gbest} have on the particle's updated velocity. Typically, c_1 and c_2 do not change throughout the iterations of PSO. Shi and Eberhart [44], however, proposed that changing w from iteration to iteration may improve the performance or solution quality of the algorithm. The authors recommend that w be started at 0.9 and slowly reduced such

that $w = 0.4$ by the final iteration of the algorithm. Once we update the velocity, the particle changes its position using Equation 6.2.

$$X_i^{j+1} = X_i^j + V_i^{j+1} \tag{6.2}$$

6.1.3 Fitness Update

The next step of the PSO algorithm involves updating the fitness of each particle, based on its current position in the solution space. As the fitness update phase is tied directly to the optimization problem, we leave the discussion of this phase until our explanation of our GPU implementation.

6.1.4 Local and Swarm Best Position Update

In the last phase of PSO we update the local best position and values for each particle as well as the global (or swarm) best position and value for the swarm as a whole. To start, each particle compares its current fitness value with the best value it found in the previous iterations (local best value). If the current fitness value is greater, the particle then replaces its local best value with its current fitness value. We maintain a record of the position of the local best value in X_{Pbest} as we use this value in the velocity update Equation 6.2. After each particle updates their respective local best value, we compare the best of the local best values against the best value found by the entire swarm (global best value) thus far. If this "best-of-the-best" value represents an improvement over the current global best value, then we replace the global best value. As was the case with the local best, we update the global best position, X_{Gbest}, as well.

6.1.5 The Complete PSO Algorithm

With all of the phases of standard PSO defined, we can build the algorithm as a whole. In Algorithm 9 we provide the basic high-level pseudocode for PSO. In this code we terminate the algorithm after a given number of iterations have been executed. We note that this is not the only choice of a termination condition. If we value solution quality over execution time we could terminate the algorithm when the global best value surpasses a threshold value. As we are concerned about performance, and the ability to deterministically measure the performance of a non-deterministic algorithm, we use the first option, and terminate processing after a set number of iterations.

Algorithm 9 Basic PSO Algorithm

```
Randomly disperse particles into solution space
for i = 0 → numIterations do
  for all particles in swarm do
    Compute fitness of current location
    Update X_Pbest if necessary
    Update X_Gbest if necessary
    Update velocity
    Update position
  end for
end for
```

6.1.6 Multi-Swarm PSO

As we are working with the highly parallel GPU hardware, we wanted to investigate the potential of a highly parallel PSO algorithm on the architecture. As a result, rather than designing and implementing a standard PSO algorithm we consider a variant of PSO that collaborates amongst multiple swarms. With multiple swarms we have the potential for more particles, and, thus, more parallelism. We further hypothesized that such a variant of PSO may provide higher quality solutions than we would otherwise generate with a large number of particles within a single swarm. Finally, as we will discuss in Section 6.3, there are a few examples of work related to (standard) PSO on the GPU, but none besides our own that investigates multi-swarm PSO on the GPU.

The particular multi-swarm PSO algorithm we choose, described by Vanneschi et al. [53], collaborates amongst swarms by replacing some of a swarm's "worst" particles with its neighboring swarm's "best" particles. By best and worst we refer to the fitness of the particle relative to all other particles in the same swarm. This swap occurs every given number of iterations, and forces communication among the swarms, ensuring that particles are mixed around between swarms. Further, Vanneschi et al. [53] use a repulsion factor for every second swarm. This repulsive factor repulses particles away from another swarm's global best position (X_{FGBest}) by further augmenting the velocity using the equation:

$$V_i^{j+1} = V_i^{j+1} + c_3 * \text{rnd}() * f(X_{\text{FGbest}}, X_{\text{Gbest}}, X_i^j) \tag{6.3}$$

where function f, as described by Vanneschi et al. [53], provides the actual repulsion force, and c_3 represents another tuning factor similar to the c_1 and c_2 values of stan-

dard PSO. We believe that this algorithm represents a good fit for the GPU, as it combines the potential for high degrees of parallelism with the iterative, synchronous nature of the PSO algorithm.

6.2 Introduction to the Task Matching Problem

The task matching problem represents a significant problem in heterogeneous, distributed computing environments, such as grid or cloud computing. The problem involves determining the optimal assignment, or *matching*, of tasks to machines such that the total execution time is minimized. More specifically, if we are given a set of heterogeneous computing resources and a set of tasks, we want to match (assign) tasks to machines such that we optimize the time taken until all machines have completed processing of their assigned tasks. We refer to this measure of time that we look to optimize as the makespan, which is determined by the length of time taken by the last machine to complete its assigned tasks.

We provide an example of one (sub-optimal) solution to a task-matching problem instance in Figure 6.2. In this case, we have three resources (machines) and six tasks. Each task in the figure is vertically sized based on the amount of time required to execute the task (we assume all three machines are equal in capabilities for this example). In the solution provided, machine three defines the makespan, as it contains the lengthiest amount of tasks. Of course, this solution is sub-optimal, as we could move task six to machine two in order to generate an improved solution. In this improved solution, machine three still defines the makespan, but the actual value will be smaller, as only task four needs processing, rather than four and six.

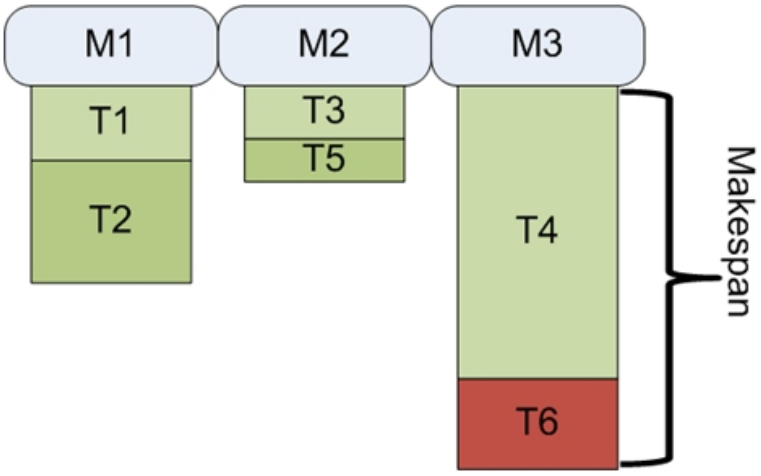

Figure 6.2: Example of task matching and makespan determination.

While a toy problem such as the one in Figure 6.2 may not seem particularly intensive, the task matching problem becomes very computationally intensive as the problem size scales upwards. With many machines, and even more tasks, the possible combinations of task to machine matchings becomes extraordinarily high. Rather than a brute force approach, we need more intelligent algorithms to solve this problem within a reasonable amount of time.

While we will discuss the PSO-related solutions to this problem in Section 6.3.3, we will conclude this section with a brief look at some of the simpler heuristic algorithms developed for solving the task matching problem. The simplest of these is the first-come first-serve (FCFS) algorithm, that simply matches a task to the most optimal machine as they are observed by the algorithm. The algorithm will choose a best machine based on the current Machine Available Time (MAT) of each machine. The MAT of a machine is the amount of time required to complete all tasks currently matched to that machine. The algorithm assigns the task to the machine with the

lowest MAT.

Two more heuristics for solving the task-matching problem are the min-max and min-min [10] algorithms. The min-min algorithm first determines the minimum completion time for each task that we want to consider across each machine. Within these minimum completion time values it searches for the *minimum* and assigns that task to the corresponding machine. The min-max algorithm handles this problem in a slightly opposite manner. The algorithm still computes the minimum completion times, but rather than assigning the task with the minimum value to the corresponding machine, it assigns the task with the maximum value.

The issue with these two algorithms is that they are suited for very particular instances of the problem. Min-min, for example, works well with many small tasks, as it will assign them to their optimal machines first, leaving the few longer tasks for last. This algorithm may improve the makespan for problems with many small tasks, but as the number of larger tasks increases the results worsen. Min-max, on the other hand, performs better when there are many longer tasks in the problem instance. Neither of these problems provide optimal solutions across all potential cases. PSO, on the other hand, searches for optimal solutions through the solution space, and may work effectively regardless of the task composition.

6.3 Related Work

As this chapter deals with a few areas that can be considered independently, we split this section of related working into a few subsections: multi-swarm PSO, PSO for the GPU, and PSO targeted at the task-matching problem. We investigate the

existing work for each of these areas independently.

6.3.1 Multi-Swarm PSO

The literature contains a number of works based around multi-swarm PSO. One such work by Liang and Suganthan [28] acts as a modification to a dynamic multi-swarm algorithm. The dynamic multi-swarm algorithm initializes a small number of particles in each swarm and then randomly moves particles between swarms after a given number of iterations. The authors augment this algorithm by including a local refining step. This step occurs every given number of iterations and updates the local best of a particle only if it is within some threshold value relative to the other particles in the swarm.

A different work by van den Bergh and Engelbrecht [52] considers having each swarm optimize only one of the problem's dimensions, and provide a number of "collaborative" PSO variants. The authors showed that their algorithm provides better solutions as the number of dimensions increases. Compared to a genetic algorithm, van den Bergh and Engelbrecht experimentally show that their collaborative PSO algorithms consistently perform better in terms of solution quality. They further compare their algorithms against a standard PSO algorithm and show that the collaborative PSO algorithms beat this standard algorithm four times out of five. The authors mention, however, that if multiple dimensions are correlated, they should be packed within a single swarm.

As was previously mentioned, we follow the work described by Vanneschi et al. [53] for our implementation on the GPU. Their "MPSO" algorithm solves an optimiza-

tion problem via multiple swarms that communicate by moving particles amongst the swarms. Every given number of iterations swarms will move some of their best particles to a neighboring swarm, replacing some of the worst particles in that swarm. They describe a further addition to this algorithm, "MRPSO", that further uses a repulsive factor on each particle. Their results show that both MPSO and MRPSO typically outperform the standard PSO algorithm, with MRPSO providing improved performance over MPSO.

6.3.2 PSO on the GPU

To the best of our knowledge, there does not exist any collaborative, multi-swarm PSO implementations on the GPU in the literature. Veronese and Krohling [4] describe a simple implementation of PSO on the GPU. Their implementations use a single swarm and split the major portions of PSO into separate kernels, with one thread managing each particle. In order to generate random numbers Veronese and Krohling [4] use a GPU implementation of the Mersenne Twister pseudo-random number generator. When executed against benchmark problems the authors show up to an approximately 23 times speedup compared to a sequential C implementation when using a large number of particles (1000).

Similar to the work of Veronese and Krohling, Zhou and Tan [59] also describe a single-swarm PSO algorithm for the GPU. The authors, too, assign one thread to manage one particle, and split up the major phases of PSO into individual GPU kernels. For random number generation, however, Zhou and Tan [59] differ from Veronese and Krohling [4] in that they use the CPU to generate pseudo-random

numbers and transfer these to the GPU. The authors achieve up to an 11 times speedup compared to a sequential CPU implementation. They take care to note, however, that they used a mid-range GPU for their tests, and they expect the results to be improved further on more powerful GPU hardware.

Mussi et al. [31] investigate the use of PSO on the GPU for solving a real-world problem: road-sign detection. When updating the position and velocity of the particle, the authors map threads to individual elements/dimension values and not a particle as a whole. Similarly, multiple threads within a block collaborate to compute the fitness value of each particle during the fitness update phase. Mussi et al. show that their GPU implementation achieves around a 20 times speedup compared to a sequential CPU implementation.

Mussi et al. [32] provide another, more recent GPU implementation of PSO. As with their earlier work in [31] the authors assign a single thread to a single dimension for each particle. Mussi et al. [32] test their algorithm against benchmarking problems with up to 120 dimensions, and show that the parallel GPU algorithm outperforms a sequential application. Finally, the authors mention in passing the ability to run multiple swarms, but do not elaborate or test such situations.

The general theme across the works we have covered has been parallelizing single swarm PSO (with, perhaps, a brief mention of multi-swarm PSO, but no actual descriptions of the work). In the most recent case, Mussi et al. [32] provided a fine-grained implementation of PSO that attempts to take advantage of the massive threading capabilities of the GPU. The authors, however, only run test sizes up to 120 dimensions and 32 particles. For our work, we wished to test across not only a

larger number of dimensions, but a large number of particles as well. As a result, we use a mixed strategy that does not lock a static responsibility to a thread, and, further, provides support for multiple swarms that collaborate with one another.

6.3.3 PSO for Task Matching

Applying PSO to the task matching/mapping problem has been studied in the past by various groups. These previous works have investigated both the continuous and discrete methods of PSO. All of the works we discuss here have one main idea in common: they work in an n dimension solution space, where n is equal to the number of tasks. One dimension maps to one task, and a location along a dimension (typically, but not always) represents the machine that the task is matched to.

To start, Zhang et al. [58] apply the continuous PSO algorithm to the task mapping problem. In their implementation, the authors use the Smallest Position Value (SPV) technique (described by Tasgetiren et al. [51]) in order to generate a position permutation from the location of the particles. Hence, the solution by Zhang et al. does not directly map a location in a dimension to a matching matching, but rather uses the locations to generate some permutation of matchings. Zhang et al. benchmark their algorithm against a genetic algorithm and show that PSO provides superior performance.

A recent work by Sadasivam and Rajendran [40] also considers the continuous PSO algorithm coupled with the SPV technique. The authors focus their efforts on providing load balancing between grid resources (machines), thus adding another layer of complexity into the problem. Unfortunately, the authors only compare their

PSO algorithm to a randomized algorithm, and show that PSO provides superior solution quality.

Moving away from continuous PSO, Kang et al. [24] experimented with the use of discrete PSO for matching tasks to machines in a grid computing environment. They compared the results of their discrete PSO implementation to continuous PSO, the min-min algorithm, as well as a genetic algorithm. The authors show that discrete PSO outperforms all of the alternatives in all test cases. Shortly thereafter, Yan-Ping et al. [57] described a similar discrete PSO solution with favorable results compared to the max-min algorithm. Both sets of authors, however, test with very small problem sizes — equal to or below 100 tasks.

Our work described in this thesis follows our previous work from Solomon et al. [49].

6.4 Multi-Swarm PSO on the GPU

To lead into our description of the GPU implementation, we will first discuss the main concepts of multi-swarm PSO for task matching without consideration of the GPU architecture. From this groundwork we can then move on to discuss the specifics of the GPU version itself.

We define an instance of the task matching problem as being composed of two components:

1. The set of tasks, T, to be mapped, and,

2. The set of machines, M, which tasks can be mapped to.

We define a task by the number of instructions it contains, or its length. In the same simple manner, we define a machine by nothing more than its MIPS (Millions of Instructions Per Second) rating. We define the problem *size* of our problem instance using two components as well:

1. The total number of tasks, $|T|$, and,

2. The total number of machines, $|M|$.

A solution for the task matching problem consists of a vector, $V = (t_1, t_2, ..., t_{|T|})$ where the value of t_i defines the machine that task i is assigned to.

We use the elements of V to compute the makespan of the solution. The makespan for v is equal to the maximum MAT of the machines in the solution. From this, we know that we need to compute the execution time of each task $t \in T$ on the machine it has been assigned to in V, in order to build up the MAT values for each machine. Our final goal, of course, is to find a V that provides as minimal a makespan as possible.

As we know, PSO works by using a number of particles, all with their own local solutions (V), that move around the solution space for a number of iterations (where each particle ideally finds a new solution every iteration). As a result, we expect to be computing the makespan many times over the course of the algorithm — which further requires constantly computing the execution time of each task on the machine it has been matched to. We use an Estimated Time to Complete (ETC) matrix to store lookup data on the execution time of tasks for each machine in order to remove redundant computations. An entry in the ETC matrix at row i, column j defines the amount of time machine i requires to execute task j, given no load on

the machine. While the ETC matrix is not a necessity, the reduction in redundant computations during the execution of the PSO algorithm makes up for the (relatively small) additional memory footprint.

Similar to the work described in Section 6.3.3, we map one dimension in the solution space to one task in the problem instance. The solution space for a given instance contains exactly $|T|$ dimensions. As any task may be assigned to any machine in a given solution, each dimension must have coordinates from 0 to $|M| - 1$.

At this point, we must deviate slightly from the standard continuous PSO representation of the solution space. Typically, a particle moving along dimension x moves along a continuous domain: any possible point along that dimension represents a solution along that dimension. Clearly, this is not the case for task mapping as a task cannot be mapped to machine 4.32427, but, rather, must be mapped to machine 4 or 5. Unlike Kang et al. [24] or Yan-Ping et al. [57], we do not move to a modified discrete PSO algorithm, but maintain the use of the continuous domain in the solution space.

We ran a few brief tests of simple, single-swarm implementations of continuous versus discrete PSO for task matching and found that a continuous domain provides improved results, as shown in Figure 6.3. However, unlike Zhang et al. [58] or Sadasivam and Rajendran [40] we do not introduce an added layer of permutation to the position value by using the SPV technique. Rather, we use the much simpler technique of rounding the continuous value to a discrete integer. We chose this simplified technique as the focus of our work rests on the GPU performance of multi-swarm PSO for task matching, not specialized techniques branching off of the main path.

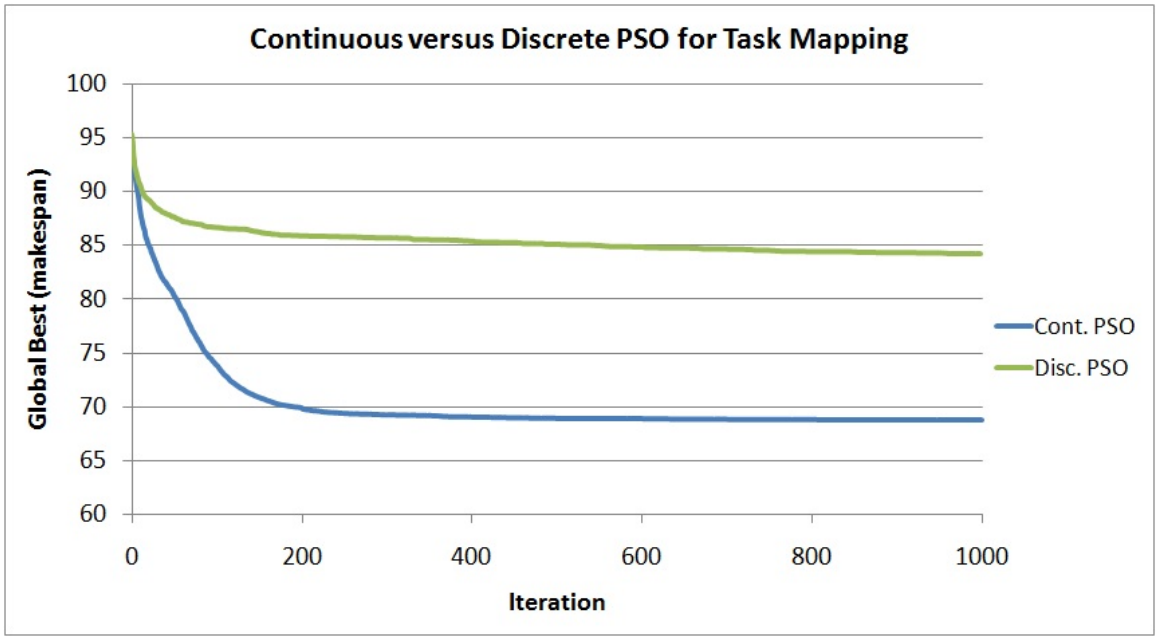

Figure 6.3: Global best results for continuous and discrete PSO by iteration.

6.4.1 Organization of Data on the GPU

We begin the description of our GPU implementation with a discussion on data organization. For our GPU PSO algorithm, we store all persistent data in global memory. This includes the position, velocity, fitness, and current local best value/position for each particle, as well as the global best value/position for each swarm. As we require random numbers at each iteration, we also store a set of pre-generated psuedo-random numbers in global memory. We use a separate one-dimension array as the storage container for each of these data sets.

For position, velocity, and particle/swarm-best positions, we store the dimensional values for the particles of a given swarm in a special ordering. Rather than group each dimension for one particle and then moving on to the next, we group the values up

by dimension. Figure 6.4 provides an example of how this data is stored (swarm-best positions are stored per swarm, rather than per particle, however). In a given swarm, we store all of dimension 0's values for each particle, followed by all of dimension 1's values, and so on. While we explain this choice further in Section 6.4.2, we immediately note that this ensures the GPU coalesces all data accesses to these memory locations due to how we structure the thread responsibilities within our kernels. We store per-particle fitness values as well as particle-best and swarm-best values in a linear manner with only one value per particle (or swarm, in the case of the swarm-best values). We use this simpler structure for this particular data as threads will access it in a coalesced fashion without requiring any extra work. Figure 6.5 shows this organization.

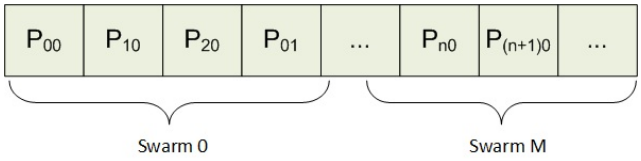

Figure 6.4: Global memory layout of position, velocity, and particle best positions (P_{xy} refers to particle x's value along dimension y).

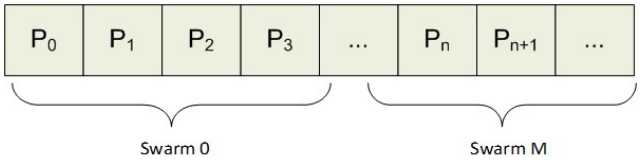

Figure 6.5: Global memory layout of fitness and particle best values.

During the calculation of a particle's fitness value, all threads require access to the ETC matrix. To compute the makespan, each particle must first add to the execution time of tasks assigned to each machine. This is, of course, handled by observing the

particle's position along each dimension. As dimensions map to tasks, we are looping through each of the *tasks* and determining which *machine* this particular solution is matching them to. As there are likely to be many more tasks than machines in the problem instance, there will likely be many duplicate reads to the ETC matrix by various threads. As a result, we place the ETC matrix into texture memory. As we discussed in Section 2.2, texture memory offers a cache at the SM level. As it is extremely likely that there will be multiple reads to the same location in the ETC matrix by different threads, we believe using the texture cache will result in a reasonable level of performance gains as, in the worst case, we require accesses to global memory, but in the best case, we access the much speedier cache.

6.4.2 GPU Algorithm

With the organization of data on the GPU set, we move on to discuss the GPU algorithm itself, the issues we ran in to, and the resolutions we came up with. The first issue surrounding our GPU algorithm is random number generation. As we know from Equation 6.1, PSO requires random numbers for each iteration. In order to generate the large quantity of random numbers required, we make use of the CURAND library included in the CUDA Toolkit 3.2 [33] to generate high quality, pseudo-random numbers on the GPU. We generate a large amount of random numbers at a time (250MB worth) and then generate more numbers in chunks of 250MB or less when these have been used up.

Our implementation of multi-swarm PSO is split up into a series of kernels that map to the various phases of the algorithm. These phases and kernels are as follows:

Particle Initialization

This phase initializes all of the particles by randomly assigning them a position and a velocity in the solution space. As each dimension of each particle can be initialized independently of one another, we assign multiple threads to each particle: one per dimension. All of the memory writes are performed in a coalesced fashion, as all threads write to memory locations in an ordered fashion. As this kernel is simple and straight-forward we do not list the pseudocode.

Update Position and Velocity

This phase updates the velocity of all particles using Equation 6.1 and then moves the particles based on this velocity. As was the case with particle initialization, we can handle each dimension independently. As a result, we again assign a single thread to handle each dimension of every particle. In the kernel, each thread updates the velocity of the particle's dimension it is responsible for, and then immediately updates the position as well. We note that when updating the velocity using Equations 6.1 and 6.3 all threads covering particles in the same swarm will each access the same element from the swarm-best position in global memory. While this may seemingly result in uncoalesced reads, global memory provides broadcast functionality in this situation [33], allowing this read value to be broadcast to all threads in the half-warp using only a single transaction. While simple, we provide the pseudocode of the GPU kernel in Algorithm 10.

Algorithm 10 Update Position and Velocity Kernel

Require: NumSwarms, NumParticles, NumTasks, NumMachines, Velocity[], Position[], lBestPos[], pBestPos[], Rands[]

1: tid = blockIdx.x ∗ blockDim.x + threadIdx.x
2: **if** tid ≤ NumSwarms ∗ NumParticles ∗ NumTasks **then**
3: Update velocity along one dimension for one particle following Equation 6.3
4: Write velocity to global memory (Velocity[tid])
5: Position[tid] ← Position[tid] + Velocity[tid]
6: **end if**

Update Fitness

In this phase, we update the fitness values for all particles after they have moved to their new positions. Not only is this phase more complex than the previous phases, but it encapsulates the irregular computation aspects of this algorithm. As a result, we cannot exploit parallelism to the same degree as the previous phases. Rather than map a thread to a single dimension of a particle, we map a single thread to each particle. Our reasoning behind this choice of thread-particle mapping is due to how makespan is computed. This computation involves first determining the MAT for each machine, and then taking the maximum value as the makespan.

These computations are irregular in nature due to the unpredictable memory accesses required to compute the MAT of a machine. When computing the MAT we must read from the ETC matrix at a location based on the task-machine matching. We do not know ahead of time which tasks will be assigned to which machines. As a result, we cannot guarantee any structure in the memory requests, and we cannot

ensure coalescing.

One option for parallelization involves having a thread compute the makespan for a single machine and then perform a parallel reduction to find the makespan for each particle. The issue with this approach, however, is that a particle's position vector is ordered by task, not by machine. We do not and cannot know which tasks are assigned to which machine ahead of time. If we parallelized this phase at the MAT computation level then all threads would have to iterate through all of the dimensions of a particle's position anyways, in order to find the tasks matched to the machine the thread is responsible for. As a result, we choose to take the coarser-grained approach and have each particle compute the makespan for a given particle.

We implement two different kernels in order to accomplish this coarser-grained approach. With the first approach, we use shared memory as a scratch space for computing the MAT for each machine. Each thread requires $|M|$ floating-point elements of shared memory. Due to the small size of shared memory, however, larger values of $|M|$ (dependent on the number of particles in a swarm) require an amount of shared memory exceeding the capabilities of the GPU. To solve this, we develop a second, less optimal kernel, where we use global memory for scratch space. Given only one thread block executing per SM, the first kernel can support 128 threads (particles) with a machine count of 30, whereas the second kernel supports any value beyond that.

The second, global memory kernel shows our reasoning for choosing the ordering of position elements in global memory (Figure 6.4). When computing the makespan, each thread reads the position for its particle in the current dimension being consid-

ered in order to discover the task-machine matching. All the threads within a thread block work in lockstep with one another, and, thus, work on the same dimension at the same time. The threads within a thread block, therefore, read from a contiguous area in global memory and exploit coalescing. This coalescing results in an approximate 200% performance improvement over an uncoalesced version based on our brief performance tests.

Algorithm 11 Shared Memory Fitness Update Kernel

Require: TotalParticles, NumTasks, NumMachines, Matching[]

```
1: sScratch[] //Shared memory location
2: mySwarm ← blockIdx.x ∗ TotalParticles ∗ NumTasks
3: myScratchOffset ← threadIdx.x ∗ NumMachines
4: makespan ← 0
5: Each thread clears their scratch table elements
6: for i ← 0 to NumMachines do
7:    matchingVal ← Matching[mySwarm +(i∗TotalParticles)+threadIdx.x]
8:    etcVal ← ETC Matrix texture lookup
9:    sScratch[myScratchOffset + ⌊matchingVal⌋] += etcVal
10:   //Keep a running total of the current highest makespan
11:   if etcVal > makespan then
12:      makespan ← etcVal
13:   end if
14: end for
15: Each thread writes makespan value out to global memory
```

We provide the pseudocode of our shared memory kernel in Algorithm 11. Note that we keep a running total of the makespan for each particle's solution within the for loop (lines 11–13). This avoids requiring another for loop after the kernel has computed the MAT values which finds the maximum MAT. The global memory kernel, not provided as pseudocode, is exactly the same — only `sScratch` changes, as we use global memory, rather than shared memory, as the scratch space for each thread.

Update Best Values

This phase updates both the particle best and global best values. We use a single kernel on the GPU and assign a single thread to each particle, as we did with the fitness updating. As was the case with previous phases, we assign all threads covering particles in the same swarm to the same thread block. The first step of this kernel involves each thread determining if it must replace its particle's local best position, by comparing its current fitness value with its local best value. If the current fitness value is greater, then the thread replaces its local best value/position with its current fitness value and position.

In the second step, the threads in a block collaborate to find the minimal local best value out of all particles in the swarm using a parallel reduction. If the minimal value is better than the global best, the threads replace the global best position. Threads work together and update as close to an equal number of dimensions as possible. This allows us to have multiple threads updating the global best position, rather than relying on only a single thread to accomplish this task. Similar to the

initialization phase, this kernel is very straightforward and, as a result, we do not provide the pseudocode.

Swap Particles

Our final phase, the swap particles phase, replaces the n worst particles in a swarm with the n best particles of its neighboring swarm. Following the work of Vanneschi et al. [53] we set the swarms up as a simple ring topology in order to determine the direction of swaps. We use two kernels for this phase. The first kernel determines the n best and worst particles in each swarm. For this kernel, we again launch one thread per particle, with thread blocks composed only of threads covering particles in the same swarm. In order to determine the n best and worst particles, we iterate n parallel reductions, one after the other. Each parallel reduction determines the nth best/worst particle in the swarm covered by that thread block. We improve the performance of this lengthy kernel by reading from global memory only once: at the beginning of a kernel each thread reads in the fitness value of its particle into a shared memory buffer. This buffer is then copied into two secondary buffers which are used in the parallel reduction (one for managing best values, one for worst).

Once a reduction has been completed, we record the index of the located particles into another shared memory buffer. We then restart the reduction for finding the $n + 1th$ particle by invalidating the best/worst particle from the original shared memory buffer, and recopying this slightly modified data into the two reduction shared memory buffers. This process continues until all best/worst particles have been found. At this point, n threads per block write out the best/worst particle indices to global memory

in a coalesced fashion.

The second kernel handles the actual movement of particles between swarms. This step involves replacing the position, velocity, and local best values/position of any particle identified for swapping by the first kernel. For this kernel we launch one thread per dimension per particle to be swapped.

As the second kernel is very simple, we provide only the pseudocode for the first kernel in Algorithm 12. Note that the parallel reductions for finding the best and worst particle are performed at the same time (in order to reduce the number of loops required).

6.5 Results

We start with an investigation into the effects that increasing the swarm, task, and machine counts individually have on the execution time of our algorithm. Figure 6.6 shows the results for variable swarm counts. In these tests we used 80 tasks and 8 machines. We used such a low number of tasks to ensure that our algorithm uses the shared memory fitness kernel. We do this in order to reduce the effects that tasks and machines have on the run time in order to isolate the effect of the swarm count as much as possible. As expected, the GPU implementation outperforms the sequential CPU implementation to a very high degree. With the swarm count set at 60 the GPU algorithm achieves an approximate 32 times speedup over the sequential algorithm.

We investigated these results further by observing how the individual execution times of the top four kernels change as the swarm count increases (we do not include the other kernels, as their total execution times are a fraction of any of the pictured

Algorithm 12 Swap Particles Kernel

Require: NumSwarms, NumParticles, NumToSwap, Fitness[], BestSwapIndices[], WorstSwapIndices[]

```
1: tid = blockIdx.x * blockDim.x + threadIdx.x
2: if tid ≤ NumParticles then
3:    Load our fitness data into shared memory
4: end if
5: //Perform the parallel reductions, each one finding a best/worst value.
6: for 0 to NumSwap do
7:    for i ← blockDim.X /2 to 0, step >>= 1 do
8:       if tid < i then
9:          Perform one parallel reduction step for finding best value.
10:         Perform one parallel reduction step for finding worst value.
11:      end if
12:   end for
13:   Invalidate fitness values in shared memory
14:   (Those corresponding to the newest best/worst particle found)
15:   __syncthreads()
16: end for
```

kernels). We show these results in Figure 6.7. The update position and velocity kernel forms the greatest contributor to the increase in the GPU's execution time as the swarm count increases. We explain this by revisiting the overall responsibilities of this kernel. That is, the update position and velocity kernel requires a large number of

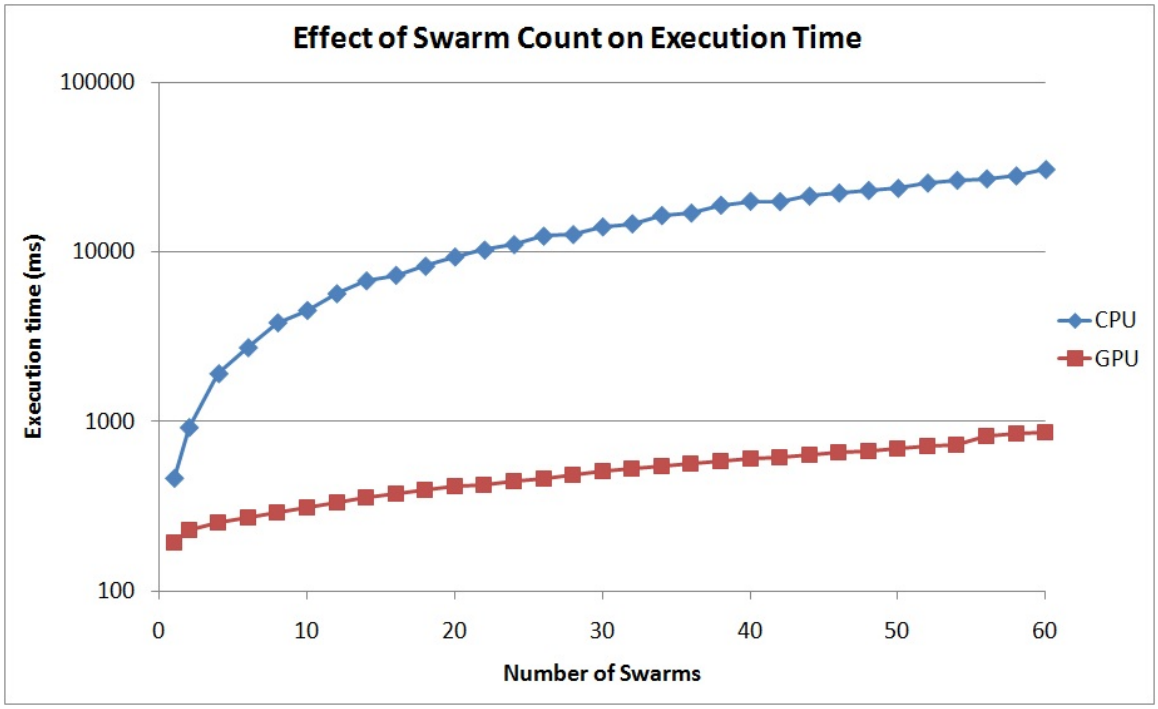

Figure 6.6: Comparison between sequential CPU and GPU algorithm as swarm count increases.

global memory reads and writes (to read in the many-dimensioned position, velocity, and current bests position data), and is relatively computationally intensive when determining the new velocity. When combined with the fact that we are launching a thread per dimension per particle the GPU's resources quickly become saturated, hence the linear increase in execution time.

We explain the "jump" present in the last three data points of the fitness kernel as being due to oversaturation of the GPU's resources. The GTX 260 GPU has 27 SMs available. With, for example, 56 swarms, we have 56 thread blocks assigned to the fitness kernel. With the resources required by the configuration tested, each SM can support only 2 thread blocks simultaneously. Hence, the GPU executes 54 thread blocks simultaneously, leaving 2 thread blocks waiting for execution. These 2

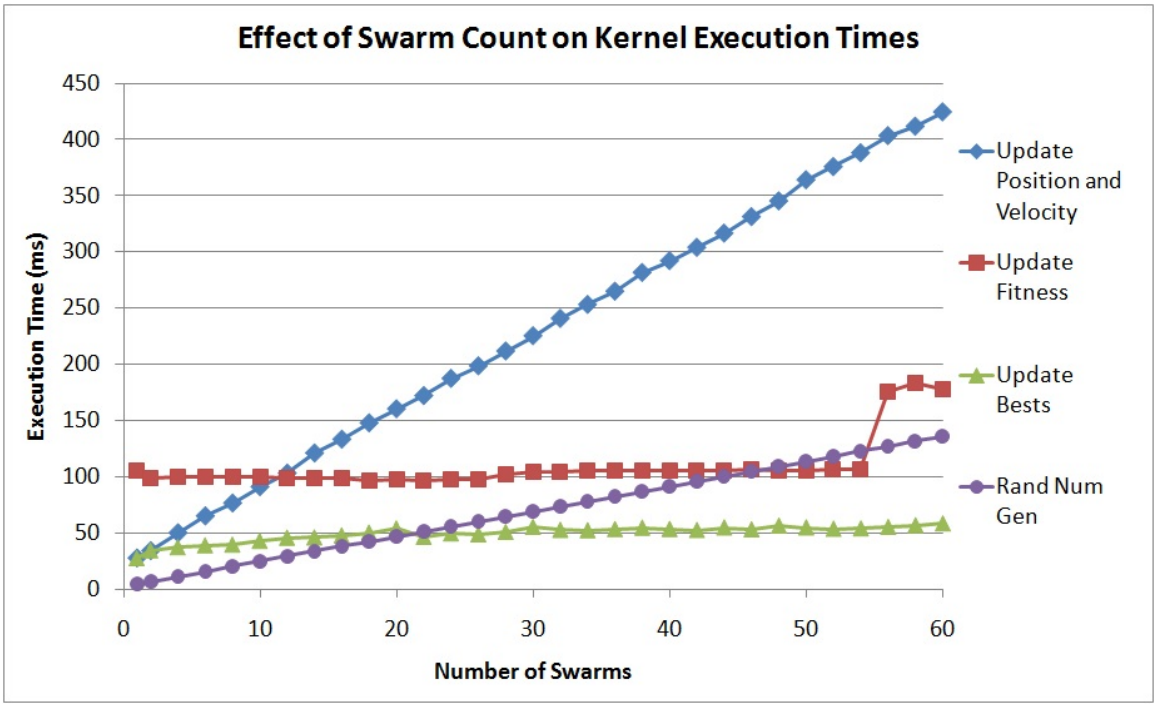

Figure 6.7: Total execution time for the various GPU kernels as the swarm count increases.

thread blocks must then wait until resources are freed on an SM by another thread block completing execution, causing a significant increase in the execution time of the kernel.

Another interesting observation with the fitness kernel is that despite the swarm count increasing, the execution time remains relatively static until a swarm count of 56. This is, again, caused by the SM saturation point being reached with a swarm count of 56. Prior to that swarm count, all threads blocks may execute immediately on an SM — the GPU has enough resources to allow parallel execution of all threads required by the kernel. This ensures that we do not observe any significant performance loss as the swarm count increases until we run out of available SMs for thread blocks at a count of 56.

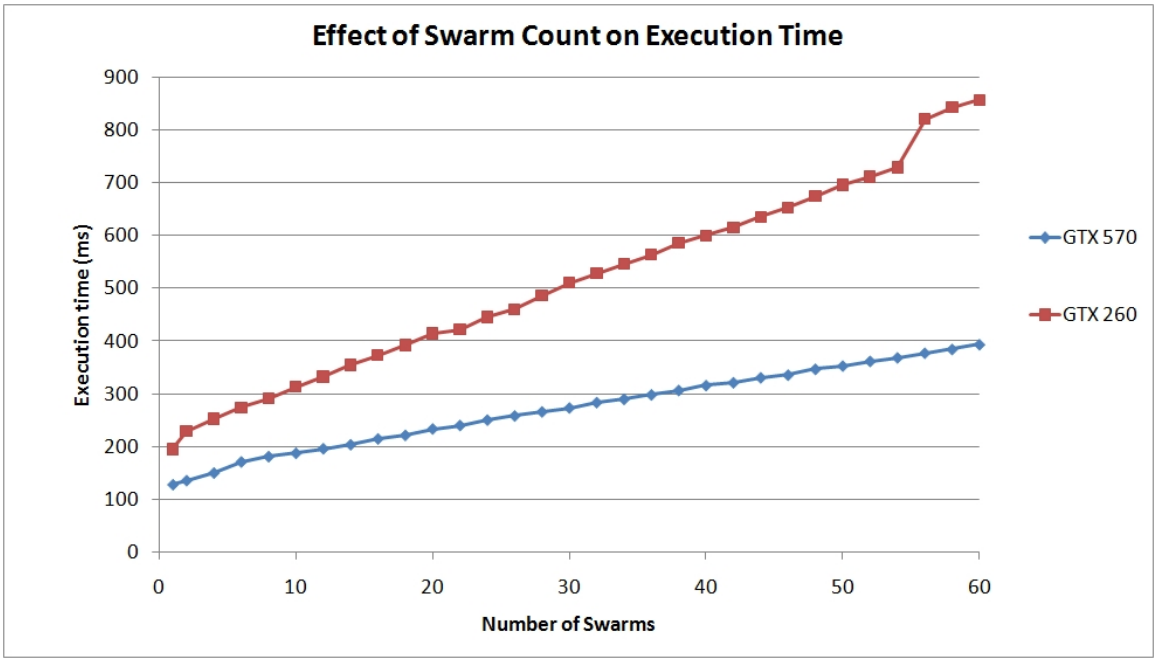

Figure 6.8: Comparison between GTX 260 and 570 GPUs as swarm count increases.

To help prove this, we take our experiments a step further and observe the performance of the algorithm on a GTX 570. While the GTX 570 technically contains less SMs than the GTX 260, each SM contains more SPs and allows for more threads and/or thread blocks to execute on each simultaneously. As a result, we expect the jump in execution time with swarm counts greater than or equal to 56 to be absent in the GTX 570 tests. As we show in Figure 6.8, our expectation matched our experimentation. We see that the GTX 570 performance line does not contain the same jump in the last three data points (at swarm count sizes of 56, 58, and 60). The superior overall performance of the GTX 570 is expected and is simply due to the nature of testing on newer, more powerful hardware with more available hardware parallelism compared to the GTX 260.

Before concluding with the swarm count tests we also investigated our use of

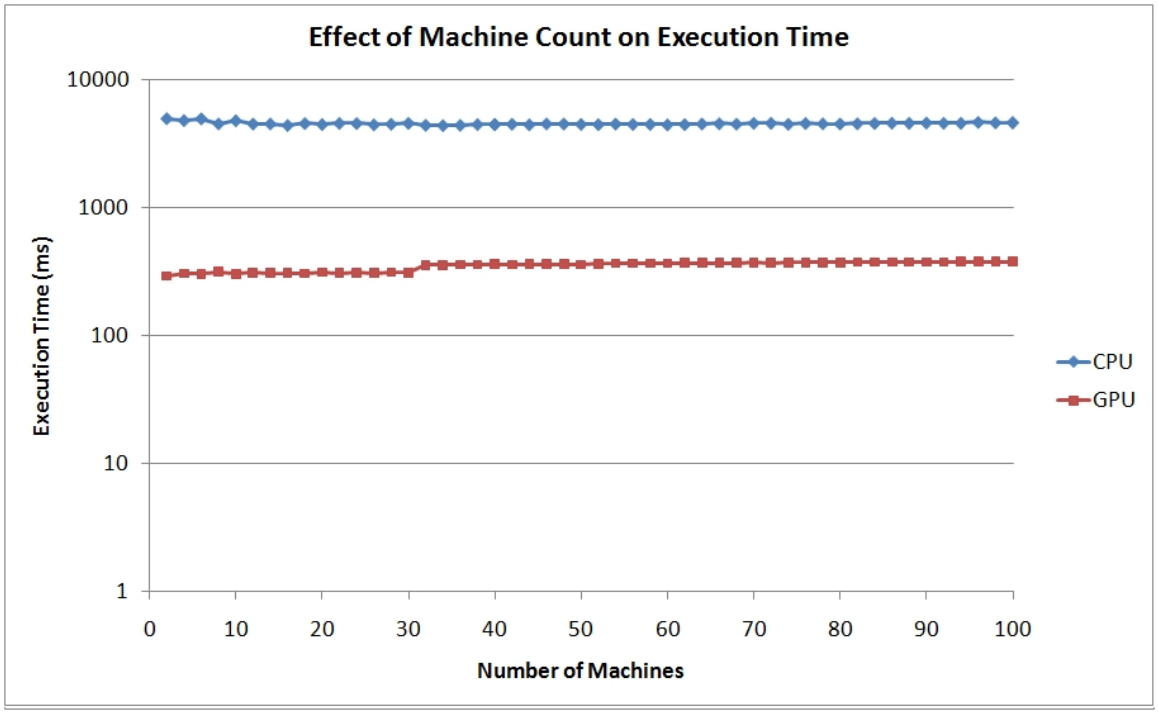

Figure 6.9: Comparison between sequential CPU and GPU algorithm as machine count increases.

texture memory in the fitness kernel. We profiled a few runs of the algorithm using the CUDA profiling tool which allowed us to measure the cache hits. The results from the analysis showed that our hypothesis that texture memory would help the fitness kernel's performance was correct. The profiler reported ETC Matrix cache hits anywhere from 88% to 97%. As a result, we dramatically reduced the number of global memory reads required to compute the makespan, and, instead, exploit the speedier texture caches of the GPU.

Moving on, we examine the performance and scaling of the algorithm when we increase machine counts as well as increase the task counts. For the machine count scaling we keep the task count and the number of swarms static at 80 and 10 respectively. As the machine count increases, we observe the effect that switching to

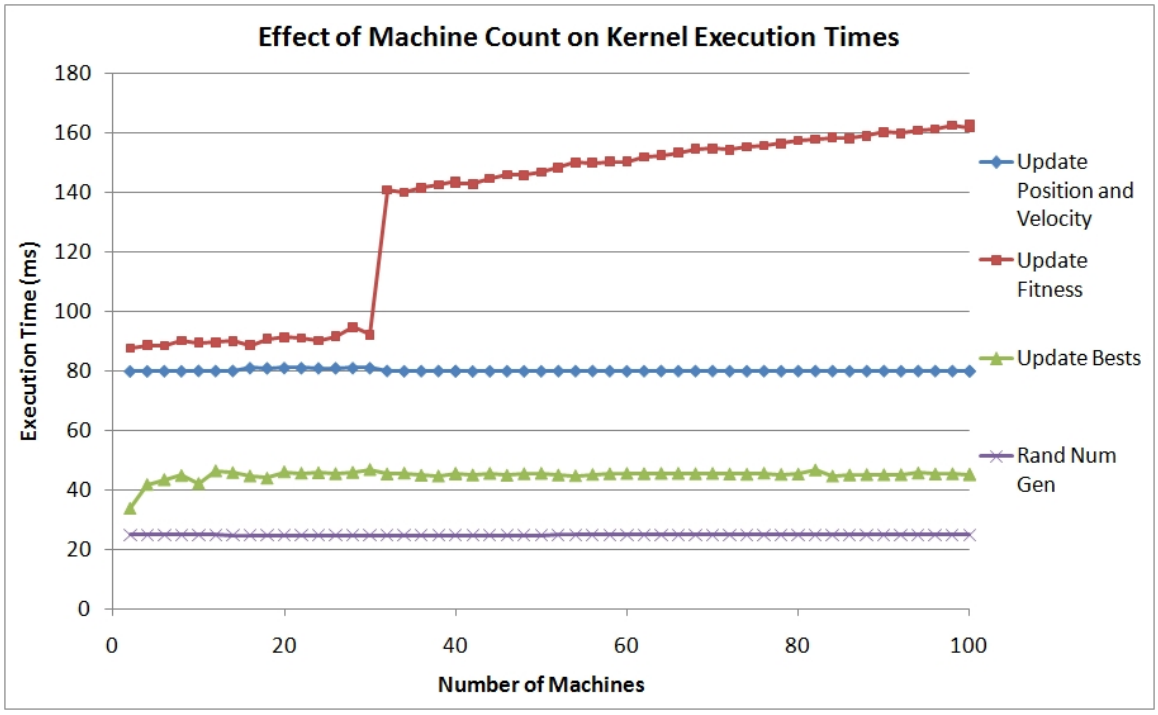

Figure 6.10: Total execution time for the various GPU kernels as the machine count increases.

the global memory fitness kernel has on the execution time. Figure 6.9 shows the results with machine counts from 2 to 100. Unlike the swarm count tests, we see that the execution time does not change dramatically as the machine count increases. However, the GPU execution time still increases by 14% when the machine count increases from 30 to 32. This occurs due to the shift from shared memory to global memory use for the fitness kernel, which, in turn, results in a 50% increase in the total execution time for this kernel.

Figure 6.10 provides the execution time results of some of the kernels as the machine count increases. We immediately observe that all of the kernels, with the exception of the fitness update kernel, exhibit roughly static execution times. We expect these results, as increasing the machine count does not result in any compu-

tational or memory access increases for these kernels (the same effect we saw when considering the performance of the algorithm itself with varying machine counts). We do, however, see a substantial increase in the execution time of the fitness kernel after 30 machines. As we know, this is the point where the algorithm shifts from using the shared memory fitness kernel to the global memory kernel. These results help us to observe the significant improvements in performance achieved by using shared memory over global memory.

Moving on to the task count scaling tests, we keep the machine count and number of swarms static at 8 and 10 respectively. We provide the results for task count scaling in Figure 6.11. The results are very similar to those of the swarm count tests in that the GPU algorithm significantly outperforms the sequential CPU algorithm, and the execution time increases as the task count increases. Overall, however, the GPU cannot provide the same level of speedup while the swarm count remains low, despite task sizes increasing. We expect this, as increasing the number of swarms increases the exploitable parallelism at a faster rate than the task count. We do not provide a separate graph of the various kernel execution times as they are very similar to those in Figure 6.7, in that the update position and velocity kernel dominates the run time again as the algorithm uses the shared memory fitness kernel throughout.

Finally, we ran two tests using a large number of tasks and swarms, with one using the shared memory kernel (10 machines) and the other using the global memory kernel (100 machines) in order to gauge the overall performance of the algorithm as well as come up with the overall percentage of execution time each kernel uses. We provide the results in Table 6.1 (the percentages for the initialization and swapping kernels

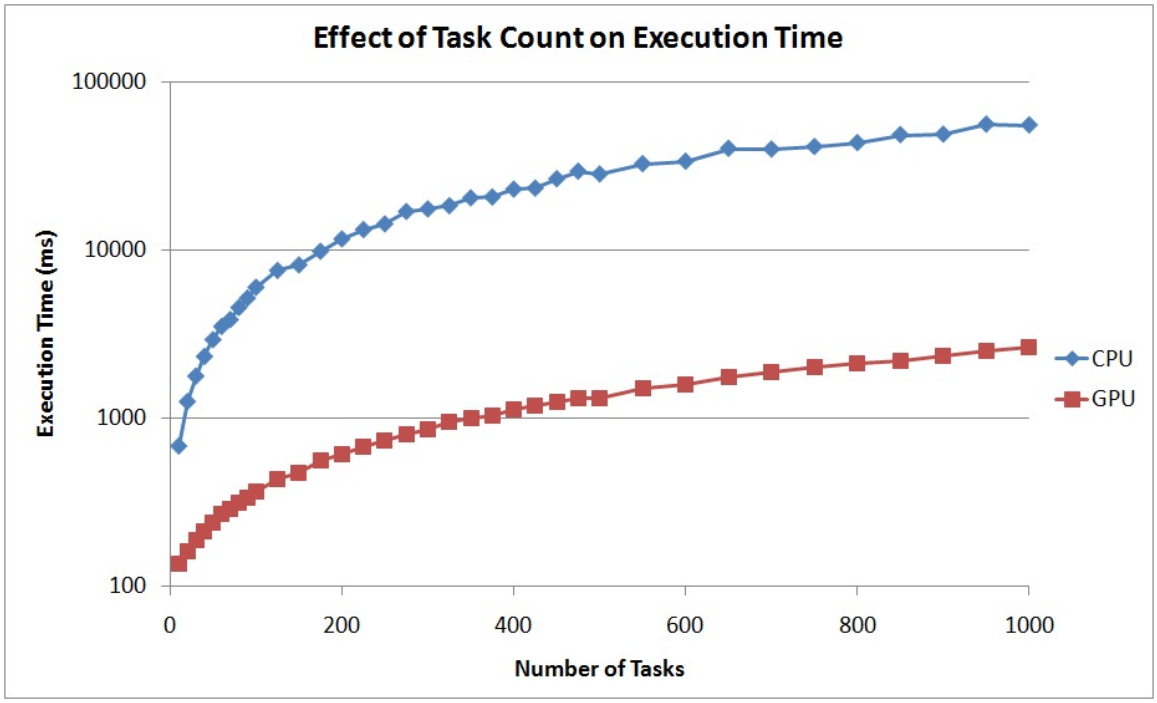

Figure 6.11: Comparison between sequential CPU and GPU algorithm as task count increases.

are not included in the figure as, combined, they contribute less than 1% to the overall execution time). We see that the update position and velocity kernel dominates the run time of the shared memory instance, whereas the global memory instance sees the fitness kernel moving to become the top contributor to the overall execution time. We expect this behavior, as the only change in the global memory instance resides within the fitness update kernel. As expected, the shared memory instance sees an improved speedup (compared to the sequential CPU algorithm) of 37 compared to the global memory instance's speedup of 23.5.

Kernel	Shared Memory Fitness	Global Memory Fitness
Update Pos/Vel	54%	34%
Update Fitness	21%	50%
Update Bests	6%	4%
Rand Gen	19%	12%

Table 6.1: Percentage of execution time taken by most significant kernels

6.5.1 A Short Discussion on Solution Quality

Similar to the option pricing problem, PSO for task-matching involves an output answer that cannot be simply labeled as correct or incorrect. While the BFS and MPM algorithms either provided a correct solution or not, PSO, as a heuristic, attempts to locate as optimal a solution as possible. In order to test out the solution quality, we benchmark the answer we get from our GPU PSO algorithm with some alternative algorithms. We compare the solution quality against a sequential single-swarm PSO implementation and a First-Come-First-Serve (FCFS) algorithm that serially assigns tasks to the machine with the lowest MAT value at the time (in this case, the MAT value includes the time to complete the task in question).

For the solution quality tests we use 10 swarms with 128 particles per swarm. c_1 is set to 2.0, c_2 to 1.4, and w to 0.9. We also introduce a *wDecay* parameter which reduces w each iteration, we set this value to 0.995, and run $1,000$ iterations of PSO for each problem. Finally, we randomly generate 10 task and machine configurations for each problem size considered, and run PSO against each of these data sets. Each data set is run 100 times, and the averaged results are taken over the $100 * 10$ runs for each task/machine count configuration.

Table 6.2 provides the averaged results of the solution quality experiments, nor-

Num Tasks	Num Machines	MSPSO	PSO
60	10	0.906	0.925
60	15	0.935	0.921
70	10	0.939	0.923
70	15	0.941	0.933
80	10	0.964	0.934
200	40	1.322	1.312
1000	100	3.106	3.109

Table 6.2: Solution quality of MSPSO and PSO normalized to FCFS solution (< 1 is desired).

malized to the FCFS solution. We first tested small data sets of sizes similar to those from Sadasivam and Rajendran [40] as well as Yan-Ping et al. [57]. We can see from these that, unfortunately, MSPSO does not outperform the single-swarm PSO to any significant degree, and performs worse on many occasions. Furthermore, as the problem size increases, both variants of PSO fail to generate improved solutions when compared to FCFS. In short: we do not see a reasonable level of quality improvement from MSPSO with small problem sizes, and both variants of PSO utterly fail to provide acceptable solution quality as the problem size increases.

Our explanation for this failure to provide a reasonable level of quality in the solution rests with the nature of the solution space. Our hypothesis is that the unstructured, random-esque nature of the solution space presents an environment inimical to the intelligence of the particles. These particles attempt to use their memory and intelligence to track down optimal values in the solution space and are influenced by previously-known optimal locations. In essence, they follow some structure in the solution space and hope their exploration leads to an ideal solution. Unfortunately, with the task matching problem, we have no real structure to the

solution space. With even one task changing assignment from one machine to another the makespan may dramatically change. As a result, the intelligence of the particles cannot help us here. The end result, we believe, is that the PSO algorithm devolves into a randomized algorithm (or worse, since the intelligence of the particles reduces the overall area of the solution space explored). The added cooperation between swarms in MSPSO further provides no benefits, and perhaps even serves to cluster particles *between* swarms in all the same areas. In essence, the *exploration* aspects of PSO help us to no greater degree than a randomized algorithm, and, thus, the *exploitation* aspects are rendered useless, as exploitation of areas around local optima provide no help given the unstructured nature of the solution space. As we will discuss in Chapter 8, this is an area with definite possibilities for future work and investigation.

6.6 Summary

We have studied and described the design and implementation of a multi-swarm PSO algorithm for GPUs in this chapter. Targeting the task matching problem, we noted that the algorithm contained both regular and irregular (fitness update) properties. Unlike the irregularity of the BFS and MPM algorithms, however, we were able to more readily optimize the fitness update for the GPU as the main issue occurred with data-access patterns. Due to the relatively small memory footprint of the ETC matrix along with the expectation of redundant reads to the same elements, we were able to exploit texture memory for the storage of the ETC matrix. Unfortunately, the scratch space used for the computations of MAT values were forced into global

memory when the number of machines grew large. Despite this issue, however, we observed up to a 23.5 times speedup when using the global memory for storage of scratch space. Even more impressive was the 37 times speedup we measured when using shared memory for the scratch space.

As expected, the overall performance (in terms of speedup) of our PSO algorithms falls between the option pricing and BFS algorithms. We hypothesized this would be the case due to our PSO algorithm featuring more regularity than the BFS, but periods of irregularity that were not present in the option pricing algorithm.

In the next chapter, we link our knowledge of the four algorithms and problems we have studied and discuss what efforts were required to get each one running efficiently on the GPU. Having explored a variety of algorithms we feel that we have a greater understanding of what works well on the GPU, as well as the difference in effort required to optimize these algorithms for the GPU.

Chapter 7

Discussion and Comparison

In this chapter, we look at combining our knowledge about the four algorithms we investigated, and discuss/compare their design, implementation, optimization, and performance on the GPU hardware. We believe such a discussion provides a high level of value-add to our results, as we will observe the major differences between regular and irregular problems/algorithms on the GPU. Out of the problems we investigated we have a regular problem (option pricing), a simple irregular problem (single-source shortest path), a complex irregular problem (maximum flow), and a problem that features sections of both regularity and irregularity (PSO for task matching). As a result, we have a wealth of information regarding a variety of problems with which to make comparisons. These problems share some similar traits, such as the ability to exploit a high degree of parallelism, but differ in many ways as well. How we developed GPU-specific optimizations for the algorithms we used to solve each problem each varied quite significantly as well. Our efforts ranged from the simple (global memory coalescing) to the complex (active vertices modification), and we measured the effect

each had on the performance of our GPU implementations.

7.1 Data Structures and Memory Performance

We first consider the use of various data structures for storing the relevant data in each algorithm. As our choice of data structures and data organization were always instrinsically linked to memory performance, we include that discussion here as well. With the GPU we have efficiency concerns involving accesses to global memory that must be taken into account when developing a high performance GPU algorithm. Not only are we concerned about the frequency or amount of global memory accesses, but also the patterns of access between threads within warps. Ideally, we want each of the 16 threads within a half-warp to access data from the same block of memory in order to combine the original 16 accesses into a single access. The simplest method of accomplishing this coalescing is to ensure that each thread reads one element over in global memory from the previous thread (based on thread IDs within the warp)[1].

In terms of data itself, three of the algorithms we investigated bear strong similarities to one another. The option pricing, BFS, and MPM algorithms all use graph structures throughout their processing. As a result, we will compare these three with one another first, before moving on to discuss the PSO algorithm, which focuses on a very different data structure.

While the aforementioned three algorithms focus on graph structures, the option pricing algorithm uses a lattice/tree structure rather than the randomized graphs of

[1]In all but the edge case where some of the 16 threads accesses happen to cross over to another block in global member, the hardware will perform coalescing in full (and even in this case, coalescing will occur — we will see more accesses than the one we get with full coalescing, but the total will still be below the worst case of 16).

the BFS and MPM algorithms. In all cases, however, we use data structures that remove the inherent sparsity in the graphs and store only the relevant data. In the lookback option pricing algorithm we make use of a single one-dimensional array representing the vertices at the current layer of computation, while we use a dual-array system to describe the edges connecting vertices in the graph for the BFS and MPM algorithms. Further, thanks to the massive parallel capabilities of the GPU, both algorithms map a single thread to a single vertex and retain the ability to work on massive data sets.

In the case of the lookback option pricing, the regularized nature of the lattice allows us to not only make use of a simpler data structure, but also exploit global memory coalescing. In effect, every thread reads in structured, consecutive data from global memory, which allows for coalescing to take place. This results in a reduction of the latencies typically associated with global memory and provides significant performance improvements.

Unfortunately, due to the irregular nature of the BFS and MPM algorithms, we cannot guarantee coalescing, and we must also make use of a more complex data structure than the linear array used for option pricing. Our attempts to exploit coalescing are largely limited to the loading of edge offsets, which we handle in a coalesced manner as they are stored in a one-dimensional array and accessed in a regular fashion. However, the areas where we do exploit coalescing are but a small portion of the overall memory operations performed in each kernel.

The similarities for the data structures end here, however. With the lookback option pricing the computations themselves drive the creation of the lattice, and, as

such, the structure is known, well-defined, and, thus, very *regular*. On the other hand, the unstructured graphs for the BFS and MPM algorithms are created independently from the algorithm. From this, we see that a simple regular problem allows for full exploitation of coalescing, as well as a simplification of the data structure (a lattice devolves into a single one-dimensional array). When we make the move to an algorithm with irregular properties, whether it is simple (BFS) or complex (MPM), we require more complex data structures and can no longer take advantage of coalescing to the same degree.

Finally, we have PSO for task matching, which does not follow the same patterns as the other algorithms. We do not use any data structure beyond directly storing individual one-dimensional arrays for packing particle and swarm-related data. Unlike the previous algorithms we discussed, however, the overarching data structure itself is not the area of interest but rather the organization of data *within* that structure. We exploit coalescing by organizing the data within the arrays such that the ordering of data may no longer be the simplest, most intuitive method, but works well with the GPU hardware. The irregular nature of the fitness update computations required us to think of an alternative solution beyond coalescing. As the values used in the fitness update are typically used more than once, we exploited the texture memory of the GPU in an effort to improve the performance via caching. The only other alternative was uncoalesced reads from global memory, clearly not an ideal scenario.

We also note that we found heavy use of the higher performing memories in the GPU to be possible for only the algorithms that exhibited regular properties. For option pricing, we stored the data we knew that we would need for the current iteration

into shared memory. For PSO we not only used shared memory in this same manner, but we also used texture memory to store the ETC matrix. Of course, we were unable to exploit these memories for the purely irregular problems outside of token shared memory use when loading graph data. Overall, we see simplified data structures, and improved exploitation of memory when working with the predictability of regular problems as compared to irregular problems.

7.2 Load Balancing

When we use the term "load balancing" for the GPU we refer to the saturation of active threads within each warp and/or thread block. Ideally, we want all 32 threads within a warp to be active, effectively allowing 32 threads to execute more-or-less simultaneously. Consider the alternative case where we have some threads in a warp that are inactive. Clearly, these threads will not perform any useful computations when this warp is provided an instruction to execute. As a result, we can see a loss of thread-level parallelism in this scenario as only x_i threads, where $x_i < 32$, will be executing the instruction simultaneously for warp i, rather than 32.

For the option pricing and PSO algorithms we have this load balancing without any added effort. For each layer of the lattice that is being computed in the option pricing algorithm, we assign a single thread to a single vertex. Thus, every warp with the exception of the "last" warp is guaranteed to be saturated with active threads. Our methods for PSO are similar: either we map a thread to a particle, or one dimension of a particle based on the kernel we are currently launching. All particles are active at all times in the algorithm and, as such, we do not launch any threads

that will be inactive.

With the BFS and MPM algorithms, however, we had to explicitly develop a load balancing strategy. We accomplish this with the active vertices strategy. The end results for the layered network construction (and BFS) and push/pull phases are the same as it was with the lookback option pricing: we are guaranteed that every thread in every warp (with the exception of the "last" warp) will contain 32 threads covering active vertices. The MPM pruning phase works slightly differently however, in that we activate threads we believe will be pruned in the next iteration. As a result, we dramatically improve the number of active threads per warp, but not to the same, guaranteed extent as the other two phases.

Unfortunately, as we discussed in Section 4.3.1, this active vertices strategy requires the overhead of a scan operation in order to compute the vertex numbers of each active vertex and write them in order to a state array. Hence, we do not get load balancing "for free" as was the case in the option pricing and PSO algorithms. Due to this, we actually saw performance degradation occur for the BFS/layered network construction phase.

7.3 Hybrid Computations

For this work, we only considered the lookback option pricing as a candidate for hybrid computing. In Section 3.4 we provided the performance results between the regular GPU implementation and the hybrid implementation. As we discussed, the hybrid version does not outperform the strictly GPU implementation by anything more than a few milliseconds.

Due to these disappointing results, we do not consider the BFS or MPM algorithms as a reasonable candidate for such computations. With the lookback option pricing, we had only a small amount of data to transfer from the GPU to the CPU when the computations switched to the CPU. With the BFS and MPM algorithms we have a very large amount of data to transfer that covers the current state of vertices and edges in G. Along with this large data set we have another problem: when is an optimal time to transfer the computations back to the CPU? A solution on more traditional hardware will typically reduce the size of the graph by removing vertices and edges from the graph, thus shrinking the data set. Due to the performance issues with global memory, and the data structure we used to store the graph data, we believe it is more optimal to use state arrays to *virtually* remove vertices and edges. We now have a new difficulty associated with determining the current problem size remaining along with the inability to easily transfer the disjointed active data set. Due to the irregular nature of the computations, and the fact that we do not modify the graph structure itself in the GPU's global memory, we cannot determine how far along the computations are with any reasonable level of performance. For these reasons, we leave all BFS and MPM computations on the GPU.

The PSO algorithm, on the other hand, represents a unique scenario. For the most part, each swarm does not communicate with one another, except during the swap phase. This phase, however, does not require the transmission of a large amount of data; only the information pertaining to the best and worst particle sets must be transferred. We believe PSO may represent a possible candidate for hybrid computing where the CPU manages the execution of one swarm at the same time as the GPU

executes multiple swarms. We discuss this possibility further in Section 8.

7.4 Speedup

Our results for speedup compared to a sequential CPU implementation were exactly as we expected. We see the most significant speedup with a regular problem: lookback option pricing. When we move to another regular problem with some irregular properties (PSO for task-matching) we see a drop in the speedup, but still record it as being significantly higher than the last two, irregular problems we tested. We expect this due to what we have previously discussed in this chapter. With the regular option pricing algorithm we get most of our GPU optimizations for free, and fully exploit coalescing/load balancing. With PSO, we have the same very regular aspects across most of the algorithm, allowing for the exploitation of coalescing and load balancing, but also run into some irregular sections as well (fitness update). During the fitness update phase we cannot exploit coalescing whatsoever, and turn to the texture memory/cache for support. Further, we must move to coarser-grained parallelism in this phase, bringing the GPU that much closer to the sequential CPU implementation.

Finally, with the BFS and MPM algorithms we end up fighting for possible optimizations. The few that we find possible to accomplish (load balancing) require work of their own to use (scan operation), pushing the execution time higher. Coalescing is impossible to guarantee in almost all occasions due to the unstructured and unpredictable nature of the graph and the data access patterns of the algorithm. As a result, we see the GPU offering less of a performance improvement over the sequential

implementation when compared to the option pricing algorithm. The BFS, being a simpler irregular algorithm with only a single phase, offers improved speedup over the more complex, irregular MPM algorithm.

The comparisons we have drawn help to show the difficulty associated with parallelization of algorithms for solving irregular problems. We observed what efforts were required to optimize the algorithms, and how we accomplished these in the areas of memory performance, load balancing, data structures, and hybrid computing. In the case of memory performance especially, many of the optimizations we found possible for a regular algorithm were simply not possible to accomplish when we introduced irregularity. This leads to our immediate conclusion that some of the most important points of optimization are more-or-less free when dealing with a regular problem, while an irregular problem requires significantly more effort in order to attempt to optimize for the GPU.

7.5 A Brief Aside on Price:Performance

Before closing this chapter, we wish to briefly touch on the price-to-performance ratio offered by the GPU. We use this to explain why we compare the GPU results to a sequential CPU implementation rather than other parallel systems, as well as further support for the GPU as a parallel architecture. In essence, the GPU provides an extremely powerful parallel chip at a substantially lower cost than traditional parallel systems. Consider the costs of purchasing a GPU against the potential costs of multiple computer systems and a high performance interconnection network, or a modern multi-core processor with a large number of cores. For instance, while the

GPU we use for our experiments is out-dated and can no longer be easily purchased, a Canadian retailer [19] lists a comparable GPU (in terms of CC count) at $125 (Canadian dollars).

A recent work by Delong and Boykov [5] claimed to achieve near-linear speedup for a push-relabel max flow algorithm on an 8-core Intel processor. Their work applies the maximum flow algorithm to grid graphs for image processing. Near-linear speedup on an 8-core processor implies a maximum of 8 times speedup compared to a sequential implementation. We have shown our GPU implementation of the MPM algorithm to be capable of up to 9.5 times speedup. In this very rough comparison, we observe a mid-range GPU providing improved speedup over a high-end multi-core CPU in the same category of algorithm.

For another example we look to the recent work of Li and Wada [27] on parallel PSO. Their work investigated the performance of a parallel PSO algorithm that focused on hiding communication latencies behind computation. They parallelized a single PSO swarm across 16 nodes, each running modern Intel Xeon processors. Their results showed a roughly linear speedup — 16 nodes resulted in slightly lower than 16 times speedup compared to a single node. The authors tested their PSO application against benchmark optimization problems. Our GPU algorithm, which focused on solving a problem with some irregular properties, produced up to 37 times speedup compared to a sequential CPU implementation.

While we cannot directly compare the results of either of these examples to our implementations, we mention them to show that the relatively inexpensive GPU provides extremely competitive performance for both regular and irregular algorithms

when compared to traditional CPU architectures. Given the high costs associated with a consumer-grade Intel hexacore CPU[2] ($\$1,180$, roughly 9 times the price of our mid-range GPU [21]), or even a quad-core CPU [20] at $300, as well as the cost associated with interconnection networks required to set up multiple systems (or the added costs of motherboards supporting multiple CPUs), we feel that the GPU offers a substantial improvement on existing traditional architectures when viewed from the price-to-performance standpoint.

As a result, we stand by our decision for comparing the performance of our GPU algorithms to sequential CPU implementations and measuring the absolute speedup. We feel that our performance tests show the most value when we investigate the absolute speedup possible for a parallel algorithm on the GPU. By doing so, we more readily understand the strengths and weaknesses of the GPU by comparing the speedup results of each algorithm.

[2]We choose an Intel CPU here as the work from Delong and Boykov [5] and Li and Wada [27] use Intel processors. As a quote for the 8-core variants was not available, we use a quote for the lesser-priced 6-core variants. Despite the lower price of the 6-core variants, our argument does not change and the GPU still maintains a significantly lower cost.

Chapter 8

Conclusions and Future Work

We have studied the design, implementation, and performance of four algorithms for solving four regular and irregular problems on the GPU. Through this study we have gained a greater understanding of what strategies improve performance on the GPU (and to what degree) as well as the efforts required to achieve greater performance. We have learned what it takes to efficiently map algorithms with regular and irregular properties to the hardware, and discovered which optimization efforts work and which do not.

Starting with the binomial lattice algorithm for pricing lookback options we investigated a regular, data-parallel algorithm that represented a very good fit for the GPU. We showed that global memory coalescing was possible to exploit in full, and that load balancing warps was provided "for free". With our performance analysis, we saw tremendous speedup compared to a sequential CPU implementation. These results further emphasized the idea that regular algorithms map very well to the GPU architecture.

We then moved on to the BFS algorithm, chosen for its irregular properties as well as its simplicity. Due to the highly parallel nature of the GPU we were able to map threads to vertices in a one-to-one fashion, essentially using very fine-grained parallelism not readily available on many other architectures. We noted that the optimizations we took for granted in the option pricing algorithm were not always possible to accomplish in the BFS. In essence, the irregularity locked out the potential of exploiting global memory coalescing. With extra effort, however, we were able to load balance warps, but this load balancing came at a substantial cost. In the end, this optimization step resulted in longer execution times than the basic algorithm. With the BFS we saw a steep drop in the speedup when compared to the option pricing algorithm. Not unexpectedly, the performance results emphasized the difficulties associated with irregular algorithms on the GPU.

We found these difficulties magnified when we moved on to our more complex algorithm for solving an irregular problem, the MPM algorithm. Here we had not one but a number of phases to focus on. By taking into consideration the parallel nature of our implementation, our design merged some phases together, resulting in a total of three phases required for the algorithm. Mapping each phase to an individual kernel provided us with the necessary global synchronization present in the original algorithm. Interestingly enough, we observed improved performance with the active vertices modification when applied to the pruning and push/pull flow phases — the opposite of what happened with the BFS/layered network construction phase. Our performance results showed a speedup of up to 9.5, clearly lower than the results we observed for the BFS and due to the poorer performance of the pruning and push/pull

phases.

Finally, we investigated a multi-swarm PSO algorithm for solving the task matching problem. While the solution quality results were unfortunate, we felt that the performance results were impressive. By exploiting texture memory, we reduced the negative impact the irregular fitness update phase had on the execution time of our algorithm. For smaller data sets we further used shared memory in this phase, reducing the overall reliance on global memory. With the other phases being very regular and data-parallel in nature, we achieved significant speedup results, up to 37 times faster than a sequential CPU implementation.

By combining our knowledge of the design of each algorithm with our in-depth performance analyses, we discussed the major differences between each algorithm on the GPU. While the general data of three of our algorithms was based around graph structures, we described how the lookback option pricing algorithm achieved a much more optimal mapping to the GPU due to its regular nature. The BFS and MPM algorithms, on the other hand, required data structures that maintained the original graph structure of the data and reduced the possibilities for global memory coalescing and load balancing. PSO represented a special case where we were able to exploit these optimizations in almost all the phases by restructuring the ordering of data, but could not do the same for the fitness update phase. In all, our performance results showed that the speedup lowers as the algorithm becomes more complex and irregular.

One major point of future work that we identify immediately is that of hybrid computing. We investigated the potential of switching back to the CPU for the look-

back option pricing problem but did not observe any significant level of performance improvement. While we feel that the BFS and MPM algorithms do not represent ideal candidates for hybrid computing, due to the large amount of data required (which would have to be transferred to the CPU's main memory), we believe that this may be interesting to study for multi-swarm PSO. With multi-swarm PSO, each swarm works independently for n iterations until we swap particles. Even with the repulsion factor included, only a small amount of data (the location of the global best particle) must be transmitted between swarms every iteration. As a result, we believe it is worthwhile to investigate the performance ramifications of executing at least one swarm on the CPU, while the GPU manages the other swarms. Perhaps multi-threading on the CPU could be used as well in order to improve the performance further. While a hybrid approach failed for the lookback option pricing algorithm, we believe the potential of this approach for multi-swarm PSO merits an investigation.

While not directly related to our parallelization work, we want to bring attention to the lack of solution quality in our PSO for task matching algorithm. While PSO may not provide an ideal algorithm for solving this problem, we believe combining ideas from maximum flow warrants an investigation. We plan to map the task matching problem to a graph structure and treat the problem as a maximum flow problem, where vertices represent machines, and edges represent task-trading paths from machines with high MAT to machines with low MAT. We would further compare a future parallel implementation on the GPU to our standard MPM maximum flow iteration to gauge performance.

Finally, the GPU architecture is constantly evolving year by year. Not only are

the core counts and computational performance increasing, but the hardware manufacturers add additional feature sets as well, such as the on-board cache memory we discussed for the Fermi architecture. Beyond even these incremental steps, however, lie new architectures such as AMD's Fusion architecture, which brings an integrated GPU into the same die as the CPU. With both GPU and CPU sharing the same memory, we feel that experimenting with hybrid computing becomes even more interesting. We believe future investigation into such an architecture would make a fine companion piece to the research covered in this thesis. Not only would we investigate the viability of a Fusion-style architecture for parallel computing with regular and irregular algorithms, but we would also be able to compare the design, performance, and optimization steps with our existing work on the GPU.

Through our work, we have shown that the GPU has proven itself capable of accelerating both regular and irregular algorithms. While the performance of algorithms with irregular properties clearly suffers when compared to those with regular properties, the overall results still show dramatic speedups for all types of algorithms. Whether or not the close coupling of the GPU and CPU provides more added value than the typical discrete GPU card is an area we are absolutely interested in pursuing in the future.

Bibliography

[1] R. J. Anderson and J. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In *4th Annual Symposium on Parallel Algorithms and Architectures*, 1992.

[2] D. A. Bader and V. Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *18th ISCA International Conference on Parallel and Distributed Computing Systems*, pages 41–48, 2005.

[3] John C. Cox, Stephen A. Ross, and Mark Rubinstein. Options pricing: A simplified approach. *Journal of Financial Economics*, 7:229–263, 1979.

[4] Lucas de P. Veronese and Renato A. Krohling. Swarm's flight: Accelerating the particles using C-CUDA. In *IEEE Congress on Evolutionary Computation*, pages 3264–3270, Trondheim, Norway, May 18–21 2009.

[5] Andrew Delong and Yuri Boykov. A scalable graph-cut algorithm for N-D grids. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, Anchorage, Alaska, USA, June 24–26 2008.

[6] E.A. Dinic. Algorithm for solution of a problem for maximal flow in a network with pwer estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[7] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the Association for Computing Machinery*, 19:248–264, 1972.

[8] Michael Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[9] L.R. Ford and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[10] Richard F. Freund, Michael Gherrity, Stephen Ambrosius, Mark Campbell, Mike Halderman, Debra Hensgen, Elaine Keith, Taylor Kidd, Matt Kussow, John D. Lima, Francesca Mirabile, Lantz Moore, Brad Rust, and H. J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with smartnet. In *The Seventh IEEE Heterogeneous Computing Workshop*, pages 184–199, Orlando, Florida, USA, Mar. 30 1998.

[11] Noriyuki Fujimoto. Faster matrix-vector multiplication on GeForce 8800GTX. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, pages 1–8, Miami, Florida, USA, April 14–18 2008. IEEE.

[12] A.V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, Laboratory of Computer Science, MIT, 1987.

[13] A.V. Goldberg and R.E. Tarjan. A new approach to maximum-flow problem. *Journal of the Association for Computing Machinery*, 35:921–940, October 1988.

[14] Khronos Group. OpenCL — the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`, September 2011.

[15] Pawan Harish, Vibhav Vineet, and P.J. Narayanan. Large graph algorithms for massively multithreaded architectures. Technical Report IIIT/TR/2009/74, International Institute of Information Technology Hyderabad, Hyderabad, India, February 2009.

[16] Espen Gaarder Haug. *The Complete Guide to Option Pricing Formulas*. McGraw-Hill, 2nd edition, 2006.

[17] John Hull. *Options, Futures, and other Derivative Securities*. Prentice Hall, May 2008.

[18] Mohamed Hussein, Amitabh Varshney, and Larry Davis. On implementing graph cuts on CUDA. In *First Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, October 4 2007.

[19] Memory Express Computer Products Inc. eVGA GeForce GTS 450. `http://www.memoryexpress.com/Products/MX30370`, October 20 2011.

[20] Memory Express Computer Products Inc. Intel Core i7-960. `http://www.memoryexpress.com/Products/MX27708`, October 20 2011.

[21] Memory Express Computer Products Inc. Intel Core i7-990X. `http://www.memoryexpress.com/Products/MX32412`, October 20 2011.

[22] Kennedy J. and Eberhart R. Particle swarm optimization. In *IEEE Intl. Conf. on Neural Networks*, pages 1942–1948 (vol.4), Perth, Australia, Nov. 27–Dec. 1 1995. IEEE.

[23] Grégoire Jauvion and Tuan Nguyen. Parallelized trinomial option pricing model on GPU with cuda. `http://www.arbitragis-research.com/cuda-in-computational-finance/coxross-gpu.pdf/at_download/file` - unpublished, August 7th 2008.

[24] Qinma Kang, Hong He, Hongrun Wang, and Changjun Jiang. A novel discrete particle swarm optimization algorithm for job scheduling in grids. In *Fourth Intl. Conf. on Natural Computation*, pages 401–405, Jinan, China, Aug. 25–27 2008. IEEE.

[25] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Cascaval. How much parallelism is there in irregular applications? In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–14, Raleigh, NC, USA, February 2009. ACM.

[26] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of Programming Language Design and Implementation 2007*, pages 211–222, San Diego, CA, USA, June 2007.

[27] Bo Li and Koichi Wada. Communication latency tolerant parallel algorithm for particle swarm optimization. *Journal of Parallel Computing*, 37(1):1–10, January 2011.

[28] J. J. Liang and P. N. Suganthan. Dynamic multi-swarm particle swarm optimizer with local search. In *IEEE Congress on Evolutionary Computation*, pages 522–528, Edinburgh, UK, Sept. 2–4 2005.

[29] V.M. Malhotra, M. Kumar, and S.N. Maheshwari. An O($|V|^3$) algorithm for finding maximum flows in networks. *Information Processing Letters*, 7(6):277–278, May 1978.

[30] Pedro J. Martín, Roberto Torres, and Antonio Gavilanes. CUDA solutions for the SSSP problem. In *Proceedings of the 9th International Conference on Computational Science (ICCS '09)*, pages 904–913, Baton Rouge, LA, USA, May 25–27 2009. Springer-Verlag.

[31] Luca Mussi, Stefano Cagnoni, and Fabio Daolio. Gpu-based road sign detection using particle swarm optimization. In *Ninth International Conference on Intelligent Systems Design and Applications*, pages 152–157, Pisa, Italy, Nov. 30–Dec. 2 2009. IEEE.

[32] Luca Mussi, Fabio Daolio, and Stefano Cagnoni. Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. *Information Sciences*, In Press, Sept. 3 2010.

[33] NVIDIA. *CUDA Programming Guide Version 3.1*. NVIDIA, May 2010.

[34] NVIDIA. *Tuning CUDA Applications for Fermi*. NVIDIA, February 2010.

[35] NVIDIA. *CUDA C Best Practices Guide*. NVIDIA, May 2011.

[36] Nvidia. Nvidia CUDA developer zone. `http://developer.nvidia.com/category/zone/cuda-zone`, September 2011.

[37] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, N.J., 1982.

[38] Victor Podlozhnyuk. Binomial option pricing model. `http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/binomialOptions/doc/binomialOptions.pdf`, April 2008.

[39] Tobias Preis, Peter Virnau, Wolfgang Paul, and Johannes J Schneider. Accelerated fluctuation analysis by graphics cards and complex pattern formation in financial markets. *New Journal of Physics*, 11(9), September 2009.

[40] G. Sudha Sadasivam and Viji Rajendran. An efficient approach to task scheduling in computational grids. *International Journal of Computer Science and Applications*, 6(1):53–69, 2009.

[41] Shubhabrata Sengupta, Mark Harris, Michael Garland, and John D. Owens. *Scientific Computing with Multicore and Accelerators*, chapter 19, pages 413–442. CRC Press, December 2010.

[42] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John Owens. CUDPP project. `http://code.google.com/p/cudpp/`, September 2010.

[43] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Efficient parallel scan algorithms for GPUs. Technical Report NVR-2008-003, NVIDIA, December 2008.

[44] Y. Shi and R. Eberhart. A modified particle swarm optimizer. In *IEEE World Congress on Computational Intelligence*, pages 69–73, Anchorage, Alaska, USA, May 4–9 1998. IEEE.

[45] Y. Shiloach and U. Vishkin. An $o(n^2logn)$ parallel max-flow algorithm. *Journal of Algorithms*, 3:128–146, 1982.

[46] Steven Solomon, Ruppa K. Thulasiram, and Parimala Thulasiraman. Option pricing on the GPU. In *IEEE 12th International Conference on High Performance Computing and Communcations (HPCC)*, pages 289–296, Melbourne, Australia, September 1–3 2010.

[47] Steven Solomon and Parimala Thulasiraman. Performance study of mapping irregular computations on GPUs. In *The 11th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-10)*, Atlanta, USA, April 19–23 2010. IEEE.

[48] Steven Solomon, Parimala Thulasiraman, and Ruppa K. Thulasiram. Exploiting parallelism in iterative irregular maxflow computations on GPU accelerators. In *IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 297–304, Melbourne, Australia, September 1–3 2010.

[49] Steven Solomon, Parimala Thulasiraman, and Ruppa K. Thulasiram. Collaborative multi-swarm PSO for task matching using graphics processing units. In *13th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1563–1570, Dublin, Ireland, July 12–16 2011.

[50] Reiji Suda, Takayuki Aoki, Shoichi Hirasawa, Akira Nukada, Hiroki Honda, and Satoshi Matsuoka. Aspects of GPU for general purpose high performance computing. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 216–223, Yokohama, Japan, 2009. IEEE Press.

[51] M. Fatih Tasgetiren, Yun-Chia Liang, Mehmet Sevkli, and Gunes Gencyilmaz. Particle swarm optimization and differential evolution for the single machine total weighted tardiness problem. *International Journal of Production Research*, 44(22):4737–4754, Nov. 2006.

[52] Frans van den Bergh and Andries P. Engelbrecht. A cooperative approach to particle swarm optimization. *IEEE Trans. on Evolutionary Computation*, 8(3):225–239, June 2004.

[53] Leonardo Vanneschi, Daniele Codecasa, and Giancarlo Mauri. An empirical comparison of parallel and distributed particle swarm optimization methods. In *The Genetic and Evolutionary Computation Conference*, pages 15–22, Portland, Oregon, USA, July 7–11 2010.

[54] Fabien Viger and Matthieu Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In *The Eleventh International Computing and Combinatorics Conference*, Lecture Notes in Computer Science, pages 440–449, Kunming, China, August 16–29 2005.

[55] Fabien Viger and Matthieu Latapy. Graph generation software. `http://www-rp.lip6.fr/~latapy/FV/generation.html`, September 2009.

[56] Vibhav Vineet and P. J. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–8, Anchorage, AK, June 2008. IEEE.

[57] Bu Yan-Ping, Zhou Wei, and Yu Jin-Shou. An improved PSO algorithm and its application to grid scheduling problem. In *Intl. Symp. on Computer Science and Computational Technology*, pages 352–355, Shanghai, China, 20–22 Dec. 2008. IEEE.

[58] Lei Zhang, Yuehui Chen, Runyuan Sun, Shan Jing, and Bo Yang. A task scheduling algorithm based on PSO for grid computing. *Intl. J. of Computational Intelligence Research*, 4(1):37–43, 2008.

[59] You Zhou and Ying Tan. GPU-based parallel particle swarm optimization. In *IEEE Congress on Evolutionary Computation*, pages 1493–1500, Trondheim, Norway, May 18–21 2009.