# A PERFORMANCE MODEL FOR GPU ARCHITECTURES: ANALYSIS AND DESIGN OF FUNDAMENTAL ALGORITHMS

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAIʻI AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

MAY 2018

By

Ben Karsin

Thesis Committee:

Nodari Sitchinava, Chairperson
Henri Casanova
Lipyeow Lim
Jason Leigh
Philip von Doetinchem

Keywords: parallel, algorithms, many-core, GPU, sorting, model

# ACKNOWLEDGMENTS

First, I would like to thank my wife, Kelsey, for being patient and supportive during my long journey into Academia. I would also like to thank my advisor, Nodari Sitchinava, for guiding me in this work and encouraging me to dedicate myself to the field. I would like to thank Henri Casanova for starting me down this path and for constantly lending his vast knowledge. I would also like to thank my dissertation committee members, Lipyeow Lim, Jason Leigh, and Philip von Doetinchem. Finally, I would like to thank Kyle Berney for the helpful discussions related to this work.

# ABSTRACT

Over the past decade, "many-core" architectures have become a crucial resources for solving computationally challenging problems. These systems rely on hundreds or thousands of simple compute cores to achieve high computational throughput. However, their divergence from traditional CPUs makes existing algorithms and models inapplicable. Thus, developers rely on a set of heuristics and hardware-dependent rules of thumb to develop algorithms for many-core systems, such as GPUs.

This dissertation attempts to remedy this by presenting the Synchronous Parallel Throughput (SPT) model, a general performance model that aims to capture the factors that most impact algorithm performance on many-core architectures. The model focuses on two factors that often create performance bottlenecks: memory latency and synchronization overhead. We instantiate the SPT model on three separate modern GPU platforms using a series of microbenchmarks that measure hardware parameters such as memory access latency and peak bandwidth. We further show how multiplicity affects performance by hiding latencies and increasing overall throughput. We consider three fundamental problems as case studies in this dissertation: general matrix-matrix multiplication, searching, and sorting. Using our SPT model, we analyze a state-of-the-art library implementation of a matrix multiplication algorithm and show that our model can generate runtime estimates with an average error of 5% across our GPU platforms. We then consider the problem of batched predecessor search in the context of two levels of the GPU memory hierarchy. In slow, global memory, we demonstrate the accuracy of the SPT model, while in fast, shared memory, we determine that the memory access patterns create a performance bottleneck that degrades performance. We develop a new searching algorithm that improves the access pattern and increases performance by up to 293% on our GPUs. Finally, we look at comparison-based sorting on GPUs by analyzing two state-of-the-art algorithms and, using our SPT model, determine that they each suffer from bottlenecks that stifle performance. With these bottlenecks in mind, we develop GPU-MMS, our GPU-efficient multiway mergesort algorithm, and demonstrate that it outperforms highly optimized library implementations of existing algorithms by an average of 21% when sorting random inputs and up to 67% on worst-case input permutations. These case studies demonstrate both the accuracy and applicability of the SPT model for analyzing and developing GPU-efficient algorithms.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1
# INTRODUCTION

In recent years, massively parallel, "many-core" architectures have become increasingly popular for solving computationally challenging problems [76, 65, 44, 64]. These architectures, which include modern Graphics Processing Units (GPUs) [44] and Intel Xeon Phi processors [38], are composed of thousands of simple compute cores and are designed to deliver high computational throughput, enabling them to outperform traditional CPUs for many applications [11, 58, 68]. For some applications, these architectures can achieve an order-of-magnitude performance speedup over comparable CPU systems. However, no general model is available to accurately analyze the performance of algorithms on these types of systems. In fact, many details, such as the memory hierarchy and execution pipeline, are not well understood, forcing algorithm designers to rely on heuristics and trial-and-error methods.

*This dissertation aims to remedy these shortcomings by proposing a general model that incorporates the factors that most impact the performance of these high-throughput, parallel architectures. Using a series of microbenchmarks, we instantiate our model on three modern GPU platforms, allowing us to accurately estimate the runtime of an algorithm on a given platform. As case studies, we consider three fundamental problems: matrix multiplication, searching, and sorting; and we show that our model can both accurately predict runtime and provide insight into designing and optimizing algorithms for GPUs.*

This introduction provides an overview of many-core architectures, with an emphasis on GPUs, outlines features of many-core systems that are most relevant to our model, discusses related performance models, and motivates the work.

## 1.1 Many-core Architectures

Until recent years, single-core processing units saw consistent performance improvements following Moore's law. However, as power consumption and heat dissipation became more problematic, designers turned to parallelism as another way to improve performance. Today, processing units in nearly all commercially-available desktop computers are multicore, with many having dozens of compute cores. However, each of the processor cores available in these systems is complex, with its own internal logic and cache systems, limiting the number of cores that can fit on a single chip. Additionally, maintaining cache coherence across many such processor cores is difficult. Many-core architectures, on the other hand, offer hundreds or thousands of very simple compute cores. While these highly parallel systems provide a great deal of computational power, efficiently utilizing it can prove difficult. Groups of compute cores share resources such as memory caches and instruction units. These architectures differ greatly from traditional CPUs, for which most existing

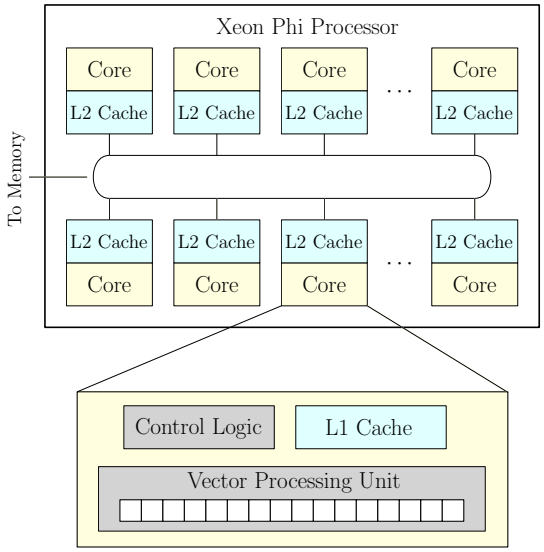


Figure 1.1: High-level view of the Intel Xeon Phi processor architecture. Each *core* has a vector processor that can operate on many elements per cycle.

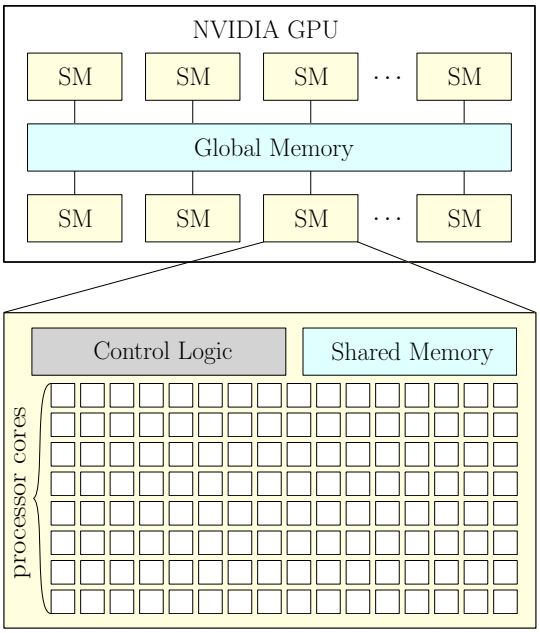


Figure 1.2: High-level view of the architecture of NVIDIA GPUs. GPUs have a series of *streaming multiprocessors* (SMs), each of which has many cores.

algorithms were developed. As such, many algorithms and performance optimizing techniques are not well-suited for many-core architectures. Furthermore, the details of currently available many-core systems vary greatly, making it difficult to develop general, platform-independent algorithms. Despite this, today many-core systems are an important resource for high-performance computing. Commercially available many-core systems, such as Graphics Processing Units (GPUs) and Intel Xeon Phi coprocessors, provide a cheap resource that can be used to solve computationally intensive problems. Additionally, many of the fastest supercomputers today employ many-core processing units in their compute nodes (e.g., the Sunway TaihuLight supercomputer [26], the Zettascaler supercomputer [6], etc.).

Figures 1.1 and 1.2 provide a high-level architectural overview of Intel Xeon Phi processors [38] and NVIDIA GPUs [65], respectively. We see that, while details of these architectures differ, they are both made up of a series of computational units, called *cores* on Xeon Phi and *streaming multiprocessors* (SMs) on NVIDIA GPUs. In each case, each computational unit has a dedicated cache and a large number of compute cores (a vector processor on Xeon Phi and many compute cores on GPUs). This illustrates that, while different many-core systems can vary greatly in architectural details, they have similar overall structures. There are several common features that most many-core systems have in order to efficiently use the large number of compute cores and maximize computational throughput:

- a multi-level memory hierarchy with varying access scope,
- a mult-level compute hierarchy with different levels of synchronization available, and
- fast context-switching, enabling the use of multiple threads (i.e., oversubscription) to hide memory access latency.

In this dissertation, we present the Synchronous Parallel Throughput (SPT) model, a general performance model that is designed to model the performance of parallel systems that rely on thousands of execution threads to mitigate the cost of high-latency operations. Our model focuses on three factors that most impact the performance of many-core architectures: memory access latency, bandwidth limitations, and synchronization overhead.

### 1.1.1 Latency and Bandwidth

The latency associated with a memory access on many-core systems is frequently orders of magnitude larger than that of performing a computation. Thus, memory-bound applications can cause cores to sit idle while data is being retrieved, resulting in wasted computational throughput and performance loss. Many high-throughput architectures combat this with techniques such as *oversubscription* [65] (spawning multiple threads per core) and *instruction-level parallelism* [65] (executing a series of independent instructions) that, effectively, reduce memory latency. However, these techniques provide limited benefits, as every memory system has a peak bandwidth that cannot be surpassed. The SPT model incorporates these concepts when evaluating algorithm performance and, as we see in our case studies, these factors must be considered when designing GPU-efficient algorithms.

### 1.1.2 Synchronization

The high degree of parallelism afforded by many-core architectures makes synchronization a necessary and potentially costly operation. The SPT model incorporates the cost of system-wide barrier synchronization. However, some many-core architectures allow for fine-grain synchronization between subsets of cores or threads. We incorporate the cost of fine-grain synchronization by considering such synchronizations as any other type of operation. Through experimentation, we show that it accurately models the cost of such synchronization on our GPU platforms.

## 1.2 Overview of GPU Architectures

While the SPT model, presented in this dissertation, is a general performance model for many-core architectures, we demonstrate its applicability by using it to analyze the performance of a series of algorithms on three modern NVIDIA GPU platforms. Thus, in this section we provide an

overview of modern GPU architectures, highlighting key features that we look at in more detailed when we instantiate our model in Chapter 4. For more information about GPU architectures or any features we discuss, see standard references [65, 44]. As illustrated in Figure 1.2, modern GPUs comprise thousands of physical processing cores, organized hierarchically into *streaming multiprocessors* (SMs). Each SM contains a small, fast *shared memory* that is private to each SM, and a large, slower *global memory* that is shared among all SMs.

### 1.2.1 Execution organization

To achieve optimal utilization of all of its thousands of processor cores, as well as hide memory and instruction latency, GPUs support the execution of many more threads than physical processor cores. To manage the execution of so many threads, the programmer groups threads into *thread-blocks* (TBs), also called cooperative thread arrays. The GPU schedules all threads of a TB on a single SM, allowing them to access and communicate via the same shared memory partition. The GPU further partitions TBs into *warps*, each containing $w$ threads[1]. All threads within a warp execute in SIMT fashion, requiring they execute the same operations at each step. Since threads within a warp execute in lock-step, they implicitly synchronize after each operation. For more detailed information on the SIMT execution paradigm and branch divergence, we direct interested readers to [53]. Synchronization can also be performed between warps within the same thread-block through the use of the SYNCTHREADS command. SYNCTHREADS is a blocking[2] operation that forces all threads within the thread-block to wait until they all call it. Finally, GPUs allow for synchronization across all threads through the use of the CUDADEVICESYNCHRONIZE command.

### 1.2.2 Memory hierarchy

The modern GPU architecture includes a memory hierarchy with each level a having different latency, throughput, access scope, and optimal access pattern. In Chapter 4 discuss the details and measure performance impact of each type of memory.

#### Global memory

Global memory is the primary way to communicate between threads of different TBs, but previous works suggest that it is at least an order of magnitude slower than other memory types [65, 24], such as shared memory. Consequently, global memory use must be limited to achieve peak performance. To achieve high memory throughput, all threads within a warp must, together, access consecutive elements in global memory. This is called a coalesced memory access, allowing a warp to reference $w$ elements in a single access. If we consider a warp to be a single unit of execution, we say that global memory accesses are performed in *blocks* [65], where $B = w$ elements are accessed from a

[1] $w = 32$ for most modern GPUs

[2] An operation is *blocking* if it must complete before the calling thread is able to continue execution.

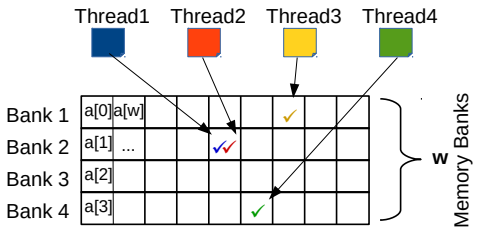


Figure 1.3: Illustration of shared memory access patterns that result in no bank conflicts.

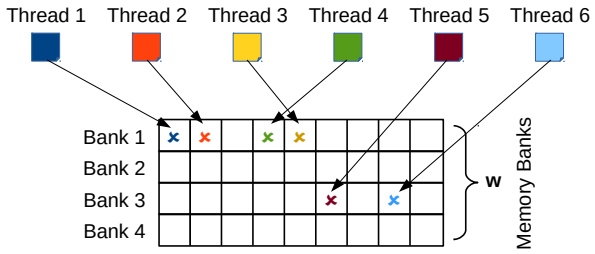


Figure 1.4: When multiple threads access the same shared memory bank, a conflict occurs and accesses are serialized.

single transfer, as in the PEM model [9], discussed in Section 2.1.1. See [61, 24] for a more in-depth discussion of the global memory access pattern.

### Shared memory

Shared memory is a smaller, faster memory that is private to each SM. Each TB designates its required usage and that much shared memory is assigned to that TB while it is resident on the SM. Consequently, if a TB requires a large amount of shared memory, it may prevent additional TBs from being scheduled on the same SM, potentially reducing performance. The shared memory of each SM is implemented as a series of *memory banks*, each of which can be accessed independently in parallel (Figure 1.3). However, if threads within the same warp attempt to access the same memory bank, as illustrated in Figure 1.4, a conflict occurs and accesses are serialized. Shared memory is also capable of *multicasting*, allowing the same address to be accessed by multiple threads at once, thus only concurrent accesses to distinct cells of a memory bank cause conflicts. Note that for most modern GPUs, the number of memory banks and threads per warp is equal. Thus, we assume both the number of memory banks and threads per warp is $w$.

### Registers

Registers are available as limited number of fast, thread-private memory locations. On most modern GPUs, each thread is limited to a small number of registers (e.g., 255), and each SM also has a limited number of total registers. Any registers that a thread uses beyond 255 will be placed in *local memory*, which is a portion of dedicated memory that is cached in shared memory. While a thread can access any of its registers in unit time, the access pattern must be know at compile time, which can limit the applicability of registers. We look at the impact of register usage on performance in Chapter 4.

**Cache mechanisms**

The L3 cache available on most NVIDIA GPUs is a mechanism to mitigate the cost of uncoalesced accesses. My caching single elements that are frequently read, it can reduce the requirement for repeated uncoalesced accesses to single elements. The L2 cache is a dedicated portion of shared memory that the GPU uses to automatically cache elements that are accessed from global memory. While the performance implications of cache hits are similar to manually loading data into shared memory, we do not explicitly consider the impact of the cache in this work and instead simply view cache hits as accesses to the faster shared memory, rather than to global memory.

## 1.3 Performance Models

At a high level, the SPT model views an algorithm as a series of *kernels*, separated barrier synchronization. This is similar to the Bulk Synchronous Parallel (BSP) model [78] (an overview is provided in Section 2.1) if we equate kernels to *supersteps*. Unlike the BSP, however, we model the performance of each kernel by identifying a set of *operations* and computing the achieved throughput of each. The operations we consider depend on the particular many-core architecture we are considering, though we define the model generally, without considering a specific system. Nevertheless, there are certain commonalities among operations that impact performance, including accessing memory or performing operations in registers. To model the frequency of these operations for a given algorithm, we use some well-known models of computation.

The Random Access Machine (RAM) [17] model, one of the most well-known computational models, considers the complexity of an algorithm to be the number of memory accesses, where any memory cell can be accessed in unit time. The Parallel Random Access Machine (PRAM) [37] model extends this by considering an architecture with multiple processors. This dissertation uses the Concurrent Read Exclusive Write (CREW) PRAM model, since the relevant memory systems that we consider allow for multiple threads to concurrently read from the same memory location. The Parallel External Memory (PEM) model [9], which measures algorithmic performance in accesses to a particular memory system that allows for *blocked* memory access (i.e., $B$ elements are accessed in a single memory operation) is also relevant to this work, as many slow memory systems exhibit blocked access patterns to increase overall throughput. We provide a more thorough overview of these models, as well as other related performance models in Section 2.1.

## 1.4 Case studies

As case studies, this work looks at three fundamental problems: matrix-matrix multiplication, searching, and sorting. The algorithms we consider in each of these areas has different characteristics (e.g., computational complexity, memory access patterns, etc.), allowing us to evaluate the accuracy

of our performance model on a wide range of applications.

### 1.4.1 General matrix-matrix multiplication

As a first case study, we look at a simple yet compute-bound problem that is well-suited to many-core architectures: matrix multiplication. General matrix-matrix multiplication (GEMM) is one of the most fundamental dense linear algebra operations with a wide range of applications. Algorithms that solve GEMM are known to be compute, with the naive solution requiring $O(n^3)$ work on $O(n^2)$ data. The current state-of-the-art library implementations use this naive algorithm, though they use optimizations that reduce the number of global and shared memory accesses. In Chapter 5, we model the algorithm used by a state-of-the-art library implementation that avoids all system-wide synchronization and utilizes levels of the GPU memory hierarchy.

### 1.4.2 Searching

As a second case study, we consider the fundamental problem of searching, with a specific focus on performing batches of *predecessor search* queries. The performance of these operations are frequently memory-bound and exhibit non-deterministic memory access patterns that depend on the queries themselves. On the GPU, this leads to sub-optimal access patterns to two levels of the memory hierarchy, degrading performance. Thus, we consider each type of memory individually, analyzing the access pattern for worst-case sets of queries as well as estimating performance degradation for random queries. In shared memory, we develop a novel binary search algorithm that achieves up to 293% speedup on our platforms, compared to a naive approach. In *global memory*, we look at implicit search layouts and show that our model correctly predicts that performance gains of using B-tree and BST memory layouts.

### 1.4.3 Sorting

Finally, we consider the general problem of comparison-based sorting. Sorting algorithms have more of a balance of computation and memory accesses, while requiring more synchronization between threads. We use the SPT model to analyze several state-of-the-art sorting algorithms for the GPU and identify bottlenecks that degrade performance on our hardware platforms. Using the results of our analysis, we present a new multiway mergesort algorithm that incorporates several novel optimizations that mitigate these bottlenecks and improve performance. We show that our algorithm achieves an asymptotically optimal number of global memory accesses while performing well in practice. Empirically, our multiway mergesort outperforms current state-of-the-art GPU sorting implementations by up to 32.7%. Furthermore, we identify input permutations that cause these library implementations to access memory in a way that further degrades performance. On these inputs, our algorithm achieves up to a 67.2% speedup on our three platforms.

## 1.5 Dissertation Organization

This dissertation is organized as follows: Chapter 2 provides background information on relevant performance models, as well as an overview of related work. In Chapter 3 we present the general Synchronous Parallel Throughput model (SPT model). We instantiate our model in Chapter 4 by introducing a series of microbenchmarks that we run on our three GPU hardware platforms. In Chapter 5 we look at general matrix-matrix multiplication as a case study. In Chapter 6 we apply our model to the analysis of searching in each level of the GPU memory hierarchy. In Chapter 7 we analyze a series of GPU sorting algorithms and present our new multiway mergesort algorithm. Finally, in Chapter 8 we present conclusions and discuss interesting areas of future work.

# CHAPTER 2
# BACKGROUND AND RELATED WORK

In this chapter, we present a survey of related work. We first provide an overview relevant and well-known models that we can use to analyze memory access patterns. We then discuss models that are related to (or are applicable to) many-core and GPU architectures, with an emphasis on their memory hierarchies. Finally we provide a broad survey of relevant related work. For our case studies, we also include a more detailed discussion of related work in each corresponding chapter.

## 2.1 Performance Models

As discussed in Section 1.3, the Random Access Machine (RAM) [17] model is one of the simplest models that can be used to determine the complexity of an algorithm as the number of unit-time memory accesses it performs. The Parallel Random Access Machine (PRAM) [37] model provides a parallel extension of this model, where we model perform by parallel accesses to memory. In PRAM we measure complexity as work (total work done by all processors) and depth (maximum work done by a single processor, if we have unlimited processors). Brent's scheduling principle [17] allows us to further model the time complexity of an algorithm for a given number of processors, $P$. In this work we only consider the Concurrent Read Exclusive Write (CREW) PRAM model, which allows only one processor to write to a memory location at a time. While PRAM is useful in modeling simple memory systems, many types of memory do not have random unit-time access.

### 2.1.1 Parallel External Memory Model

The external memory model [4] is a well-known model analyzing the performance of algorithms that run on a computer with both a fast internal memory and a slow external memory, and whose performance is bound by the latency for slow memory access, or I/O. The Parallel External Memory (PEM) model [9] extends this model by allowing multiple processors to accesses external memory in parallel. The PEM model relies on the following problem/hardware-specific parameters

- $N$: problem input size,
- $P$: number of processors,
- $M$: internal memory size, and
- $B$: block size (the number of elements accessed during one blocked memory transfer).

Figure 2.1 illustrates how each of these parameters is incorporated into the model. In the PEM model, the *I/O complexity* of an algorithm is determined by the number of parallel blocked

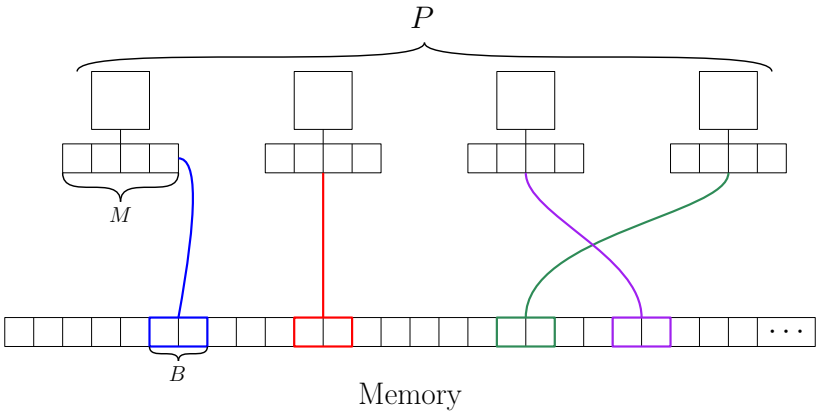


Figure 2.1: Illustration of the Parallel External Memory (PEM) model [9].

memory transfers (I/Os) that it performs. For example, scanning an input list in parallel has an I/O complexity of $\Theta(\frac{N}{PB})$. In the PEM model, sorting an input has a lower bound of

$$sort_{PEM}(N) = \Omega\left(\frac{N}{PB}\log_{\frac{M}{B}}\frac{N}{B} + \log N\right),$$

while predecessor search (given query $q$, find the largest element in a list with value $v \leq q$) has a lower bound of

$$search_{PEM}(N) = \Omega(\log_B N).$$

### 2.1.2 Bulk Synchronous Parallel Model

Rather than modeling only memory accesses, the Bulk Synchronous Parallel (BSP) model [78] models the performance of an algorithm by considering it as a series of *supersteps*. Each superstep consists of a three phases: local computation, communication, and barrier synchronization. Thus, the cost of each superstep is computed as the sum of each phase. The cost of the computation phase is computed by the longest running local computation process, $\max_{i=1}^{p}(w_i)$, where there are a total of $p$ processors and $w_i$ is the cost of computation for processor $i$. The communication cost is computed as $\max_{i=1}^{p}(h_i \cdot g)$, where $h_i$ is the number of messages that processor $i$ sends or receives and each message takes $g$ time. Finally, the cost of barrier synchronization is simply $l$. Thus, the BSP model computes the total runtime of an algorithm of $S$ supersteps as

$$\sum_{s=1}^{S}(w_s + g \cdot h_s + l) = \sum_{s=1}^{S} w_s + g\sum_{s=1}^{S} h_s + Sl,$$

where $w_s$ and $h_s$ are the maximum computation and communication cost at step $s$, respectively.

### 2.1.3 Existing GPU Models

Performance models for GPU architectures fall into two broad categories: quantitative models and asymptotic analysis. Many quantitative GPU models have been proposed that focus on fine grain modeling and the use of benchmarks. Hong and Kim [35] provide one of the first such models for modern GPUs by using microbenchmarks to determine memory bottlenecks on GPUs available at the time. As GPUs evolved, more models [83, 71] and microbenchmarks [82, 13] have been proposed for specific GPU architectures. The use of such fine-grain analysis, however, results in microbenchmarks and models that are hardware-specific and/or overly complicated. By contrast, analytical models that focus on asymptotic performance are generally applicable to a range of architectures that share the same features, as most modern GPUs do. Several such works have been presented over the years, focusing on modeling the memory hierarchy [61, 62], adapting existing models [45, 47], or modeling overall GPU performance [48]. In [61, 62], the authors provide accurate asymptotic estimates for numbers of shared and global memory accesses for given algorithms. Although in this work we could reuse these estimates, instead we derive asymptotic estimates directly from the PRAM [37] and PEM models [9]. Of the models designed to estimate overall GPU performance, the Threaded Many-core Memory (TMM) model [54], AGPU mode [47], and the model presented by Kothapalli et al. [48] provide the most complete view of GPU performance to date. We discuss these models in more detail and compare them to our work in Section 3.2.

## 2.2 Related Work

Over the past decade, many works have focused on designing efficient algorithms to solve classical problems on GPUs and other many-core architectures [21, 58, 70, 39, 69, 73, 28, 31]. These works have introduced several optimization techniques, such as coalesced memory accesses [24, 68, 23], branch reduction [50, 41], and bank conflict avoidance [41, 14]. While these techniques provide performance improvements for GPU algorithms, the works focus on specific aspects of GPUs rather than modeling the overall performance to determine performance bottlenecks. In this dissertation, we present a general performance model that can be used to determine these bottlenecks. As case studies, we consider three primary applications, so we discuss prior work for each of these applications/problems.

### 2.2.1 Matrix Multiplication

General matrix-matrix multiplication (GEMM) is one of the most fundamental dense linear algebra operations. A wide range of application domains rely on GEMM operations, including fluid dynamics, computational statistics, signal processing, and computational geometry. As such, a lot of research

has looked at improving both the theoretical complexity [75, 16, 81] and practical performance [79, 63, 30, 51] of algorithms that perform GEMM.

For two $n \times n$ matrices, the naive GEMM solution requires $O(n^3)$ operations. This was considered optimal until Strassen [75] introduced a method that requires $O(n^{2.81})$ operations. Since then, advances have been made to reduce the exponent (e.g., Coppersmith and Winograd [16] propose an algorithm that requires only $O(n^{2.376})$ operations[1]). While these algorithms reduce the asymptotic complexity of the GEMM operation, they have higher associated constant factors, making it difficult to achieve peak performance in practice. Furthermore, it is not known how to optimize these "fast matrix-multiply" algorithms to reduce memory accesses (i.e., in the I/O and PEM models). Thus, many state-of-the-art library implementations use the naive $O(n^3)$ algorithm.

Many linear algebra library implementations are available that target specific architectures to achieve peak performance [5, 77, 67, 31, 30]. The well-known LAPACK library [5] provides a wide range of linear algebra algorithms optimized for CPU architectures. The cuBLAS [67] and clBLAS [1] libraries are available for GPU architectures in CUDA and openCL, respectively. The MAGMA library, which we focus on in this work, provides implementations optimized for various architectures including GPUs [77, 63], Xeon Phi processors [31], and mixed heterogeneous systems [30].

The algorithms implemented by these libraries have improved over the years to better suit the underlying architectures. Early GPUs were not able to efficiently perform matrix-matrix multiplications. In 2007, however, GPUs with memory hierarchies became available, enabling them to realize compute-bound implementations of GEMM [79]. Since then, practical advances have been made through GPU-efficient algorithm design [63, 2, 51], new autotuning techniques [52, 49], and optimizations available through library implementations [67, 77, 20]. In Chapter 5, we analyze the GEMM algorithm available in the MAGMA library as a case study.

### 2.2.2 Searching

Searching encompasses a broad class of problems, so we focus primarily on *predecessor search*. While, over the years, many approaches have been proposed as alternatives to naive binary search [15, 25, 17], in this work we focus on two well-known approaches, level-order binary search trees [17] and B-trees [15]. B-trees are data structures that allow for I/O-efficient searching by storing $B$ sorted elements at each node. Each node has $(B+1)$ children, thereby enabling predecessor search to be performed by accessing only $\log_{B+1} n$ nodes. Since there are many results on searching, we only focus on recent work relevant to the problem of searching on GPUs [39, 40, 70, 43, 73]. These previous works all focus on dynamic structures that make efficient use of the slow memory accesses like global memory or CPU-GPU data transfers. Kaczmarski [39, 40] studies the construction of the B$^+$-tree data structure to index data in GPU global memory (B$^+$-trees are a variation of B-trees

[1] More recently Williams [81] proposed an algorithm that requires only $O(n^{2.373})$ operations

where keys in internal nodes are duplicates, so all values are contained in the leaf nodes [17]). By focusing on the $B^+$-tree, the author takes advantage of coalesced global memory accesses. Similarly, Shekhar [70] proposes GPU-efficient global memory data structures designed to improve performance of the IP lookup operation. Kim et al. [43] create a hybrid CPU/GPU data structure to achieve high peak query throughput. Soman et al. [73] address the limited memory on the GPU by looking at compressing search tree data structures. We note that none of these works consider efficient searching in shared memory, despite many applications using it as a subroutine [28, 50, 19]. Even current state-of-the-art algorithms such as GPU mergesort [11, 34] require searching in shared memory.

### 2.2.3 Sorting

Comparison-based sorting is the next case study that we consider. According to a recent survey of several GPU libraries [59], the fastest currently-available sorting implementations include the CUB [57], modernGPU (MGPU) [11], and Thrust [34] libraries. CUB employs a GPU-optimized radix sort, and thus can only be applied to primitive datatypes that can be represented as small integers. MGPU and Thrust use variations of mergesort (based on Green et al. [28]) along with many hardware-specific optimizations to achieve peak performance. While highly optimized, these mergesort implementations require sub-optimal numbers of global memory accesses and incur shared memory bank conflicts. Leischner et al. [50] introduced *GPU samplesort*, a randomized distribution sort aimed at reducing the number of global memory accesses. Their work was continued by Dehne et al. [19] with a deterministic version of the samplesort algorithm. The work of Afshani and Sitchinava [3] focuses on shared memory only and presents an algorithm that sorts small inputs in shared memory without bank conflicts. Koike and Sadakane [47] present a multiway mergesort algorithm for the GPU that also aims at reducing global memory accesses. Despite these efforts, no unified, provably efficient, and practical sorting algorithm has been presented. Thus, mergesort remains the algorithm of choice in top-performing GPU libraries [34, 11]. In this dissertation we evaluate the techniques presented in by Afshani and Sitchinava [3], and Koike and Sadakane [47], and identify, via analysis, several improvements that allow our GPU-efficient sorting algorithm to out-perform state-of-the-art implementations in practice.

# CHAPTER 3
# THE SYNCHRONOUS PARALLEL THROUGHPUT MODEL

Massively parallel architectures, such as modern GPUs and Intel Xeon Phi processors, can, at peak performance, significantly out-perform comparable CPU architectures for many applications. One key feature that sets these architectures apart is that they are designed for high throughput, meaning that they focus on a large amount of parallelism and high peak memory bandwidth, rather than fast sequential computation and low latency memory. As as result, these types of processors require a large amount of parallelism to perform well, making many existing algorithms unsuitable. Existing performance models are also not easily applied, as attempting to model so many individual threads of execution becomes unmanageable. In this chapter, we present the Synchronous Parallel Throughput model (SPT model), a general performance model to analyze the performance of algorithms on massively parallel architectures designed to achieve high-throughput using a memory hierarchy and requiring periodic synchronization. In Chapter 4, we apply the SPT model to our three GPU hardware platforms, and in Chapters 5, 6, and 7 we use case studies to evaluate the accuracy and usefulness of our instantiated model. For reference see Tables 3.1 and 3.2 (available at the end of this chapter) for a brief definition of the hardware-dependent and algorithm-dependent parameters used by our model, respectively.

## 3.1 Model Definition

The Synchronous Parallel Throughput model (SPT model) is a general performance model that focuses on memory access and synchronization to estimate *parallel throughput*. The SPT model aims to capture the performance characteristics of massively parallel architectures that are designed to maximize throughput and employ a memory hierarchy, such as modern NVIDIA and AMD GPUs and Intel Xeon Phi [38] coprocessors. In this section we present a general description of the SPT model that can be applied to architectures with varying types of memory and synchronization mechanisms.

**Definition 3.1.1** *A* processing unit *is a hardware system that has $P$ processor cores and a finite set, $\Phi$, of distinct* operations *that each processor can perform. All processors run concurrently and can execute any number of operations in $\Phi$ in any order.*

This general definition of a *processing unit* lets us identify, for each particular hardware architecture, the operations that are most relevant. Since we are concerned with high-throughput architectures, we want $\Phi$ to only include operations that most impact throughput (e.g., accessing memory or performing costly synchronizations) and we ignore fine-grain operations such as different

hardware computation operations. In Chapter 4, we identify which operations to include in our model for GPU architectures.

**Definition 3.1.2** *An algorithm is a set of operations in $\Phi$, performed by each processor core, divided into a series of $|\mathcal{K}|$ different* kernels*, $\mathcal{K} = \{\mathcal{K}_1, \mathcal{K}_2, \cdots, \mathcal{K}_{|\mathcal{K}|}\}$. Each kernel is separated by a barrier synchronization, so it runs independently and all operations complete between kernel executions.*

Note that the number of kernels $|\mathcal{K}|$ and the work done by each kernel $\mathcal{K}_i$ depends on the input size, $N$. However, for simplicity of notation, we denote kernel $i$ and $\mathcal{K}_i$ and not $\mathcal{K}_i(N)$. For each processor, kernel, and operation, we define:

**Definition 3.1.3** $\textsc{count}(p, \phi, \mathcal{K}_i)$ *is the number of times processor $p$ performs operation $\phi$ during kernel $\mathcal{K}_i$.*

Since each kernel $\mathcal{K}_i$ depends on the input size $N$, $\textsc{count}(p, \phi, \mathcal{K}_i)$ also depends on $N$, though we omit it to simplify the notation.

To avoid having to analyze each processor individually, we define

**Definition 3.1.4** $\mathcal{A}_{i,\phi}$ *is the* maximum *number of times a single processor performs operation $\phi$ during kernel $\mathcal{K}_i$. Formally:*

$$\mathcal{A}_{i,\phi} = \max_{p=1}^{P}(\textsc{count}(p, \phi, \mathcal{K}_i)),$$

### 3.1.1 Assumptions and Limitations

Since many-core architectures rely on massive parallelism to achieve peak performance, we assume that the scheduler used by our system and the algorithm being analyzed are able to achieve a balanced workload (i.e., for each operation $\phi \in \Phi$, the number of times each processor performs $\phi$ is load-balanced). Formally, for every pair of processors $p$ and $q$, $\phi \in \Phi$ and kernel $\mathcal{K}_i$, we assume that $\textsc{count}(p, \phi, \mathcal{K}_i) = \textsc{count}(q, \phi, \mathcal{K}_i) + O(1)$. While this assumption may limit the accuracy of our model when analyzing certain algorithms, it is a fair assumption for the architectures we are considering for three main reasons.

- Many-core architectures rely on a high degree of parallelism to achieve peak performance and an unbalanced workload can, in the worst case, result in sequential execution, which, for these systems, would result in a performance loss of several orders of magnitude. Thus, to have a remotely competitive algorithm, it must achieve a reasonably balanced workload.
- Massively parallel systems often have groups of processors that make up a unit that performs SIMT operations (described in Section 1.2). Without modeling each individual thread of execution and how it is assigned to each processing unit, we cannot predict how an unbalanced

workload may interplay with this mechanism. Two processor cores performing different operations may result in either concurrent or sequential execution, depending on weather they are part of the same SIMT processing unit or not.

- We are not attempting to accurately model the scheduling and execution of individual threads, rather, we are concerned with the overall throughput achieved by the system.

Following from the assumption that we have a balanced workload,

**Theorem 3.1.5** *For any operation $\phi \in \Phi$, processor $p$, and kernel $\mathcal{K}_i$, it is true that*

$$\textsc{count}(p, \phi, \mathcal{K}_i) + O(1) = \mathcal{A}_{i,\phi}.$$

**Proof** By Definition 3.1.4, for some processor $x$, $\mathcal{A}_{i,\phi} = \textsc{count}(x, \phi, \mathcal{K}_i)$. We assume that, for any processors $p$ and $q$, $\textsc{count}(p, \phi, \mathcal{K}_i) + O(1) = \textsc{count}(q, \phi, \mathcal{K}_i)$. Thus, $\textsc{count}(p, \phi, \mathcal{K}_i) + O(1) = \textsc{count}(x, \phi, \mathcal{K}_i) = \mathcal{A}_{i,\phi}$. ∎

### 3.1.2 Total Runtime

We analyze each kernel individually to determine overall algorithm runtime. We define $t_\phi$ to be the time to perform a single operation $\phi$, we then estimate the *time spent* by kernel $\mathcal{K}_i$ performing operation $\phi$.

**Definition 3.1.6** *$T_{i,\phi}$ is the time spent by kernel $\mathcal{K}_i$ performing operation $\phi$, i.e.,*

$$T_{i,\phi} = \mathcal{A}_{i,\phi} \cdot t_\phi$$

We can then define the total runtime of kernel $\mathcal{K}_i$ (as a function of $N$) as:

**Theorem 3.1.7** *Let $T_i$ be the total runtime of kernel $\mathcal{K}_i$. Since $\mathcal{A}_{i,\phi}$ is the maximum number of times a single processor calls $\phi$,*

$$T_i = \sum_{\phi \in \Phi} (T_{i,\phi} + O(t_\phi)) = \sum_{\phi \in \Phi} (\mathcal{A}_{i,\phi} + O(1)) \cdot t_\phi$$

**Proof** There exists some set of processors ($p_1$, $p_2$, $\cdots$), such that the operations they perform forms the *critical path* of the algorithm, i.e.,

$$T_i = \textsc{count}(p_1, \phi_1, \mathcal{K}_i) \cdot t_{\phi_1} + \textsc{count}(p_2, \phi_2, \mathcal{K}_i) \cdot t_{\phi_2} + \cdots .$$

Since, by Theorem 3.1.5, we assume that the distribution of each operation is balanced among all processors, $\textsc{count}(p, \phi, \mathcal{K}_i)$ and $\textsc{count}(q, \phi, \mathcal{K}_i)$ differ by at most $O(1)$, for any processors $p$ and $q$

and operation $\phi$. Thus, for some processor $p$,

$$\begin{aligned} T_i &= \sum_{\phi \in \Phi} (T_{i,\phi} + O(t_\phi)) = \sum_{\phi \in \Phi} (\textsc{count}(p, \phi, \mathcal{K}_i) + O(1)) \cdot t_\phi \\ &\approx \sum_{\phi \in \Phi} (\mathcal{A}_{i,\phi} + O(1)) \cdot t_\phi \quad \blacksquare \end{aligned}$$

Each kernel operates independently, with barrier synchronizations between each kernel, so we define the runtime of the overall algorithm as

$$\begin{aligned} \mathcal{T} &= \sum_{i=1}^{|\mathcal{K}|} (T_i + L_{sync}) \\ &= (|\mathcal{K}| \cdot L_{sync}) + \sum_{i=1}^{|\mathcal{K}|} \sum_{\phi \in \Phi} (T_{i,\phi} + O(t_\phi)) \\ &= (|\mathcal{K}| \cdot L_{sync}) + \sum_{i=1}^{|\mathcal{K}|} \sum_{\phi \in \Phi} (\mathcal{A}_{i,\phi} + O(1)) \cdot t_\phi, \end{aligned}$$

where $L_{sync}$ is the time to perform a single barrier synchronization. Since $\mathcal{A}_{i,\phi}$ is a function of $N$, we drop the extra $O(1)$ more operations and approximate the runtime as

$$\mathcal{T} = (|\mathcal{K}| \cdot L_{sync}) + \sum_{i=1}^{|\mathcal{K}|} \sum_{\phi \in \Phi} (\mathcal{A}_{i,\phi} \cdot t_\phi).$$

Intuitively, we view our performance estimate of a given algorithm as a $|\mathcal{K}| \times |\Phi|$ matrix, with rows corresponding to kernels and columns to types of operations, as illustrated in Figure 3.1. Cell $i, j$ (row $i$, column $j$) contains $T_{i,j}$, the time spent by kernel $\mathcal{K}_i$ performing operation $j$. The sum of row $i$ is $T_i$, the total time to run kernel $\mathcal{K}_i$. Viewing this matrix, the sum of each *column* corresponds to the total time spent performing a specific operation.

**Definition 3.1.8** *$\mathcal{T}_\phi$ is the total time spent performing operation $\phi$, across the* entire algorithm*, i.e.,*

$$\mathcal{T}_\phi = \sum_{i=1}^{|\mathcal{K}|} T_{i,\phi}.$$

Thus, we estimate the total runtime as

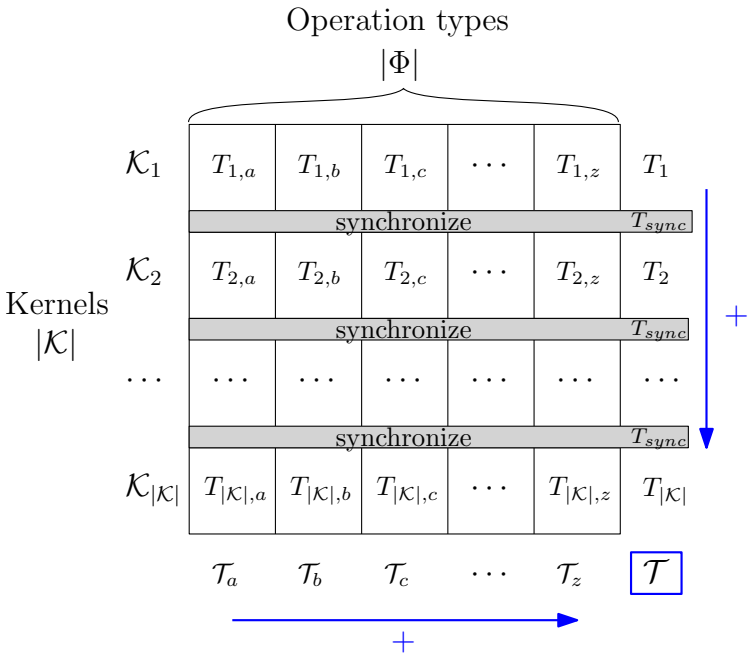


Figure 3.1: Illustration of how we model the runtime, $\mathcal{T}$, of an algorithm with $|\mathcal{K}|$ kernels, on an architecture with a the set of $\Phi$ operation types.

$$\begin{aligned}\mathcal{T} &= (|\mathcal{K}| \cdot L_{sync}) \sum_{\phi \in \Phi} \mathcal{T}_\phi \\ &= (|\mathcal{K}| \cdot L_{sync}) \sum_{\phi \in \Phi} \sum_{i=1}^{|\mathcal{K}|} T_{i,\phi}\end{aligned}$$

Since this is equivalent to our previous approximation for total runtime, the performance estimate can be computed as either the sum of all kernel runtimes (rows in Figure 3.1) or as the sum of time spent performing each operation (columns in Figure 3.1).

### 3.1.3 Time per operation, $t_\phi$

While the model described thus far is general in the sense that any set of operations can be used to define $\Phi$. However, since the SPT model is designed to model high-throughput architectures, we focus on operations that have high latencies, such as accessing slow memory systems or performing expensive synchronizations. The time to perform each of these operations depends on several factors, which we incorporate into our model to get a better estimate of overall performance. Many different architectures mitigate the cost of high latency operations by interleaving operations (e.g., memory accesses) [65, 33], preventing cores from becoming idle while waiting for the operation to

complete (e.g., memory to be accessed). We focus on two common methods that parallel architectures use to mitigate the cost of high-latency operations: *oversubscription* and *instruction-level parallelism (ILP)*.

#### Oversubscription

Oversubscription (defined by Iancu et al. [36]) refers to scheduling multiple threads per physical core. Formally, we say that the oversubscription of a executing program is the number of threads running per core (i.e., $\frac{\text{threads running}}{P}$). With fast context switching, threads can be switched out while they are waiting for operations to complete, allowing other threads to be scheduled and preventing cores from sitting idle. Since many-core architectures often use oversubscription to achieve peak performance, we incorporate the impact of this into our performance estimation. We define $\mathcal{M}_{i,o}$ as the oversubscription achieved by kernel $\mathcal{K}_i$, i.e., the number of threads running per physical core, during $\mathcal{K}_i$. For example, on a system with $P$ processors, kernel $\mathcal{K}_i$ runs $\mathcal{M}_{i,o} \cdot P$ threads). The limiting factor of $\mathcal{M}_{i,o}$ is frequently resource utilization (e.g., memory usage per thread). In Chapter 4, we discuss how to compute $\mathcal{M}_{i,o}$ on our GPU hardware platforms. When analyzing the performance of a single kernel, we simplify our notation by omitting the kernel index subscript, so kernel $K$ has oversubscription of $\mathcal{M}_o$ (i.e., on a system with $P$ processors, kernel $K$ runes $\mathcal{M}_o \cdot P$ threads.

#### Instruction-level parallelism

Instruction-level parallelism (ILP) allows a single thread to hide latency by interleaving *independent* operations. For example, a thread can issue a memory request and continue performing meaningful computation while the memory is being fetched. We define $\mathcal{M}_{i,I}$ to be the average instruction-level parallelism (ILP) of kernel $\mathcal{K}_i$, i.e., the average number of consecutive independent operations that a single thread executes during $\mathcal{K}_i$.

### 3.1.4 Multiplicity

Both oversubscription and ILP provide the benefit of hiding latency of performing certain operations (e.g., memory accesses), thus increasing the number of instructions being executed on a core and increasing overall throughput. For simplicity of our model, we combine these terms and refer to their product as *multiplicity*[1], $\mathcal{M}_i$, where $\mathcal{M} = \mathcal{M}_{i,o} \cdot \mathcal{M}_{i,I}$.

Recall that we compute the time spent in kernel $\mathcal{K}_i$ performing operation $\phi$ as

$$T_{i,\phi} = \mathcal{A}_{i,\phi} \cdot t_\phi$$

[1]Koike and Sadakane [47] use the term multiplicity to refer to what we call oversubscription. However, since we observe that oversubscription and ILP have very similar practical implications, we refer to their combined product as multiplicity.

Where $t_\phi$ is the time to perform one operation of type $\phi$. We incorporate the impact of multiplicity into this parameter by defining the hardware-specific maximum latency and peak bandwidth of performing each operation, $\phi \in \Phi$:

**Definition 3.1.9** *$L_\phi$ is the hardware-specific maximum latency of performing one operation $\phi$, i.e., when $\mathcal{M} = 1$, $t_\phi = L_\phi$ is the time to perform one operation $\phi$.*

**Definition 3.1.10** *$\mathcal{B}_\phi$ is the hardware-specific peak bandwidth of performing the operation $\phi$, measured as the peak number of times that operation $\phi$ can be performed per unit time, per processor.*

By increasing $\mathcal{M}$, we can effectively decrease the time needed to perform each operation. However, once peak bandwidth is reached, we cannot further increase the throughput, so the time to perform one operation $\phi$ is

$$t_\phi = max\left(\frac{1}{\mathcal{B}_\phi}, \left\lceil \frac{L_\phi}{\mathcal{M}_i} \right\rceil\right).$$

Therefore, we measure the time spent by kernel $\mathcal{K}_i$ performing operation $\phi$ as

$$\begin{aligned} T_{i,\phi} &= \mathcal{A}_{i,\phi} \cdot t_\phi \\ &= \mathcal{A}_{i,\phi} \cdot max\left(\frac{1}{\mathcal{B}_\phi}, \left\lceil \frac{L_\phi}{\mathcal{M}_i} \right\rceil\right) \end{aligned}$$

We include the ceiling to measure time in integer units (e.g., clock cycles).

### 3.1.5 Model Simplifications

If, for a given algorithm, the multiplicity achieved remains the same across all kernel calls (i.e., $\mathcal{M}_i = \mathcal{M}_j$ for all $1 \leq i \leq j \leq |\mathcal{K}|$), or if the algorithm is comprised of a single kernel, we can simplify our calculation by using a single multiplicity value, $\mathcal{M}$, and defining the *total* operations across the entire algorithm as

$$\mathcal{A}_\phi = \sum_{i=1}^{|\mathcal{K}|} \mathcal{A}_{i,\phi}$$

This allows us to ignore individual kernels and estimate the total runtime of the algorithm as $\mathcal{T} = |\mathcal{K}| \cdot L_{sync} + \sum_{\phi \in \Phi}(\mathcal{T}_\phi)$, where $\mathcal{T}_\phi = \mathcal{A}_\phi \cdot max\left(\frac{1}{\mathcal{B}_\phi}, \left\lceil \frac{L_\phi}{\mathcal{M}} \right\rceil\right)$.

## 3.2 Comparison with existing models

In this section, we discuss how the SPT model compares to other, well-known existing models. We first discuss well-known general models and highlight similarities and ways that analysis using

the SPT model can use previous work. We then compare the SPT model with existing models designed for many-core architectures such as GPUs. A broad overview of these models is provided in Section 2.1, though we provide more detail here, as needed.

### 3.2.1 The PRAM Model

A central feature of the SPT model is that we estimate overall performance by considering the throughput of each type of *operation*, such as accesses to different levels of the memory hierarchy. Thus, if we consider operation $\phi$ to be accessing a memory system that allows random access (i.e., any location can be accessed in unit time), we can use the PRAM model to compute $\mathcal{A}_{i,op}$, the number of accesses performed during kernel $\mathcal{K}_i$. This lets us use the extensive prior work with the PRAM model, such as proven lower bounds and optimality. In Chapter 4, we use PRAM when modeling the fast *shared memory* and *register* operations of our GPU platforms.

### 3.2.2 The PEM Model

The External Memory [4] (EM) and Parallel External Memory [9] (PEM) models, discussed in Section 2.1, model the complexity of an algorithm as the number of sequential and parallel *block* accesses to memory, respectively. This type of block memory access is common in slow, high-latency memory systems, so it a useful model when modeling algorithms that are bound by slow memory accesses. The parallel architectures that we focus on for the SPT model often rely on such memory systems, so the we can use PEM to analyze algorithms that utilize these types of memory. For example, recall from Section 1.2 that global memory requires that consecutive elements be read by all threads of a warp to achieve peak performance (i.e., for an access to be *coalesced*). If threads within a warp read distant elements, multiple global memory accesses are required, reducing memory throughput. This is equivalent to the "blocked" access pattern of the PEM model, where $w$ consecutive elements must be read in a block to achieve peak performance. Thus, in Chapter 4, we use the PEM model to count global memory accesses when analyzing algorithm performance on our GPU platforms.

### 3.2.3 The BSP Model

The SPT model is similar to the well BSP model, presented by Valiant [78] and discussed in Section 2.1, if we equate kernels to *supersteps*. Recall from Section 2.1 that the BSP model considers an algorithm as a series of supersteps, each of which is composed of computation, communication, and synchronization steps. Similarly, the SPT model views an algorithm as a series of kernels, with each kernel performing a series of operations that facilitate computation and communication (i.e., memory accesses), and with a barrier synchronization between kernels. Figure 3.2 illustrates how these two models consider algorithms as a series of steps (supersteps or kernels) with synchronization

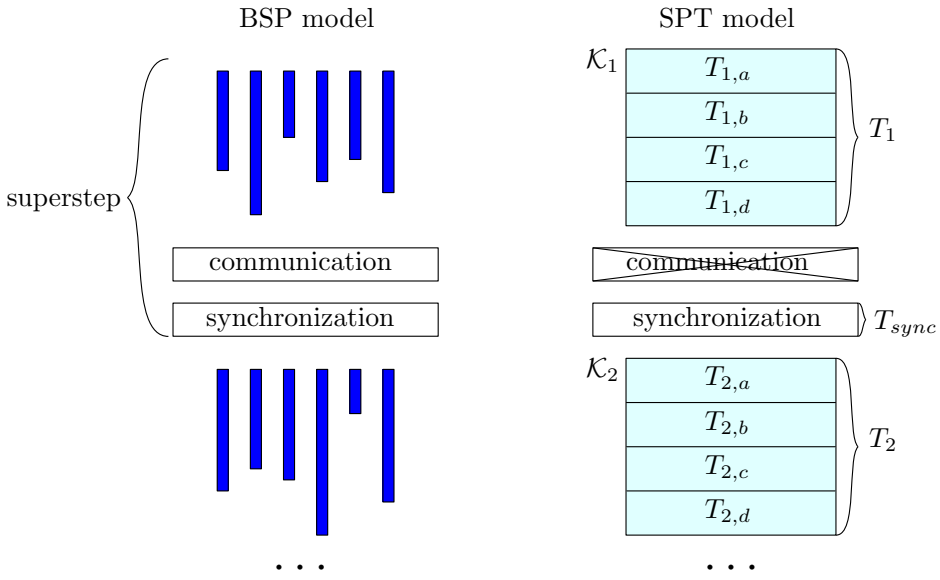


Figure 3.2: Illustration of the BSP model [78] and how our SPT model differs. In the SPT model, we estimate the runtime of each kernel by the time needed to access each type of memory (memories $a$, $b$, $c$, and $d$ in this example).

between each step. There are, however, several key differences that make the SPT model more suitable to parallel architectures such as GPUs. The first difference relies on our assumption of load-balancing (Theorem 3.1.5), which lets us avoid computing the critical path and ignore the complexities associated with thread scheduling. This also lets the SPT model estimate throughput by simply counting operations and the cost per operation, while, for the BSP model, we must calculate the critical path of each superstep by considering the execution time of each processor. Our simple assumption lets us completely avoid analyzing individual processors, which is essential for the massively parallel architectures that we focus on. Finally, the SPT model incorporates the details of any type of operation that we wish to model. This is especially useful when modeling access patterns of memory systems, such as those of GPU platforms, as we see in Chapter 4.

### 3.2.4 Prior GPU Performance Models

Section 2.1.3, presents an overview existing performance models for modern GPU architectures. We now discuss the details of several of these models and compare them with the SPT model. We focus our discussion on general performance models rather than works that use benchmarks or autotuning techniques to design specific algorithms. We note that, while the SPT model is not specific to GPU architectures, it is designed with these types of systems in mind, and, as we show throughout this dissertation, it is well-suited to model GPUs.

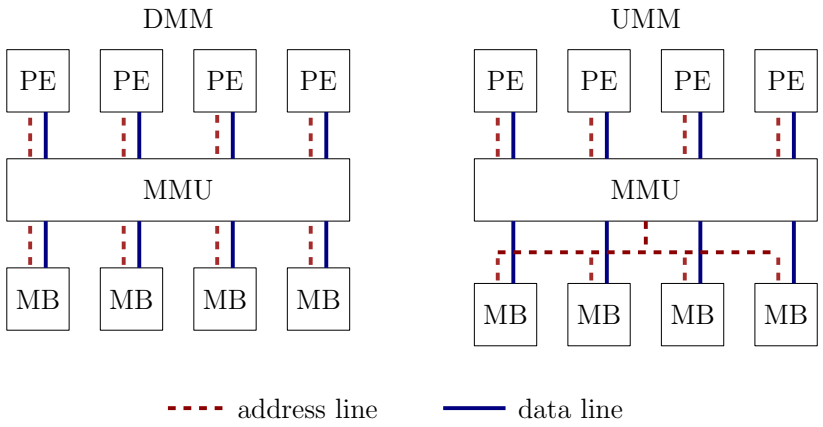


Figure 3.3: Illustration of the DMM and UMM models developed by Nakano [61]. The interconnection of the address line between the memory management unit (MMU) and memory banks (MBs) results in different optimal memory access patterns.

**Discrete Memory Model, Universal Memory Model, and Hierarchical Memory Model**

The Discrete Memory Model [61] (DMM) is a simple model designed to capture the essential features of GPU shared memory while the Universal Memory Model [61] (UMM) focuses on the GPU global memory system. Each of these models considers a series of Processing Elements (PEs), a set of Memory Banks (MBs), and a single Memory Management Unit (MMU) that connects them. Figure 3.3 illustrates how the DMM and UMM models view the interconnection between MBs and PEs. The address and data lines between MBs, MMU, and PEs dictate the elements that can be accessed in parallel by the PEs. In the DMM, each MB has its own address and data lines to the MMU, so a single access can be performed on each MB at a time, or else accesses are serialized (i.e., shared memory bank conflicts occur). In the UMM, however, all MBs share an address line to the MMU, so the same address must be accessed from each MB at once (i.e., a coalesced global memory access). The authors present algorithms that perform 2-d array transposition efficiently for the DMM and UMM model [61]. The Hierarchical Memory Model [62] (HMM), illustrated in Figure 3.4, combines the DMM and UMM into a single model of parallel computation that captures the features of both shared and global memory systems of modern GPUs. It models GPU architectures as a single instance of UMM and a series of DMMs, corresponding to the different streaming multiprocessors on modern GPUs.

The SPT model is applied to architectures by selecting a series of operations and estimating overall performance based on the throughput of each operation. However, we do not specify the mechanism used to count the number of each type of operation performed. On modern GPUs, accessing global memory or shared memory are operations that frequently cause performance bottlenecks and should therefore by included in the SPT model when applied to GPUs. Thus, existing models of architectures with specific memory systems or optimal access patterns, such as DMM,

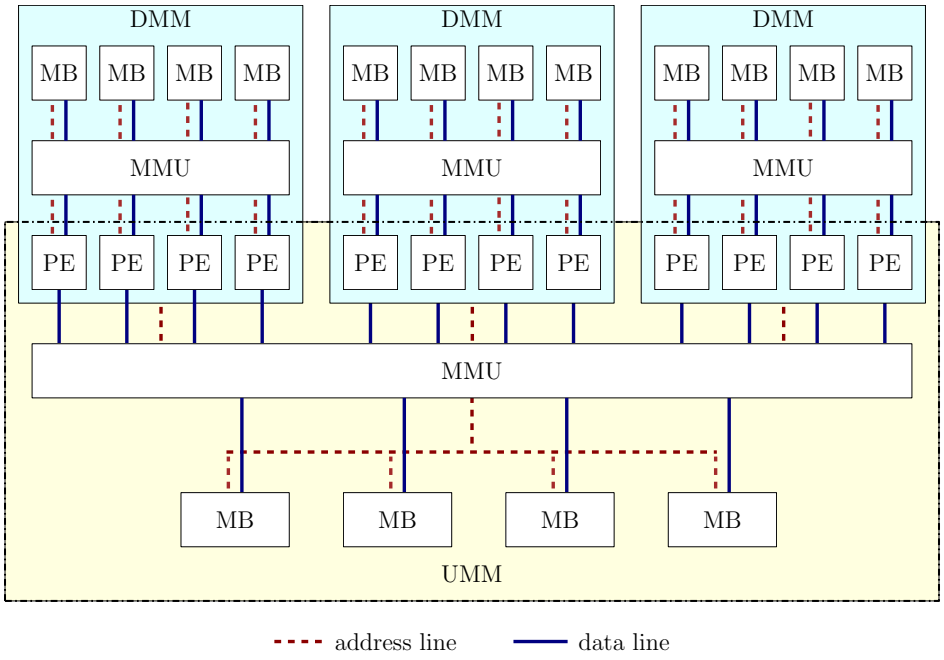


Figure 3.4: Illustration of the HMM model that combines several DMMs (shared memory) with a UMM (global memory) into a hierarchy.

UMM, and HMM, are a useful resource that the SPT model can use when analyzing algorithms. We note that, while the DMM, UMM, and HMM models focus on GPU memory systems, in this work we opt to analyze memory accesses using the simpler PRAM and PEM models. In addition to the relative simplicity of the PRAM and PEM models, they allow us to more easily count shared memory and global memory accesses across the entire system, while the DMM, UMM, and HMM only consider accesses performed by a single *warp* on the GPU, and extending it to the entire system adds complexity.

### Model by Kothapalli et al.

The model presented by Kothapalli et al. [48] was one of the first GPU models to attempt to predict overall performance. Similar to our SPT model, they model GPU architectures using a variation of the BSP model [78] by equating individual kernels as supersteps. Within each kernel, they analyze performance by considering only accesses to global memory and shared memory. They provide their own method of analyzing global memory access patterns and use the Queue-Read, Queue-Write (QRQW) PRAM [27] model to incorporate the impacts of shared memory bank conflicts. There are, however, several performance factors that they do not consider and that, in this work, we show can have a significant impact on GPU performance: 1) time spent performing register operations, 2) the peak bandwidth of each operation, and 3) instruction-level parallelism (ILP). Thus, the model presented by Kothapalli et al. [48] is a specific instantiation of the SPT model:

the SPT model instantiated on a GPU architecture with $\Phi = \{\phi_1 =$ access global memory, $\phi_2 =$ access shared memory$\}$, with $\mathcal{M}_{i,I} = 1$ for all kernels and $\mathcal{B}_{\phi_1} = \mathcal{B}_{\phi_2} = 0$. Using the memory models used by Kothapalli et al. [48] to determine $\mathcal{A}_{i,\phi_1}$ and $\mathcal{A}_{i,\phi_2}$ for each kernel $\mathcal{K}_i$ of a given algorithm then results in their overall model. Note that this corresponds to the SUM variation of their model, as the SPT model computes $\mathcal{T}$ as the sum of each $\mathcal{T}_\phi$.

### Threaded Many-core Model

The Threaded Many-core Model [54] (TMM) considers global memory latency and the impact of multiplicity, along with computation. The TMM has been used to analyze a range of fundamental algorithms [55, 56]: string matching, fast Fourier transform, merge sort, list ranking, and all pairs shortest path. The TMM has been shown to be useful in determining the asymptotic complexity of algorithms on GPUs. As with our SPT model, the TMM incorporates the impact of multiplicity (they refer to as $\mathcal{T}$) on memory access time. However, they ignore the impact of bandwidth, implying that one can continue to reduce the cost of memory access time until you reach the hardware limited number of threads. In Chapter 4, we show that this is not the case and, in fact, one can quickly be limited by bandwidth, even with a relatively small multiplicity. Furthermore, the TMM does not include the cost of synchronization. Rather, they measure algorithm performance by looking at the critical path of execution through the program.

### The AGPU Model

The AGPU model, presented by Koike and Sadakane [47] is a thorough GPU-specific performance model that looks at several details of the GPU compute and memory hierarchies. They consider the GPU divided into a series of *multiprocessors*, each with $b$ cores and having $M$ shared memory. They assume a simplification from the GPU hardware, in that they divide the resources of the GPU into warp-size multiprocessors and do not consider multi-warp thread-blocks. Nevertheless, they present a detailed model that takes into account global and shared memory access patterns and, for a given algorithm, present two asymptotic complexity metrics: the time spent accessing global memory and the time spent accessing shared memory. They also provide a discussion of *multiplicity*, which we refer to as oversubscription, though they do not relate it directly to the asymptotic performance of the algorithm. As with the model presented by Kothapalli et al. [48], AGPU can be seen as a specific instantiation of the SPT model that only considers global memory and shared memory as operations. Additionally, the AGPU model only considers the asymptotic performance of algorithms, so, unlike the SPT model, it cannot be used to estimate runtime.

While the AGPU model provides a clear and detailed method of analyzing the asymptotic performance of algorithms on GPUs, it neglects to consider several key factors that we show to have a significant impact on practical performance: 1) time spent performing operations on registers, 2) the interplay between latency, bandwidth, and multiplicity, and 3) synchronization cost. In

particular, without sufficient multiplicity, practical performance quickly degrades, as we show in Chapter 4. Furthermore, in Chapter 7 we consider the sorting algorithm designed with AGPU [47] and show that insufficient multiplicity causes performance loss. We present a series of improvements to their algorithm and demonstrate a significant performance increase.

Table 3.1: Parameters defined by the hardware system.

| Parameter | Definition | Defined In |
|---|---|---|
| $\mathcal{U}$ | A hardware processing unit | Definition 3.1.1 |
| $P$ | # of cores in $\mathcal{U}$ | Definition 3.1.1 |
| $\Phi$ | Set of operations the system can performs | Definition 3.1.1 |
| $L_\phi$ | Time to perform a single call to operation $\phi$ | Section 3.1.2 |
| $l_\phi$ | Max. latency of $\phi$ (i.e., when $\mathcal{M} = 1$) | Definition 3.1.9 |
| $\mathcal{B}_\phi$ | Peak bandwidth of $\phi$ (i.e., max. number of $\phi$ per cycle) | Definition 3.1.10 |

Table 3.2: Parameters defined by the algorithm.

| Parameter | Definition | Defined In |
|---|---|---|
| $\lvert\mathcal{K}\rvert$ | # of kernels in the algorithm | Definition 3.1.2 |
| $\textsc{count}(p, \phi, K_i)$ | # of times processor $p$ performs operation $\phi$ in kernel $K_i$ | Definition 3.1.3 |
| $\mathcal{A}_{i,\phi}$ | $\mathcal{A}_{i,\phi} = \max_{p=1}^{P}(\textsc{count}(p, \phi, K_i))$ | Definition 3.1.4 |
| $\mathcal{M}_{i,o}$ | Oversubscription of kernel $\mathcal{K}_i$ | Section 3.1.3 |
| $\mathcal{M}_{i,I}$ | Instruction-level parallelism (ILP) of kernel $\mathcal{K}_i$ | Section 3.1.3 |
| $\mathcal{M}_i$ | Multiplicity of kernel $\mathcal{K}_i$ (i.e., $\mathcal{M}_{i,o} \cdot \mathcal{M}_{i,I}$) | Section 3.1.4 |
| $T_{i,\phi}$ | Time spent in kernel $\mathcal{K}_i$ performing $\phi$ | Definition 3.1.6 |
| $T_i$ | Total runtime of kernel $\mathcal{K}_i$ | Theorem 3.1.7 |
| $\mathcal{T}$ | Total runtime of the algorithm | Section 3.1.2 |

# CHAPTER 4
# INSTANTIATING THE MODEL

In this chapter, we consider the features specific to modern GPU architectures to instantiate the SPT model. We first identify operations that we consider the most significant when modeling overall GPU performance. As discussed in Section 1.2, GPU architecture are many-core parallel architectures designed to achieve high computational throughput. Thus, we focus on high latency operations such as accessing slow memory systems or performing synchronization. In particular, for our GPU platforms, we define the set of operations as $\Phi = \{g, s, r, y\}$, where

- $g$ is accessing global memory,
- $s$ is accessing shared memory,
- $r$ is performing an operation in registers, and
- $y$ is performing intra-thread-block synchronization (i.e., SYNCTHREADS).

Each of these operations has a different corresponding latency ($L$) and bandwidth ($\mathcal{B}$), as well as other performance considerations such as optimal memory access patterns. Thus, we consider each operation individually, identifying the best way to model it and developing microbenchmarks to measure the associated hardware parameters ($L$ and $\mathcal{B}$). We measure these parameters for each of our hardware platforms, with the results included in Table 4.2, at the end of this Chapter. For simplicity, in this Chapter we discuss the model in terms of only a single kernel of execution, and so omit the kernel index from parameter subscripts (e.g., $\mathcal{A}_g$ is the number of global memory accesses and $\mathcal{M}$ is the multiplicity).

## 4.1 Methodology

We perform all experiments using three hardware platforms, each with a different modern graphics card. All computations are performed on the graphics cards and no attempt is made to use CPU compute resources. Furthermore, execution times are measured as time spent computing on the GPU, while time to transfer data between the CPU and GPU is not included, as is customary in these types of experiments. The specifications of each of our GPUs, along with the versions of GCC and CUDA used are listed in Table 4.1. All experiments are compiled with the -O3 optimization flag. Performance metrics such as bank conflicts are obtained via the NVPROF profiling tool [66], included in the CUDA toolkit. Since running the NVPROF tool impacts performance, execution time is measured on separate runs using the *cudaEvent* timer that is available with CUDA. Each experiment is repeated ten times, and we report on mean values, showing min-max error bars when non-negligible, unless otherwise mentioned.

Table 4.1: Details of our three GPU hardware platforms.

| Platform name | ALGOPARC | GIBSON | UHHPC |
|---|---|---|---|
| GPU Model | Quadro M4000 | GeForce GTX 770 | Tesla K40m |
| Generation | Maxwell GM204 | Kepler GK104 | Kepler GK110B |
| Global Memory | 8GB | 4 GB | 12 GB |
| Total Cores | 1664 | 1536 | 2880 |
| Clock Rate | 780 MHz | 1046 MHz | 745 MHz |
| Total shared memory | 1248 KiB | 512 KiB | 960 KiB |
| Total 32-bit registers | 832K | 512K | 960K |
| "Peak" Global memory bandwidth | 192 GB/s | 224.3 GB/s | 288 GB/s |
| "Peak" Shared memory bandwidth | 1.29 TB/s | 1.07 TB/s | 1.35 TB/s |
| GCC version | 5.4 | 5.4 | 4.4 |
| CUDA version | 9.1 | 7.5 | 7.5 |

## 4.2 Global memory accesses

Recall from Section 3.2.2 that we view global memory access patterns as *blocked* access, where $w$ consecutive elements are read per memory access. Therefore, to determine the asymptotic number of global memory accesses, $\mathcal{A}_g$, performed by a kernel, we use the parallel external memory model (PEM), discussed in Section 2.1. We obtain an asymptotic estimate of $\mathcal{A}_g$ using the PEM model with block size $B = w$. Using the PEM model lets us use existing analytical techniques to determine the asymptotic number of global memory accesses performed by a given kernel. Additionally, the body of work using the PEM models, including complexity bounds and algorithm design techniques are applicable to our analysis and algorithm design.

### 4.2.1 Measuring $L_g$ and $\mathcal{B}_g$

The SPT model incorporates operation latency and bandwidth when estimating the runtime of an algorithm on a particular architecture. Thus, we use microbenchmarks to measure the latency and bandwidth of global memory on each of our GPU platforms. We measure the latency as the average time, in clock cycles, to perform one (coalesced) global memory access, when $\mathcal{M} = 1$. Bandwidth, however, is the peak number of elements that can be accessed when bandwidth bound (i.e., when $\mathcal{M}$ is large). Thus, we run benchmarks that perform global memory accesses (and as little additional work as possible), for varying $\mathcal{M}$ values.

The simplest experiment to measure the latency ($L_g$) and bandwidth ($\mathcal{B}_g$) of global memory is to have a single kernel that copies arrays. For this microbenchmark, *each thread* copies $N$ elements from one array in global memory to another (also in global memory). Thus, as we increase oversubscription ($\mathcal{M}_o$), we increase the amount of work being done. We perform all accesses in a

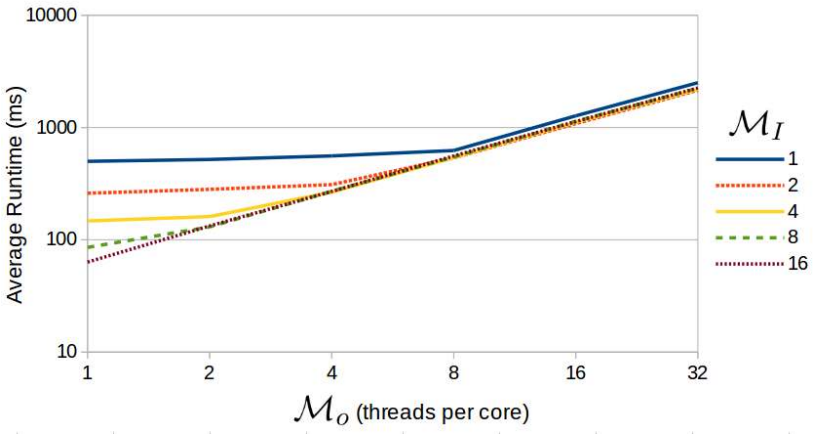


Figure 4.1: Impact of oversubscription ($\mathcal{M}_o$) and ILP ($\mathcal{M}_I$) on global memory access time. Results on ALGOPARC with $N = 2^{16}$ integers per thread. Once $\mathcal{M} = 8$, peak global memory bandwidth is reached.

coalesced manner, resulting in a total of $2N$ memory accesses per thread, so

$$\mathcal{A}_g = \frac{2 \cdot N \cdot \mathcal{M}_o}{P}.$$

We expect the runtime of this microbenchmark, in clocks cycles, to be

$$\mathcal{T} = L_{sync} + \frac{2N\mathcal{M}_o}{P} max\left(\frac{1}{\mathcal{B}_g}, \left\lceil \frac{L_g}{\mathcal{M}} \right\rceil\right).$$

We manually control oversubscription ($\mathcal{M}_o$) and ILP ($\mathcal{M}_I$), allowing us determine the impact of each on execution time. We vary $\mathcal{M}_o$ by controlling the number of thread-blocks the kernel uses. To increase $\mathcal{M}_I$, we have threads load $\mathcal{M}_I$ elements into registers before writing them into the copy array.

Figure 4.1 displays the average runtime of this microbenchmark on ALGOPARC when each thread copies $2^{16}$ integer elements (32-bits each) from one global memory array into another, for varying $\mathcal{M}_o$ and $\mathcal{M}_I$ values. We note that, since the number of total accesses grows with $\mathcal{M}_o$, linear growth implies constant throughput. Thus, the inflection point where each curve ceases to be flat is the point where $\mathcal{B}_g$ is reached. Figure 4.1 clearly indicates that this inflection point occurs when $\mathcal{M}_o \cdot \mathcal{M}_I = 8$. Thus, on ALGOPARC, when $\mathcal{M} = 8$, multiplicity ceases to hide latency, implying that $L_g = \frac{8}{\mathcal{B}_g}$. We measure $\mathcal{B}_g$ using the maximum throughput achieved and we repeat this experiment for our other hardware platforms (see Table 4.2 for the resulting parameter values).

## 4.3 Shared memory accesses

As discussed in Section 1.2, shared memory requires a unique access pattern to avoid bank conflicts and achieve peak throughput. Several previous works have presented models of GPUs that include analysis of this access pattern [61, 62, 47]. While these models can help us estimate the number of shared memory accesses performed by algorithms with predictable access patterns, many algorithms have data-dependent access patterns, resulting in an unknown number of bank conflicts. Thus, we model shared memory accesses separately from bank conflicts and combine them to estimate total accesses. If we ignore bank conflicts, any thread is able to access any location of shared memory in unit time, allowing us to use the simple CREW PRAM model to determine $\mathcal{A}_s$. This simplifies our analysis of shared memory and allows us to employ various other techniques to estimate bank conflicts. Thus, we use an additional parameter, $\beta$, captures the performance implication of shared memory bank conflicts.

We define $\beta$ to be the *average* number of sequential memory accesses per parallel shared memory request, that is, one plus the average number of bank conflicts (i.e., $1 \leq \beta \leq w$). We estimate the total number of shared memory accesses as $\beta \cdot \mathcal{A}_s$. Note that each kernel $\mathcal{K}_i$ may have a different shared memory access pattern and therefore its own $\beta_i$ value. Since we estimate bank conflicts separately from memory accesses, we can compute $\beta$ in a variety of ways, depending on the kernel begin analyzed. For deterministic algorithms, we can analytically compute $\beta$ for each kernel by either analyzing the access pattern directly or using the Discrete Memory Machine (DMM) model [61] (discussed in Section 3.2.4). However, for arbitrary data-dependent access patterns, it remains an open problem to determine the expected number of bank conflicts. For the data-dependent algorithms that we analyze in this work, we compute $\beta$ empirically using the NSIGHT performance profiler [66]. We note that, since $\beta$ is only dependent on the memory access pattern (i.e., the algorithm and input data), we can measure $\beta$ once and use the result to estimate algorithm performance on a range of different hardware platforms. We leave probabilistic analysis of bank conflicts as an interesting open problem for future work.

### 4.3.1 Measuring $L_s$ and $\mathcal{B}_s$

Since we estimate bank conflicts separately from shared memory accesses, we create a microbenchmark that incurs no bank conflicts to measure shared memory latency, $L_s$, and peak bandwidth, $\mathcal{B}_s$. To emphasize the contribution of shared memory, we perform $X$ accesses in shared memory for each element. Thus, the execution time of this microbenchmark is expected to be

$$T = \frac{N\mathcal{M}_o X}{P} max\left(\frac{1}{\mathcal{B}_s}, \left\lceil \frac{L_s}{\mathcal{M}} \right\rceil\right).$$

Figure 4.2 shows the results of this microbenchmark on ALGOPARC, with $N = 2^{15}$ and $X = 100$,

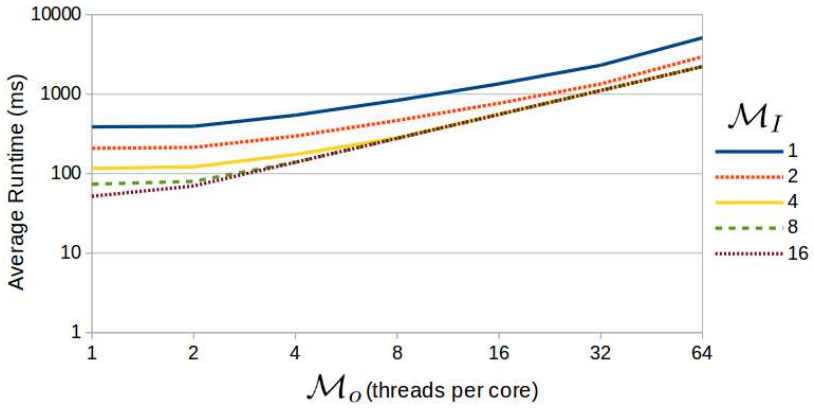


Figure 4.2: Impact of oversubscription ($\mathcal{M}_o$) and ILP ($\mathcal{M}_I$) on shared memory access time on ALGOPARC with $N = 2^{15}$ elements per thread and $X = 100$.

for varying $\mathcal{M}_o$ and $\mathcal{M}_p$. As with global memory, we see that, once $\mathcal{M}$ becomes large enough, we become bound by peak bandwidth ($\mathcal{B}_s$). We note that, when ILP ($\mathcal{M}_I$) is 1 or 2, we never reach $B_s$. This is explained by the overhead associated with the benchmark itself. Since shared memory accesses are so fast, additional computation such as loop operations impact performance. However, for $\mathcal{M}_I > 2$, we clearly see that $\mathcal{B}_s$ is reached when $16 < \mathcal{M}_o \cdot \mathcal{M}_p < 24$. Using this we estimate shared memory latency to be $L_s = \frac{20}{\mathcal{B}_s}$ for ALGOPARC.

## 4.4 Register operations

Registers constitute the fastest level of the GPU memory hierarchy, though they are thread-private and the access pattern must be known at compile time. The *__shfl()* hardware instruction allows threads within a warp to access each others registers, however, so we estimate the asymptotic number of register operations performed by a given kernel using the CREW PRAM model, allowing communication between groups of $w$ threads. Thus, the lower bounds and analytical techniques presented in prior work on the PRAM model applies to our analysis of register operations, with the caveats that the pattern of accesses to different registers must be known at compile time (i.e., static) and only threads within a warp share data.

### 4.4.1 Measuring $L_r$ and $\mathcal{B}_r$

We measure the impact of multiplicity on register computation using a microbenchmark similar to that for shared memory. We vary $\mathcal{M}_o$ and $\mathcal{M}_I$ while performing $X$ register operations on each of $N$ elements. As with the results of our shared memory benchmark, the overhead of the experiment itself makes it difficult to isolate the cost of individual operations, so, to get a more accurate estimate, we also increase $N$ with $\mathcal{M}_I$, thereby having each thread work on $N \cdot \mathcal{M}_I$ elements. This

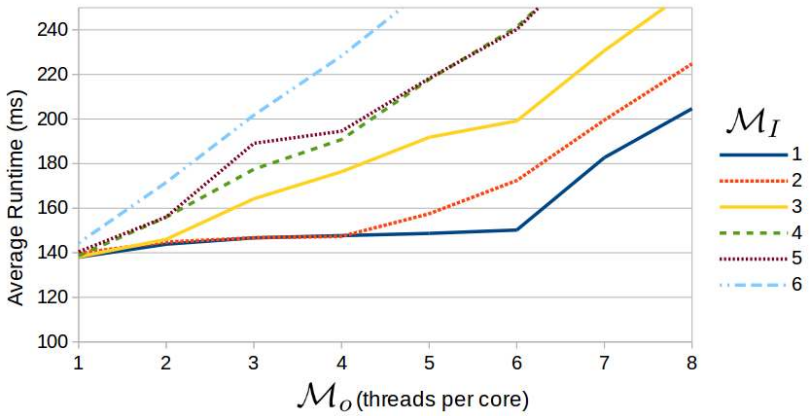


Figure 4.3: Impact of oversubscription ($\mathcal{M}_o$) and ILP ($\mathcal{M}_I$) on time to perform register operations on ALGOPARC with $N = 2^{15} \cdot \mathcal{M}_I$ elements per thread and $X = 500$. Since memory accesses increase with $\mathcal{M}$, multiplicity continues to hide latency as long as runtime remains flat.

helps ensure that the experimental overheads remain constant regardless of $\mathcal{M}$. We estimate the runtime of this benchmark as

$$\mathcal{T} = \frac{NX\mathcal{M}}{P} \cdot max\left(\frac{1}{\mathcal{B}_r}, \left\lceil \frac{L_r}{\mathcal{M}} \right\rceil\right).$$

Figure 4.3 plots the runtime of this benchmark on ALGOPARC with $X = 500$ and $N = 2^{15} \cdot \mathcal{M}_I$, using 32-bit integer elements. Since we increase work with $\mathcal{M}$, we expect runtime to remain flat while multiplicity continues to hide latency. However, once runtime starts increasing, we have reached peak bandwidth and runtime is limited by $\mathcal{B}_r$. Using these results, we measure $\mathcal{B}_r = 0.86$ on ALGOPARC. Recall that $\mathcal{B}_r$ is the number of operations per clock cycle, per core. Thus, if $\mathcal{B}_r = 1$, a register operation can be performed every cycle, which we expect to be the maximum throughput for register operations. Our measured $\mathcal{B}_r$ is 14% less than this, which we attribute to noise resulting from the microbenchmark. We use the estimate $\mathcal{B}_r = 1$ for all modeling and experiments hereafter. Thus, we can then estimate $L_r = \frac{6}{\mathcal{B}_r} = 6$ on ALGOPARC.

## 4.5 Thread-block synchronization

As discussed in Section 1.2, the number of threads in a thread-block (TB) is variable, and we defined this value as $b$. All threads in a TB have access to the same shared memory partition and run on the same SM. Therefore, the explicit intra-TB synchronization, SYNCTHREADS, is a valuable tool for communication and fine-grain synchronization. While the SPT model does not consider the impact it may have on scheduling, the SYNCTHREADS operation has a fixed cost, so we incorporate it into the model instantiation.

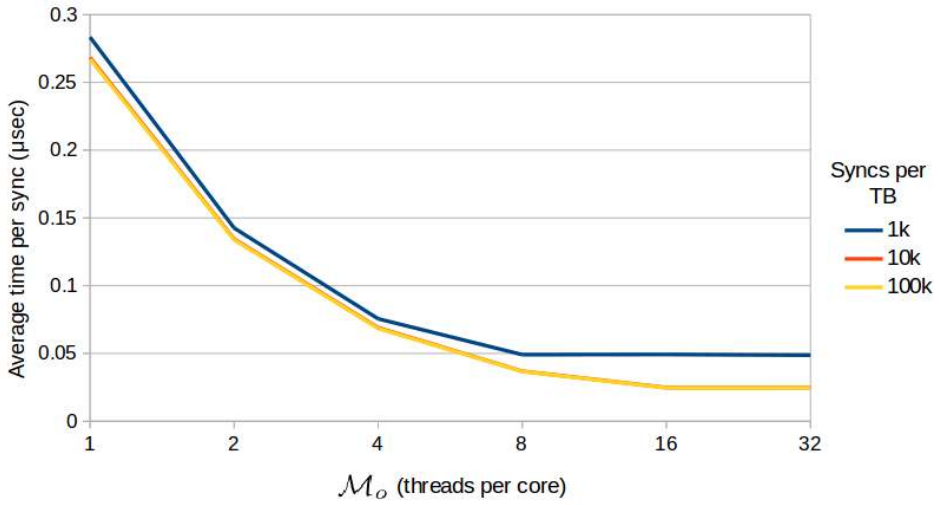


Figure 4.4: Average time per SYNCTHREADS operation, when performing a fixed $N = 10^5$ register additions per block, for different values of $\mathcal{M}_o$, on ALGOPARC.

### 4.5.1 Measuring $L_y$ and $\mathcal{B}_y$

We measure the time to perform a SYNCTHREADS operation using microbenchmarks that repeatedly call SYNCTHREADS while performing a small amount of additional work in registers (needed to prevent compiler optimization from eliminating the calls to SYNCTHREADS). Figure 4.4 plots the average time per SYNCTHREADS, for varying oversubscription ($\mathcal{M}_o$). We see that, as we increase $\mathcal{M}_o$, the time to perform each SYNCTHREADS is reduced, until a plateau is reached and the time per SYNCTHREADS remains constant. This pattern is similar to the performance profile of memory accesses when varying $\mathcal{M}$.

This allows us to measure $L_y$ and $\mathcal{B}_y$ for each of our hardware platforms. We note that, since the SYNCTHREADS command, by its very nature, cannot be independent, ILP does not apply and $\mathcal{M}_I = 1$. Therefore, we estimate the time required, in clock cycles, to perform $\mathcal{A}_y$ SYNCTHREADS operations to be

$$\mathcal{T}_y = \mathcal{A}_y \cdot max\left(\frac{1}{\mathcal{B}_y}, \frac{L_y}{\mathcal{M}_o}\right),$$

We use this microbenchmark to measure $L_y$ and $\mathcal{B}_y$ for each of our hardware platforms and present the resulting values in Table 4.2.

## 4.6 Barrier synchronization

Between kernel executions, the SPT model assumes that a device-wide barrier synchronization is performed. CUDA provides the CUDADEVICESYNCHRONIZE() operation, which we refer to here-

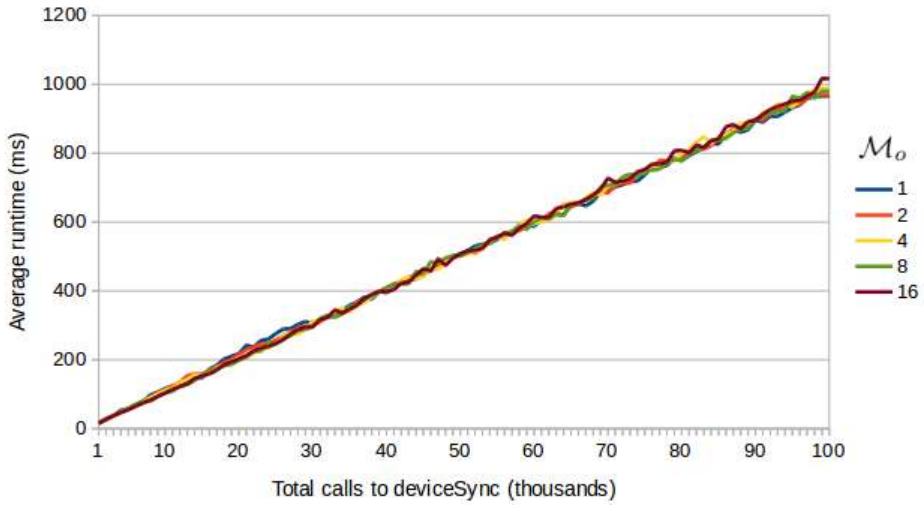


Figure 4.5: Average runtime to perform $N$ additions and a varying number of DEVICESYNC operations, for different values of $\mathcal{M}_o$ and $\mathcal{M}_I$, on ALGOPARC. Each DEVICESYNC has a fixed cost, regardless of other parameters.

after as DEVICESYNC, that explicitly synchronizes all threads on the GPU. It is a blocking command that is called from the host (CPU) and waits until all GPU threads currently running are completed[1]. The DEVICESYNC operation corresponds directly to the synchronization stage of the SPT model. Therefore, for a given algorithm on the GPU, we can determine $|\mathcal{K}|$ by the number of DEVICESYNC operations it performs.

We measure the time to perform a DEVICESYNC, $L_{sync}$, with a microbenchmark that varies the number of DEVICESYNC calls while having each thread perform a fixed amount of work (additions in global memory). Figure 4.5 plots the average runtime of this microbenchmark, for several different values of $\mathcal{M}_o$. Results indicate that each DEVICESYNC operation has a fixed cost that is independent of other parameters, which corresponds to the way we model barrier synchronization in the SPT model. Figure 4.6 plots the average time spent per DEVICESYNC operation and shows that, as we increase the total DEVICESYNC calls, the time converges to a constant value, which we estimate to be the time to perform DEVICESYNC (i.e., $L_{sync}$). We run these microbenchmarks on each of our hardware platforms and include the measured $L_{sync}$ values in Table 4.2.

## 4.7 Computing multiplicity

Recall from Section 3.1.4 that we compute multiplicity, $\mathcal{M}$, by determining oversubscription ($\mathcal{M}_o$) and ILP ($\mathcal{M}_I$). Specifically, we say that $\mathcal{M} = \mathcal{M}_o \cdot \mathcal{M}_I$. On GPUs two main factors limit the

[1]Since CUDA version 6, CUDADEVICESYNCHRONIZE() can be called from the GPU, which causes the calling thread to wait for all kernels that it spawned to finish. However, there are many limitations to this "dynamic parallelism" [65], so we do not consider it in our model.

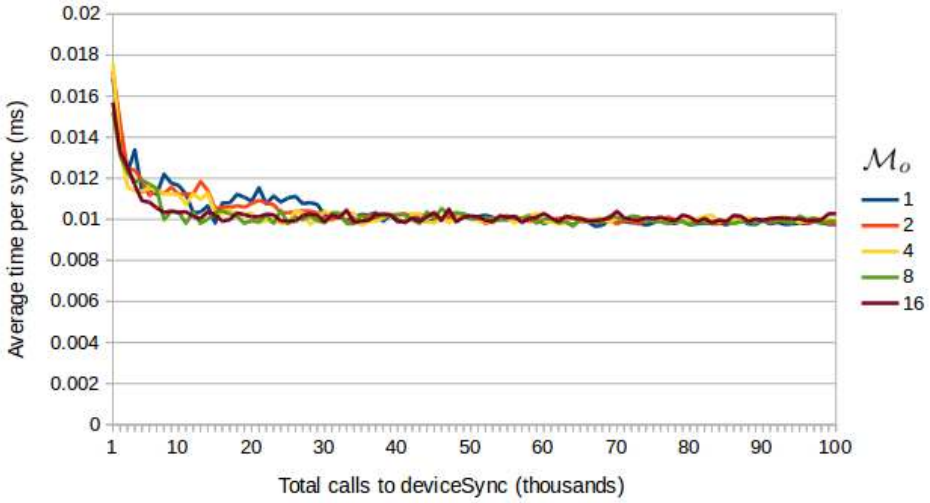


Figure 4.6: Average time spent per DEVICESYNC, for varying total number of DEVICESYNC operations, and for different values of $\mathcal{M}_o$, on ALGOPARC. As we increase the number of DEVICESYNC operations, the time per sync converges to a constant, which we estimate to be $L_{sync}$.

oversubscription, $\mathcal{M}_o$, of a kernel:

1. shared memory usage – each SM has a limited amount of shared memory, so if each TB uses too much, it limits the number of TBs that can run simultaneously. Specifically, if each TB has $b$ threads and each thread stores $E$ elements in shared memory, oversubscription is limited to:

$$\mathcal{M}_o \leq \frac{M}{b \cdot E \cdot P}$$

   where $M$ and $P$ are the total shared memory and number of compute cores on the entire GPU, respectively.

2. register usage – similar to shared memory, using too many registers per thread can limit oversubscription. If each thread uses $r$ registers, oversubscription is limited to:

$$\mathcal{M}_o \leq \frac{R}{b \cdot r \cdot P}$$

   where $R$ is the total registers on the entire GPU and $b$ is the number of threads per $TB$.

Thus, we can compute the oversubscription of kernel $K$ as

$$\mathcal{M}_o = min\left(\frac{M}{b \cdot E \cdot P}, \frac{R}{b \cdot r \cdot P}\right)$$

Unfortunately, we cannot directly estimate ILP ($\mathcal{M}_I$) for a given algorithm without considering

the algorithm and implementation itself. Thus, we rely on fine-grain analysis to determine $\mathcal{M}_I$. We note, however, that if such fine-grain analysis is infeasible, the SPT model can be used with $\mathcal{M}_I = 1$ and the effects of oversubscription will still be incorporated into runtime estimation.

## 4.8 Estimating GPU Execution Time

The instantiation of the SPT model on our GPU platforms lets us analyze an arbitrary algorithm to determine what types of operations most impact overall performance. Furthermore, the latency and bandwidth values obtained with our microbenchmarks enable us to estimate the runtime of the algorithm on a specific GPU. However accurate runtime estimates require accurate counts of the number times the algorithm performs each operation ($\mathcal{A}_\phi$, for each of $\phi \in \{g, s, r, y\}$). We propose a two-phase analysis that allows us to get an accurate operation counts without having to analyze every aspect of a given implementation. For each kernel $\mathcal{K}_i$ of an algorithm, we first determine an asymptotic bound for $\mathcal{A}_{i,g}$, $\mathcal{A}_{i,s}$, $\mathcal{A}_{i,r}$, and $\mathcal{A}_{i,y}$ by analyzing the kernel using an asymptotic model that captures the memory access pattern of each memory type (the PEM model for $\mathcal{A}_{i,g}$, CREW PRAM for $\mathcal{A}_{i,s}$ and $\mathcal{A}_{i,r}$, and simply counting for $\mathcal{A}_{i,y}$). With the resulting asymptotic bounds, we identify the portions of the kernel execution that correspond to the asymptotically dominating terms, for each type of operation. We then perform a fine-grain analysis of the portions of the kernel that correspond to these asymptotically dominating terms to estimate the associated constant factors and give us a precise estimate of $\mathcal{A}_{i,g}$, $\mathcal{A}_{i,s}$, $\mathcal{A}_{i,r}$, and $\mathcal{A}_{i,y}$. We similarly determine the ILP, $\mathcal{M}_{i,I}$ associated with these dominating terms. Finally, we compute oversubscription, $\mathcal{M}_{i,o}$ based on the amount of of shared memory and number of registers used by each TB. Combining all of these terms, we compute an estimate of the runtime of kernel $\mathcal{K}_i$ (in clock cycles) to be $T_i = T_{i,g} + T_{i,s} + T_{i,r} + T_{y,i}$, where

$$\begin{aligned}
T_{i,g} &= \mathcal{A}_{i,g} \cdot max\left(\frac{1}{\mathcal{B}_g}, \frac{L_g}{\mathcal{M}_i}\right), \\
T_{i,s} &= \mathcal{A}_{i,s} \cdot max\left(\frac{1}{\mathcal{B}_s}, \frac{L_s}{\mathcal{M}_i}\right), \\
T_{ir} &= \mathcal{A}_{i,r} \cdot max\left(\frac{1}{\mathcal{B}_r}, \frac{L_r}{\mathcal{M}_i}\right), \\
T_{i,y} &= \mathcal{A}_{i,y} \cdot max\left(\frac{1}{\mathcal{B}_y}, \frac{L_y}{\mathcal{M}_{i,o}}\right).
\end{aligned}$$

We repeat this process for all kernels and estimate the total algorithm runtime as

$$\mathcal{T} = |\mathcal{K}| \cdot L_{sync} + \sum_{i=1}^{|\mathcal{K}|} T_i$$

Table 4.2: Hardware details and measured parameters for our three GPU platforms. Parameters marked with * are empirically measured.

| Parameter (units) | Description | ALGOPARC | GIBSON | UHHPC |
|---|---|---|---|---|
| $P$ (total cores) | Total compute cores | 1664 | 1536 | 2880 |
| $M$ (total 4-byte elements) | Total shared mem. | 312K | 128K | 240K |
| $R$ (total 4-byte elements) | Total registers on GPU | 832K | 512K | 960K |
| $^*L_g$ (clock cycles) | Global mem. latency | 269.5 | 267.6 | 291.2 |
| $^*L_s$ (clock cycles) | Shared mem. latency | 85.84 | 123.1 | 111.9 |
| $^*L_r$ (clock cycles) | Register op. latency | 6 | 10 | 10 |
| $^*L_y$ (clock cycles) | SYNCTHREADS latency | 234 | 304 | 289 |
| $^*\mathcal{B}_g$ (elts. per clock, per core) | Global mem. bandwidth | 0.0301 | 0.0279 | 0.0275 |
| $^*\mathcal{B}_s$ (elts. per clock per core) | Shared mem. bandwidth | 0.233 | 0.130 | 0.131 |
| $^*\mathcal{B}_r$ (elts. per clock per core) | Register op. bandwidth | $\sim 1$ | $\sim 1$ | $\sim 1$ |
| $^*\mathcal{B}_y$ (elts. per clock per core) | SYNCTHREADS bandwidth | .0493 | .0364 | .0394 |
| $^*L_{sync}$ (clocks cycles) | Time of DEVICESYNC | 7800 | 10850 | 8640 |

# CHAPTER 5
# CASE STUDY: MATRIX MULTIPLICATION

General matrix-matrix multiplication (GEMM) is one of the most fundamental and frequently performed dense linear algebra operations. Given an $n \times k$ matrix $A$ and a $k \times m$ matrix $B$, we wish to find the $n \times m$ matrix, $C$, defined as $C = A \times B$. There are many variations of this operation, although in this work, we consider only the simple GEMM of the form $C = A \times B$. For more details on about the GEMM operation and related algorithms, see general resources on linear algebra [74] and algorithms [17].

## 5.1 State-of-the-art GPU Matrix Multiply

There are libraries that provide efficient implementations of commonly-used linear algebra algorithms, such as general matrix-matrix multiplication (GEMM), for specific hardware architectures. On the GPU, many implementations focus on optimizations that reducing global memory accesses and increasing parallelism. The GEMM operation is well-suited for GPUs because it is easily parallelizable and requires little synchronization. Furthermore, the classic GEMM algorithm is compute bound, requiring $O(n^3)$ computations to multiply two $n \times n$ matrices. Nevertheless, there are several optimizations that can improve GEMM performance on GPUs, such as *tiling* [74]. Tiling improves performance by assigning a tile of the output matrix $C$ to a thread and then having it repeatedly load *batches* of elements into cache to perform matrix multiplication on. The results of each batch multiplication are stored as partial results. Once all batches have been processed, the final result is written back to memory.

In this work, we focus on the state-of-the-art GEMM implementation that is available with the MAGMA library [77]. The algorithm they use, which we refer to hereafter as MAGMA-GEMM, is optimized to take advantage of all levels of the GPU memory hierarchy. Since our instantiated SPT model focuses on memory access times, we expect it to be able to accurately predict the impact of optimizations like tiling. We first present an overview of the MAGMA-GEMM algorithm. Then, we generate runtime estimates with our SPT model and compare them with empirical results.

### 5.1.1 Algorithm details

To attain the parallelism needed to achieve high performance on the GPU, the MAGMA-GEMM algorithm employs the standard parallelism technique of tiling the output matrix, $C$. The MAGMA-GEMM algorithm assigns each thread-block (TB) to one tile of the $C$ matrix (of size $N_{TBX} \times N_{TBY}$), as illustrated in Figure 5.1. The parameters $N_{TBX}$ and $N_{TBY}$, therefore define the number of rows of $A$ and columns of $B$ that each thread-block processes, respectively. We note that, since $N_{TBX}$

and $N_{TBY}$ are hard-coded constants, as $n$ and $m$ grow, the number of thread-blocks used increases. The process for each tile is the same, so we simply describe how a single TB computes its tile.

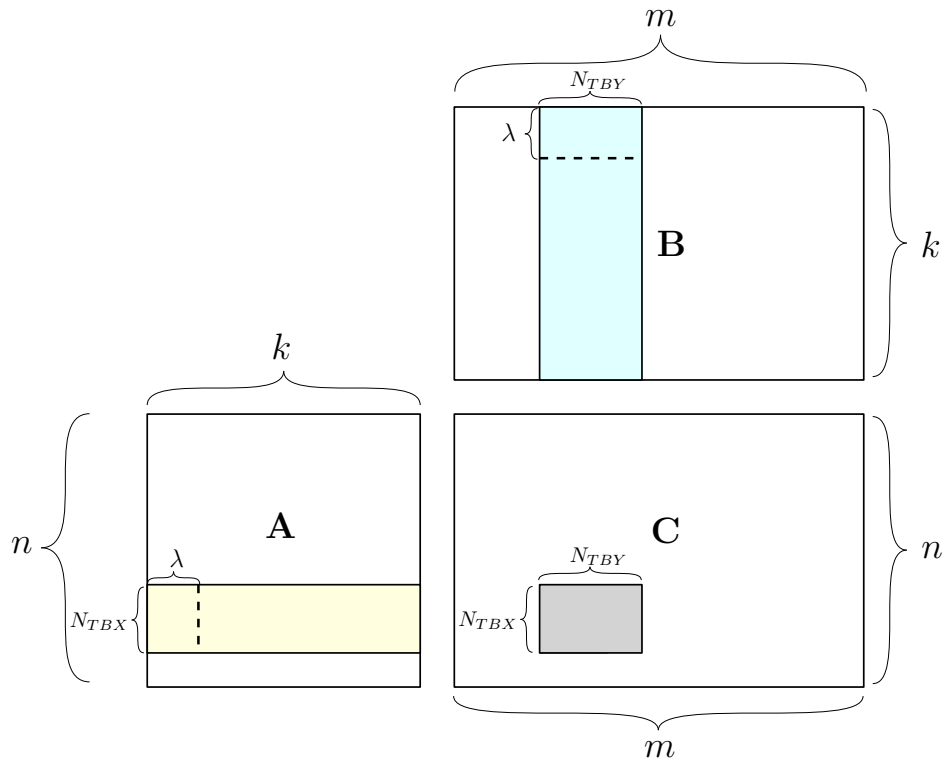


Figure 5.1: Illustration of the work done by a single TB in the MAGMA-GEMM algorithm.

A TB computes its tile in a series of batches to make use of shared memory and registers and reduce global memory accesses. As shown in Figure 5.1, the TB begins by loading a batch of $N_{TBX} \cdot \lambda$ elements from $A$ and $N_{TBY} \cdot \lambda$ elements from $B$ into shared memory. We note that the MAGMA-GEMM implementation uses $N_{TBX} = N_{TBY}$, so we define $N_{TB} = N_{TBX} = N_{TBY}$ and use it to simplify analysis. Thus, each TB loads $2 \cdot N_{TB} \cdot \lambda$ elements into shared memory. Each thread of the TB then loads a portion of the batch into registers, multiplies them, and adds the result to a set of partial sums it keeps in registers. The TB continues until all $\frac{k}{\lambda}$ batches have been processed, then each thread writes its result from registers to the final output array, $C$, in global memory. For more details about this algorithm, we refer interested readers to the MAGMA documentation and published works [77, 20, 63].

## 5.2 Algorithm analysis

We now analyze the MAGMA-GEMM algorithm in the context of our SPT model (detailed in Chapter 3). We consider each type of operation individually, and combine the results to develop an overall performance estimate. For each type of operation, we i) determine the asymptotically dom-

inating terms, ii) perform fine-grain analysis to determine constant factors, and iii) combine the result with hardware constants (measured in Chapter 4). Finally, we determine $\mathcal{M}$ by estimating the oversubscription, $\mathcal{M}_o$ and ILP, $\mathcal{M}_I$ of the algorithm, allowing us predict runtime.

Matrix-matrix multiplication requires little synchronization between threads. The MAGMA-GEMM algorithm requires only a single kernel, simplifying our analysis and letting us estimate runtime as

$$\mathcal{T} = L_{sync} + \mathcal{T}_g + \mathcal{T}_s + \mathcal{T}_r + \mathcal{T}_y.$$

### 5.2.1 Global memory accesses

One key aspect of the MAGMA-GEMM algorithm, tiling, allows blocks of memory to be loaded into shared memory to reduce the overall number of global memory accesses. Each TB loads $N_{TB} \cdot \lambda$ elements from $A$ and $N_{TB} \cdot \lambda$ elements from $B$. This allows for coalesced accesses and prevents repeated reading of these elements when performing the multiplication. Each TB repeats this for $\frac{k}{\lambda}$ batches, for a total global memory reads per TB of

$$\frac{2 \cdot N_{TB} \cdot \lambda \cdot k}{\lambda \cdot w},$$

where each coalesced read accesses $w$ consecutive elements. There are a total of $\frac{n}{N_{TB}} \cdot \frac{m}{N_{TB}} = \frac{n \cdot m}{(N_{TB})^2}$ tiles (and therefore TBs), thus the number of global memory accesses to load all batches is

$$\frac{2 \cdot N_{TB} \cdot \lambda \cdot k}{\lambda \cdot w} \cdot \frac{n \cdot m}{(N_{TB})^2} = \frac{2 \cdot n \cdot m \cdot k}{N_{TB} \cdot w}.$$

Once the results are computed, each tile must be written back to global memory to create the final $C$ matrix, for a total number of coalesced global memory *writes* of

$$\frac{(N_{TB})^2}{w} \cdot \frac{n \cdot m}{(N_{TB})^2} = \frac{nm}{w}.$$

There are $P$ total processors and each coalesced access enables a warp ($w$ threads) to read blocks of $w$ elements. Determining the constant factors associated with $\mathcal{A}_g$ is simple for this algorithm. Each read occurs only once and the final matrix $C$ is only written once, so

$$\begin{aligned} \mathcal{A}_g &= \frac{1}{P/w} \cdot \left( \frac{2 \cdot n \cdot m \cdot k}{N_{TB} \cdot w} + \frac{nm}{w} \right) \\ &= \frac{2 \cdot n \cdot m \cdot k}{P \cdot N_{TB}} + \frac{n \cdot m}{P}. \end{aligned}$$

### 5.2.2 Shared memory accesses

Once a TB loads a batch of data into shared memory, each *thread* reads $\frac{N_{TB} \cdot \lambda}{b}$ elements from the blocks of $A$ and $B$ (in shared memory) and stores them in registers, where $b$ is the number of threads per TB. Once loaded into registers, each thread multiples the elements and stores the resulting partial sum in registers as well. Since the multiplication operation itself happens in registers, each element in shared memory is only accessed a constant number of times. Thus, the number of parallel shared memory accesses is, asymptotically, equivalent to global memory, except for writing the final $C$ array, which happens directly from registers to global memory. We note that this algorithm is completely bank conflict-free, so $\beta = 1$. Unlike global memory, however, the constant factors associated with the asymptotic terms are not all 1. Each element in shared memory is accessed once when writing from global memory to shared memory, and again when reading shared memory and storing it in registers. Therefore,

$$\mathcal{A}_s = 4 \cdot \frac{n \cdot m \cdot k}{P \cdot N_{TB}}.$$

### 5.2.3 Register operations

The majority of work done by MAGMA-GEMM occurs in registers. For each batch (blocks from $A$ and $B$) loaded into shared memory, each thread loads a portion into registers and performs matrix multiplication on them. If each TB has $b$ threads and stores $2N_{TB} \cdot \lambda$ elements in shared memory per batch, each thread loads $\frac{2 \cdot N_{TB} \cdot \lambda}{b}$ elements into registers. Once loaded, each thread performs matrix multiplication, requiring $\Theta\left(\frac{(N_{TB})^2 \cdot \lambda}{b}\right)$ operations. Each thread performs its own matrix multiplication, so the entire TB must perform $(N_{TB})^2 \cdot \lambda$ accesses per batch. Recall that each TB loads $\frac{k}{\lambda}$ blocks and there are a total of $\frac{n \cdot m}{(N_{TB})^2}$ thread-blocks. Thus, the number of register accesses is

$$\begin{aligned} \mathcal{A}_r &= \Theta\left(\frac{1}{P} \cdot \left((N_{TB})^2 \cdot \lambda \cdot \frac{k}{\lambda} \cdot \frac{n \cdot m}{(N_{TB})^2}\right)\right) \\ &= \Theta\left(\frac{n \cdot m \cdot k}{P}\right). \end{aligned}$$

We note that this is the lower bound for the naive GEMM algorithm in the CREW PRAM model [17].

To determine the constant factors associated with $\mathcal{A}_r$, we need to consider the code to count register operations themselves. While counting the total register operations of a given program can be difficult, we are only concerned with operations associated with the asymptotically dominating term above. Within the innermost loop of the GEMM algorithm, the MAGMA-GEMM implementation uses the specialized hardware operation, FMA (fused multiply-add), thereby requiring only one operation. Thus, the constant factor associated with the dominating term for register operations is 1. This illustrates that, by considering constant factors and register operations, our SPT model

is able to capture the effects of low-level optimizations as well. With a leading constant factor of 1, the total register operations is

$$\mathcal{A}_r = \frac{n \cdot m \cdot k}{P}$$

### 5.2.4 Thread-block synchronizations

While the MAGMA-GEMM algorithm only requires one DEVICESYNC operation, threads within each TB must synchronize periodically. Since all threads work on the same blocks of shared memory (both loading and reading), they must use SYNCTHREADS. Specifically,before and after loading each pair of blocks from global memory to shared memory, a TB calls SYNCTHREADS. Each TB loads a total of $\frac{k}{\lambda}$ pairs of blocks (batches), and there are $\frac{n \cdot m}{(N_{TB})^2}$ thread-blocks. For each block, each TB reads each element into shared memory and writes it out, thus

$$\mathcal{A}_y = \frac{2 \cdot k \cdot n \cdot m}{P \cdot (N_{TB})^2 \cdot \lambda}.$$

### 5.2.5 Multiplicity, $\mathcal{M}$

To determine $\mathcal{M}$, we compute oversubscription, $\mathcal{M}_o$ and ILP, $\mathcal{M}_I$. We note that, while $\mathcal{M}_o$ remains constant for the entire kernel, each type of memory access may have different values of $\mathcal{M}_I$, so we determine it for each.

#### Oversubscription, $\mathcal{M}_o$

Recall from Section 4.7 that oversubscription can be limited by either shared memory usage or register usage. Since MAGMA-GEMM uses both, we compute the limitation to $\mathcal{M}_o$ caused by each, based on the algorithm parameters. Each TB of $b$ threads stores $2 \cdot N_{TB} \cdot \lambda$ elements in shared memory. If the GPU can store a total of $M$ elements in shared memory,

$$\mathcal{M}_o \leq \frac{M}{(2 \cdot N_{TB} \cdot \lambda \cdot (P/b))} = \frac{M \cdot b}{P \cdot 2 \cdot N_{TB} \cdot \lambda}$$

Each thread stores elements in a number of registers to enable fast computation: elements loaded from block $A$, elements loaded from block $B$, and partial sums to be eventually written to the final output matrix $C$. Per thread, the number of elements loaded from $A$ and $B$ into registers is $\frac{2 \cdot N_{TB} \cdot \lambda}{b}$. Each thread also keeps $\frac{(N_{TB})^2}{b}$ partial sums in registers. Thus, the use of registers limits $\mathcal{M}_o$ to

$$\mathcal{M}_o \leq \frac{R \cdot b}{P \cdot (2 \cdot N_{TB} \cdot \lambda + (N_{TB})^2)},$$

where $R$ is the total registers available on the GPU. For many modern GPUs, such as all three of

our hardware platforms, $R$ and $M$ are roughly the same (see values for our hardware platforms in Table 4.2. Since each thread uses more registers than shared memory, we say that $\mathcal{M}_o$ is limited by register usage and use that value to compute $\mathcal{M}_o$ when estimating performance.

**ILP, $\mathcal{M}_I$**

Estimating the ILP of a given set of operations requires that we consider the algorithm and implementation details and estimate the average number of independent operations that can be performed by a single thread. For the MAGMA-GEMM algorithm, threads are operating on many different pairs of elements with very little dependence between operations. When accessing both global and shared memory, each thread simply reads elements and stores them elsewhere (either in shared memory or registers). Thus, when computing $\mathcal{T}_g$ and $\mathcal{T}_s$, we say that $\mathcal{M}_I$ is equal to the number of elements each thread accesses, i.e.,

$$\mathcal{M}_I = \frac{2 \cdot N_{TB} \cdot \lambda}{b}$$

Thus, when computing $\mathcal{T}_g$ and $\mathcal{T}_s$,

$$\begin{aligned} \mathcal{M} &= \mathcal{M}_o \cdot \mathcal{M}_I \\ &= \frac{R \cdot b}{P \cdot (2 \cdot N_{TB} \cdot \lambda + (N_{TB})^2)} \cdot \frac{2 \cdot N_{TB} \cdot \lambda}{b} \\ &= \frac{2 \cdot R \cdot \lambda}{P \cdot (2 \cdot \lambda + N_{TB})} \end{aligned}$$

In practice, $R$ is quite large ($> 500K$ elements), so ILP is large enough that shared memory and global memory accesses are bandwidth-bound on our hardware platforms. Registers, however, cannot benefit from as much ILP in this algorithm. To see this, we consider the matrix multiplication operation that each thread performs in registers. We denote $a_i$, $b_i$, and $c_i$, as the elements that a thread stores from matrix $A$, $B$, and $C$, respectively. Assuming that each thread operates on $x^2$ elements from $A$ and $B$, the matrix multiplication performs the following pattern:

$$\begin{aligned} c_1 &\stackrel{+}{=} a_1 \cdot b_1 \\ c_1 &\stackrel{+}{=} a_2 \cdot b_2 \\ &\cdots \\ c_1 &\stackrel{+}{=} a_x \cdot b_x \\ c_2 &\stackrel{+}{=} a_1 \cdot b_{x+1} \\ c_2 &\stackrel{+}{=} a_2 \cdot b_{x+2} \\ &\cdots \end{aligned}$$

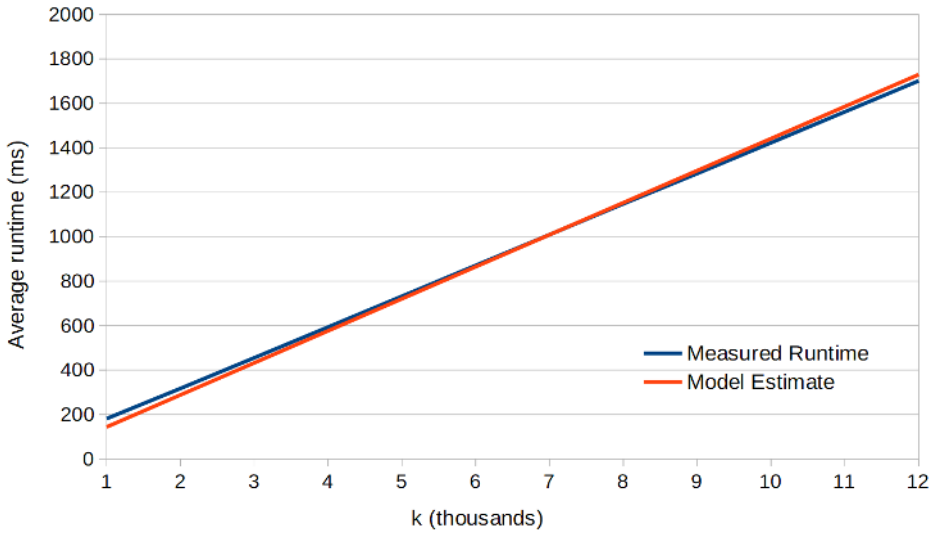


Figure 5.2: Measured runtime, compared with our model estimate, for $n = m = 10^4$ and varying $k$ on ALGOPARC.

We can see that the same $c_i$ register is being accessed for $x$ consecutive steps, preventing the operations from being independent and reducing ILP. Specifically, we only have 2 independent operations every $x$ steps, so we say that ILP is approximately 1 (i.e., $\mathcal{M}_I = 1$) for register operations.

## 5.3 Verifying Model Estimate

We put together all of the results of our analysis of the MAGMA-GEMM algorithm to develop a single, parameterized runtime estimate for each of our hardware platforms. Thus, our overall performance estimate of runtime is $T = L_{sync} + \mathcal{T}_g + \mathcal{T}_s + \mathcal{T}_r + \mathcal{T}_y$, where

$$\begin{aligned}
\mathcal{T}_g &= \left(\frac{2 \cdot n \cdot m \cdot k}{P \cdot N_{TB}} + \frac{n \cdot m}{P}\right) \cdot max\left(\frac{1}{\mathcal{B}_g}, \frac{L_g}{\mathcal{M}}\right), \\
\mathcal{T}_s &= \left(4 \cdot \frac{n \cdot m \cdot k}{P \cdot N_{TB}}\right) \cdot max\left(\frac{1}{\mathcal{B}_s}, \frac{L_s}{\mathcal{M}}\right). \\
\mathcal{T}_r &= \left(\frac{n \cdot m \cdot k}{P}\right) \cdot max\left(\frac{1}{\mathcal{B}_r}, \frac{L_r}{1 \cdot \mathcal{M}_o}\right), \\
\mathcal{T}_y &= \left(\frac{2 \cdot k \cdot n \cdot m}{P \cdot (N_{TB})^2 \cdot \lambda}\right) \cdot max\left(\frac{1}{\mathcal{B}_y}, \frac{L_y}{\mathcal{M}_o}\right),
\end{aligned}$$

where $\mathcal{M} = \frac{2 \cdot R \cdot \lambda}{P \cdot (2 \cdot \lambda + N_{TB})}$ and $\mathcal{M}_o = \frac{R \cdot b}{P \cdot (2 \cdot N_{TB} \cdot \lambda + (N_{TB})^2)}$.

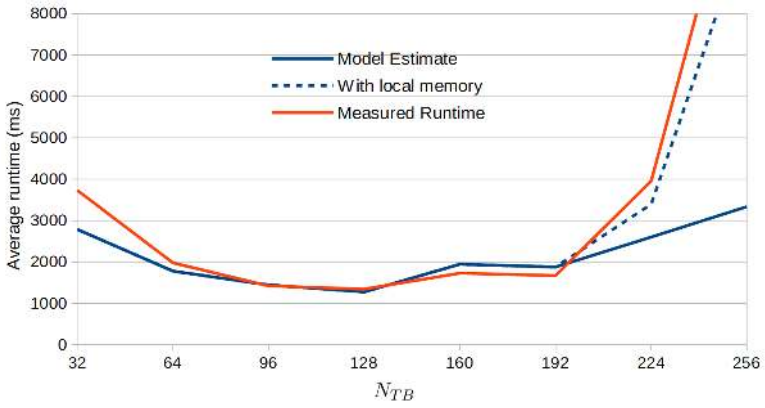


Figure 5.3: Measured runtime and our model estimate, for $n = m = k = 10^4$ and varying $N_{TB}$ on ALGOPARC. The dotted line shows our model if take into account the impact of using local memory when each thread requires too many registers.

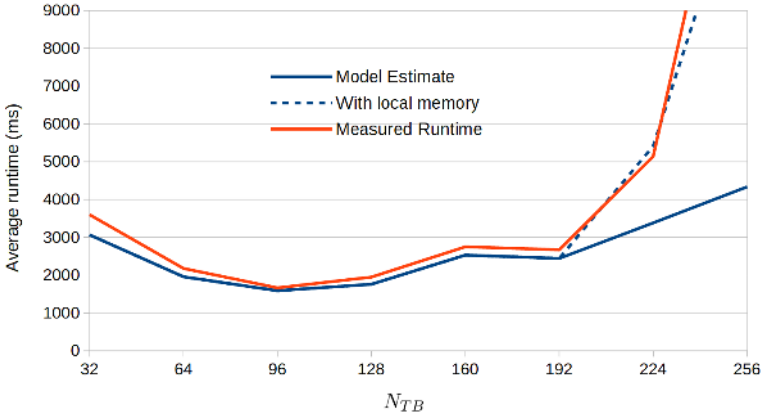


Figure 5.4: Measured runtime and our model estimate, for $n = m = k = 10^4$ and varying $N_{TB}$ on ALGOPARC. The dotted line shows our model if take into account the impact of using local memory when each thread requires too many registers.

### 5.3.1 Accuracy of runtime estimate

We verify the accuracy of our performance model by simply running MAGMA-GEMM on each of our hardware platforms for varying matrix sizes ($n$, $m$, and $k$). We use the MAGMA hard-coded values of $N_{TB} = 96$, $\lambda = 16$, and $b = 256$. Figure 5.2 plots our performance estimate and measured runtimes, averaged over 10 iterations, for $n = m = 10^4$ and varying $k$, on ALGOPARC. These results indicate that our SPT model accurately estimates the average runtime of MAGMA-GEMM, with an average error of 3.8% and 5.7% on ALGOPARC and GIBSON, respectively.[1]

[1]Unfortunately, we were unable get the MAGMA library working on the UHHPC platform.

### 5.3.2 Modeling algorithm parameters

Our performance estimate of the MAGMA-GEMM algorithm relies on several parameters that are hard-coded to specific values in the MAGMA implementation. To determine how accurate our model is with respect to these parameters, as well as to evaluate how they impact overall performance, we run a series of experiments that vary these parameters. We modify the source code of MAGMA-GEMM to vary the $N_{TB}$ parameter value and measure algorithm runtime for fixed $n = m = k$ matrix sizes. Figures 5.3 and 5.4 display the measured runtime, compared with our model estimate, for varying $N_{TB}$ parameter values and fixed $n = m = k = 10^4$, $\lambda = 16$ and $b = 256$, on ALGOPARC and GIBSON, respectively. These results indicate that, except for very small or large values of $N_{TB}$, our runtime estimate is accurate. Interestingly, it further shows that, for the ALGOPARC platform, the hard-coded $N_{TB} = 96$ is not ideal, and we achieve better performance when $N_{TB} = 128$. As we see in Figure 5.3, our runtime estimate accurately predicts this and it is confirmed by the measured runtime.

We see that, when $N_{TB}$ becomes large, our model significantly under-estimates the runtime. Recall that MAGMA-GEMM uses $\frac{(N_{TB})^2+2\cdot N_{TB}\lambda}{b}$ registers per thread. Thus, when $N_{TB}$ becomes large, each thread uses many of registers. Since there are a limited number of hardware registers, when a thread requires too many, the GPU instead uses *local* memory for excess register storage [65]. Local memory is a portion of global memory that is cached in shared memory. To estimate the impact of local memory on MAGMA-GEMM runtime, we compute the number of registers used, per thread, for each $N_{TB}$. When the total registers used exceeds 255 (the hardware limit per thread), we say that a portion of register operations require shared memory accesses (i.e., local memory). Assuming that all registers require the same number of accesses, we estimate the number of register operations that require shared memory accesses to be $\mathcal{A}_r \cdot \frac{r-255}{r}$, where $r$ is the total registers used per thread. The dotted line in Figure 5.3 shows our new model estimate that includes the cost of local memory accesses. We see that it much more accurately predicts MAGMA-GEMM runtime for large $N_{TB}$ values.

## 5.4 Conclusion

In this chapter we applied the SPT model, defined in Chapter 3, to the analysis of matrix-matrix multiplication on our GPU platforms. We considered the state-of-the-art library implementation of naive general matrix multiplication available with the MAGMA library [77]. By comparing empirical runtime with the runtime estimate from the SPT model on our ALGOPARC and GIBSON platforms, we found that the model accurately predicts runtime. Furthermore, results indicate that our analysis with the SPT model determines performance bottlenecks and is able to determine parameter configuration for peak performance.

While our analysis of the MAGMA-GEMM algorithm indicates that it suffers from no glaring

performance bottlenecks, the algorithm itself is sub-optimal. The naive algorithm used by the MAGMA library requires $O(n^3)$ total operations, although advances have been made that reduce this to $O(n^{2.373})$ [81]. These "fast matrix multiplication" algorithms, however, are significantly different from the naive approach, so developing an efficient version for GPU architectures may be challenging. Furthermore, optimization techniques such as tiling do not apply to these algorithms and it is unknown how to achieve I/O-efficiently. Nevertheless, the asymptotic improvements these fast matrix multiplication algorithms provide can result in significant practical performance improvements if they can make efficient use of GPU architectures. Thus, a promising area of future work is to use the SPT model to develop a GPU-efficient fast matrix multiplication algorithm.

# CHAPTER 6
# CASE STUDY: SEARCHING

Searching encompasses a broad class of problems that are frequently seen as a subproblem in other applications. For a typical searching problem, we are provided with an input of $N$ items and a series of queries, $Q$. We are asked to search through the input to answer each query. In this chapter, we consider the simple *predecessor* search on the GPU for two cases: if the input $N$ is large and is stored in global memory and 2) if the input is small and is stored in shared memory.

We formally define the *batched predecessor search* problem as:

**Definition 6.0.1** *Given a list $\mathcal{N}$ of $N$ keys $\mathcal{N}[0]$, $\mathcal{N}[1]$, $\cdots$, $\mathcal{N}[N-1]$, sorted in non-decreasing order, and a set $\mathcal{Q}$ of $Q$ queries, the batched predecessor search problem asks to find for each $q \in \mathcal{Q}$ the largest $i$, such that $\mathcal{N}[i] \leq q$.*

Batched predecessor search can be (sequentially) solved optimally in the standard RAM model in $O(Q \log N)$ time by running the classical binary search algorithm on $\mathcal{N}$ for each query $q \in \mathcal{Q}$. In the $P$-processor CREW PRAM model (described in Section 1.3), a straightforward and optimal parallelization consists in performing each binary search concurrently, achieving $O(\frac{Q}{P} \log N)$ time.

## 6.1 Searching in Global Memory

In this section we consider the *batched predecessor search* problem when the sorted set of search keys $\mathcal{N}$ is in global memory (i.e., when $N$ is large and does not fit in shared memory). Since we are only concerned with performing queries, we do not consider dynamic search structures and can focus on *implicit* search layouts [46] only. Implicit search layouts are ways of organizing the elements of a list to facilitate searching. Many well-known search layouts have been proposed [15, 25, 17] with differing benefits. For this work, we consider three layouts: a naive sorted list, a level-order binary search tree (BST) layout, and a static B-tree layout. Searching each layout results in a different memory access pattern, which may impact performance.

### 6.1.1 Naive Sorted List

For a given list $\mathcal{N}$ containing $N$ search keys in global memory, a simple sorted list can be searched efficiently using a standard binary search technique [46]. Thus, we can solve the *batched predecessor search* of $Q$ queries by performing binary searches concurrently. Since this section focuses on the performance of searching different memory layouts, we refer to the process of searching this layout as SORTED.

On the GPU, this batched predecessor search is performed in a single kernel, with $Q$ threads each performing a single query. $\mathcal{N}$ is stored in global memory so each thread performs binary search

by accessing elements in global memory. Recall from Section 1.2 that global memory is accessed in blocks of $w$ elements by a warp of $w$ threads. To achieve peak performance, we must make use of all $w$ elements in each block that we access. However, naive binary search exhibits very little spatial locality of data accesses, as the distance between subsequent search elements is large (until the end of the search). A naive binary search requires $\lceil \log N \rceil$ steps, accessing a key at each step, where the distance between each key being accessed decreases by a factor $\frac{1}{2}$ per step. Thus, it is not until the last $\log w$ steps that we can use more than 1 element per block accessed, resulting in $(\lceil \log N - \log w \rceil + 1)$ global memory accesses per query. Thus, the total number of *uncoalesced* accesses is to search $Q$ queries on the SORTED layout is

$$\frac{Q}{P} \cdot (\lceil \log N - \log w \rceil + 1).$$

Since all threads are performing their own binary search, we estimate that all threads will access different blocks, leading to completely uncoalesced accesses (each warp accesses $w$ blocks per read). Thus, we estimate the total global memory accesses to be

$$\begin{aligned} \mathcal{A}_g &= \frac{Q}{P/w} \cdot (\lceil \log N - \log w \rceil + 1) \\ &= \frac{Q}{P/w} \cdot \left( \left\lceil \log \frac{N}{w} \right\rceil + 1 \right) \end{aligned}$$

We see that uncoalesced accesses reduce parallelism (and thereby increase the runtime) by a factor $w$. We note, however, that the additional memory accesses caused by uncoalesced accesses are all issued independently, increasing ILP ($\mathcal{M}_I$) by a factor $w$. Since little additional work is done in registers or shared memory, we estimate the total runtime of performing batched predecessor search on the SORTED layout to be

$$\mathcal{T} = \frac{Q}{P/w} \cdot \left( \left\lceil \log \frac{N}{w} \right\rceil + 1 \right) \cdot max \left( \frac{1}{\mathcal{B}_g}, \left\lceil \frac{L_g}{\mathcal{M} \cdot w} \right\rceil \right) + L_{sync}.$$

where $L_{sync}$ is included because SYNCTHREADS is called once after the batched predecessor query kernel. This algorithm uses no shared memory and few registers, so global memory accesses are bandwidth bound (i.e., $\left( \frac{1}{\mathcal{B}_g} > \left\lceil \frac{L_g}{\mathcal{M} \cdot w} \right\rceil \right)$).

### 6.1.2 Level-order Binary Search Tree

A simple layout that facilitates easy searching is the level-order binary search tree (BST) layout. The BST layout is defined by the breadth-first (level-order) left-to-right traversal of a complete binary search tree. Given the index $i$ of node $v$ in the BST layout, the indices left and right children of $v$ are $2i + 1$ and $2i + 2$, respectively. Note that the layout indices start at 0 (e.g., the root of the BST is at index 0, with left and right children at indexes 1 and 2, respectively).

This layout provides two benefits over the standard sorted list: i) index calculation is very simple when searching and ii) the beginning of each query (i.e., the top of the BST) has a high amount of data locality, regardless of the query, allowing us to cache the beginning of the layout and improve performance. We note that the BST layout has been shown to provide good performance on CPUs due to cache prefetching [42, 12], though it is not clear that this applies to GPUs.

When analyzing the batched predecessor problem on the BST layout, we can use the fact that all queries must access elements at the top of the tree structure. If the top of the BST is cached in shared memory, the number of global memory accesses is reduced and performance is improved. Thus, when performing batched predecessor search on a BST layout, each thread-block (TB) first loads the top of the BST layout into shared memory. Each TB can store a total of $M_{TB} = \frac{M}{P/b} = \frac{Mb}{P}$ elements in shared memory, where $M$ is the total shared memory on the whole GPU (values for our platforms are in Table 4.2) and $b$ is the number of threads per TB. Initially loading first $M_{TB}$ elements into shared memory takes $\frac{M_{TB}}{b} = \frac{M}{P}$ global memory accesses, and once it is loaded, each query requires $\lceil \log N - \log M_{TB} \rceil$ global memory accesses. As with the sorted layout, all accesses are uncoalesced, so we estimate the total accesses for $Q$ queries to be

$$\mathcal{A}_g = \frac{M}{P} + \frac{Q}{P/w} \cdot \left( \left\lceil \log \frac{N}{M_{TB}} \right\rceil + 1 \right).$$

On all of our hardware platforms, $M_{TB} = 2^{14}$ when using 32-bit integers. Since the top $\log(M_{TB})$ levels of the tree are searched in the shared memory cache and there is only one kernel call, $\mathcal{T} = \mathcal{T}_g + \mathcal{T}_s + L_{sync}$, where

$$\begin{aligned} \mathcal{T}_g &= \left( \frac{M}{P} + \frac{Q}{P/w} \cdot \left\lceil \log \frac{N}{M_{TB}} \right\rceil \right) \cdot max \left( \frac{1}{\mathcal{B}_g}, \left\lceil \frac{L_g}{\mathcal{M} \cdot w} \right\rceil \right), \\ \mathcal{T}_s &= \left( \frac{M}{P} + \frac{Q}{P} \cdot \lceil \log M_{TB} \rceil \right) \cdot max \left( \frac{1}{\mathcal{B}_s}, \left\lceil \frac{L_s}{\mathcal{M}} \right\rceil \right). \end{aligned}$$

Note that when $Q > \frac{M}{w \log(N/M_{TB})}$) global memory accesses are dominated by querying. Similarly, when $Q > \frac{M}{\log(M_{TB})}$, querying dominates shared memory accesses as well. We assume that $Q$ is large, so the time to cache the first $M_{TB}$ elements into shared memory is small, compared to query time.

### 6.1.3 B-tree Layout

While the BST layout reduces the number of global memory accesses by allowing the top of the tree structure to remain in cache when searching, at the bottom of the BST there is no cache locality, resulting in excess memory accesses. The B-tree layout [15] avoids this by building a

layout specifically targeting the memory access block size of a given architecture. Each node of a *complete* B-tree contains exactly B elements and every internal node has exactly B + 1 children. The B-TREE layout is defined by the breadth-first left-to-right traversal of a B-tree. This layout is a generalization of the level-order BST layout (a BST layout is a B-TREE layout with $B = 1$). Thus, as with the BST layout, index computation is simple and the top portion of the tree can be stored in cache. However, the B-TREE layout also lets us use all $w$ elements from every global memory access.

For the GPU, we focus our analysis on the performance of the batched predecessor search of a B-TREE layout where $B = w$. This lets us read each node in the B-tree with a single global memory access. We can then search through the node in shared memory. The depth of a B-tree is $\lceil \log_{B+1} N \rceil$, and, as with the BST layout, we first load the top $M_{TB}$ elements of the tree into shared memory for each TB. Thus, the number of global memory and shared memory accesses per query is $\left\lceil \log_{w+1} \frac{N}{M_{TB}} \right\rceil$ and $\lceil \log_{w+1} N \cdot \log w \rceil$, respectively. Thus, for a batch of $Q$ predecessor queries, we estimate the total accesses to be

$$\begin{aligned} \mathcal{A}_g &= \frac{M}{P} + \frac{Q}{P/w} \cdot \left\lceil \log_{w+1} \frac{N}{M_{TB}} \right\rceil, \\ \mathcal{A}_s &= \frac{M}{P} + \frac{Q}{P} \cdot \lceil \log_{w+1} N \cdot \log w \rceil. \end{aligned}$$

No additional work is performed in registers, so $\mathcal{T} = \mathcal{T}_g + \mathcal{T}_s + L_{sync}$, where

$$\begin{aligned} \mathcal{T}_g &= \left( \frac{M}{P} + \frac{Q}{P/w} \cdot \left\lceil \log_{w+1} \frac{N}{M_{TB}} \right\rceil \right) \cdot max \left( \frac{1}{\mathcal{B}_g}, \frac{L_g}{\mathcal{M} \cdot w} \right), \\ \mathcal{T}_s &= \left( \frac{M}{P} + \frac{Q}{P} \cdot \lceil \log_{w+1} N \cdot \log w \rceil \right) \cdot max \left( \frac{1}{\mathcal{B}_s}, \frac{L_s}{\mathcal{M}} \right). \end{aligned}$$

As with the BST layout, when $Q$ is sufficiently large, the time spent querying is much larger than the time needed to load the top of the B-tree layout into shared memory.

### 6.1.4 Empirical Performance Results

Our analysis with the SPT model indicates that both the BST and B-TREE layouts should significantly reduce the number of global memory accesses needed to search, compared with a SORTED layout. In this section, we create each of these layouts and perform batched predecessor search, comparing the performance with the estimate obtained from our model. We evaluate the performance on each of our hardware platforms (detailed in Table 4.2) for different parameter values. Figures 6.1, 6.2, and 6.3 plot the measured and estimated runtime to perform a batch of predecessor queries on each of the three search layouts, for varying $Q$, on GIBSON, UHHPC, and ALGOPARC, respectively. These results indicate that our runtime estimate is accurate, despite not attempting to model the cache system directly. Furthermore, we do not perform any type of probabilistic

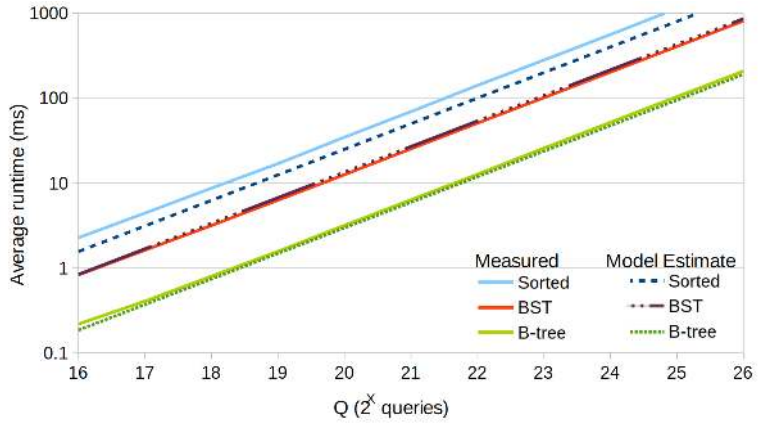


Figure 6.1: Measured and estimated runtime when querying different search tree layouts on GIBSON with $N = 2^{28}$ and varying number of queries ($Q$).

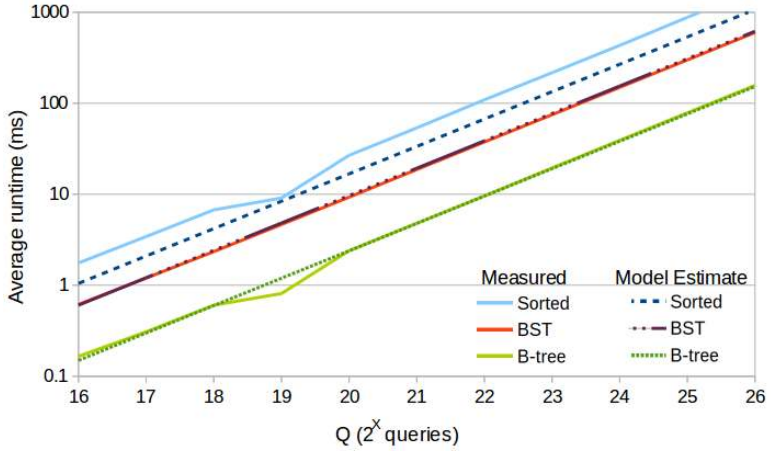


Figure 6.2: Measured and estimated runtime when querying different search tree layouts on UHHPC with $N = 2^{28}$ and varying number of queries ($Q$).

analysis on the access patterns of each query. We see that, on ALGOPARC, our runtime estimate has an average error rate of 5.71%, 21.65%, and 24.33%, compared with the measured runtime of querying SORTED, BST, and B-TREE, respectively. We attribute the higher error on the B-TREE and BST layouts to the effects of the cache system. Our estimate simply says that a portion of the tree is stored in cache and we ignore the other, more complex, cache interactions. On UHHPC and GIBSON, however, we see that our runtime estimate is the most accurate on B-TREE and BST, with an average error of 2.94%% and 6.51%, respectively, on UHHPC, and 5.67% and 8.09%, respectively, on GIBSON. Our model underestimates the runtime of querying the SORTED layout, however, with an average error of 35.77% and 28.84% on UHHPC and GIBSON, respectively. We postulate that this is due to the fact that these platforms have older, Kepler generation GPUs with fewer mechanisms to mitigate the performance loss due to uncoalesced accesses and branch divergence.

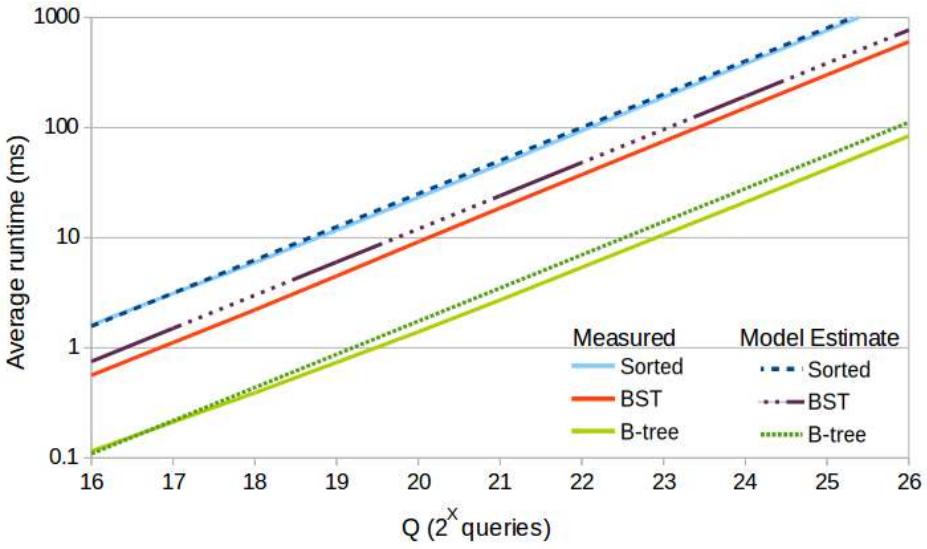


Figure 6.3: Measured and estimated runtime when querying different search tree layouts on ALGOPARC with $N = 2^{28}$ and varying number of queries ($Q$).

## 6.2 Searching in Shared Memory

In this section we consider the problem of performing a batch of $Q$ predecessor search operations on a list of $N$ sorted keys, stored in *shared memory*. Focusing on keys stored in shared memory lets us ignore many of the details of the GPU architecture that complicate analysis. Furthermore, batches of predecessor searches can be performed concurrently, avoiding the need for DEVICESYNC operations and allowing us to only consider a single kernel. Thus, in terms of the SPT model (detailed in Section 3), we are only concerned with computing the time spent accessing queries from global memory ($\mathcal{T}_g$) and searching in shared memory ($\mathcal{T}_s$). No additional work needs to be done in registers and no SYNCTHREADS operations are necessary. We note that, regardless of the search method used within shared memory, each query is only accessed once in global memory, so $\mathcal{A}_g = \frac{Q}{P}$. Furthermore, oversubscription ($\mathcal{M}_o$) and ILP ($\mathcal{M}_I$) are independent of the access pattern used to search shared memory. Searching requires dependent memory accesses, so $\mathcal{M}_I = 1$ and, since all $N$ keys are stored in shared memory, oversubscription is limited to $\mathcal{M}_o = \frac{M \cdot t}{N \cdot P}$.

Since global memory accesses, $\mathcal{A}_g$ and multiplicity $\mathcal{M}$ are independent of the queries themselves, the only thing that impacts the performance of batched predecessor search in shared memory is the access pattern of shared memory itself. Recall from Chapter 4 that the SPT model does not model bank conflicts directly, and instead defines $\beta$ as the average number of conflicts per shared memory access. Therefore, we only use the SPT model to evaluate the number of explicit memory accesses and rely on direct analysis of the memory access patterns of our different search algorithms to determine the number of bank conflicts. We first consider a naive binary search approach. We

**Algorithm 1:** Pseudo-code for binary search.

```
BinarySearch(𝒩, q):
index = ⌊N/2⌋
δ = ⌈N/4⌉
for ⌈log N⌉ times do
    if q ≥ 𝒩[index] then
        index = min(index + δ, N − 1)
    else
        index = max(index - δ, 0)
    δ = ⌈δ/2⌉
end
if q < 𝒩[index] then
    index = index - 1
return index
```

then introduce two new search algorithms: one completely bank conflict-free and one that limits bank conflicts in an effort to achieve peak performance.

### 6.2.1 Naive Binary Search

There are many small variations on how to implement binary search, each leading to a different worst-case input for the algorithm. We define here a simple version for concreteness of exposition, noting that our analysis can be performed for any of the variations. Algorithm 1 shows pseudo-code for the standard binary search algorithm for a single query $q$. The *min* and *max* functions are used to prevent out-of-bounds memory accesses, but in our implementations we instead pad the input to eliminate this extra computation.

Let *Parallel Binary Search (PBS)* be the straightforward parallel solution to the batched predecessor search problem by running Algorithm 1 for each query $q \in \mathcal{Q}$ concurrently. In the $P$-processor CREW PRAM model (discussed in Section 2.1), the PBS algorithm takes optimal $\Theta(Q \log N)$ work and $\Theta\left(\frac{Q}{P} \log N\right)$ time. However, if data $\mathcal{N}$ is stored in shared memory of the GPU, the serialization of memory accesses due to bank conflicts violates the PRAM assumption that access to each memory location takes unit time. Thus, the above bound is not representative for the worst-case runtimes on GPUs For simplicity of exposition we also assume that $w$ is a power of 2. Although this is the case for most modern GPUs, our results extend to other values of $w$ via straightforward, but tedious analysis.

#### Analysis of the PBS algorithm

The PBS algorithm uses executes a single kernel that performs all $Q$ queries on the list $\mathcal{N}$ in shared memory. Clearly, if the queries are in global memory, $\mathcal{A}_g = \frac{Q}{P}$. However, searching is performed in shared memory, so we focus our analysis on $\beta$ and $\mathcal{A}_s$. Our analysis combines both bank conflicts

are shared memory accesses, so we denote $\beta\mathcal{A}_s$ as the total number of shared memory accesses performed by the PBS algorithm, as a function of $Q$, $N$, $P$, and $w$.

**Theorem 6.2.1** *For every sorted sequence $\mathcal{N}$, $|\mathcal{N}| = N \geq 2w^2$, there exists a set of queries $\mathcal{Q}$, $|\mathcal{Q}| = Q \geq p$, for which*

$$\beta\mathcal{A}_s = \Omega\left(w \cdot \frac{Q}{P} \log \frac{N}{w^2}\right).$$

This theorem states that, in the worst case, the PBS algorithm performs additional accesses due to bank conflicts, resulting in performance loss on the GPU (compare to the PRAM complexity $\Theta(\frac{Q}{P} \log N)$). The proof of this theorem relies on the following lemma:

**Lemma 6.2.2** *For every sorted sequence $\mathcal{N}$, $|\mathcal{N}| = N \geq 2w^2$, there exists a query set $\mathcal{Q}$, $|\mathcal{Q}| = Q = w$, for which the total shared memory accesses required to run the PBS algorithm using $w$ threads of a warp is at least*

$$\beta\mathcal{A}_s = \Omega\left(w \cdot \log \frac{N}{w^2}\right).$$

**Proof** For simplicity, we construct the query set $\mathcal{Q}$ so that each query $q \in \mathcal{Q}$ is an element of $\mathcal{N}$. Recall from Section 1.2.2 that with $w$ memory banks, a bank conflict occurs when separate threads within a warp access shared memory cells $\mathcal{N}[i]$ and $\mathcal{N}[j]$, such that $i \neq j$ and $i \equiv j \pmod{w}$. Since the PBS algorithm is deterministic, for a given sorted sequence $\mathcal{N}$, the access pattern of each thread is a function of the query $q \in \mathcal{Q}$ that the thread is processing. More specifically, in the $k$-th iteration of the binary search loop ($k = 1, 2, \ldots, \log N$), a thread may access one of $2^{k-1}$ possible memory addresses. Let $r = 2 \log w + 1$. Then for the $w$ queries, in the $r$-th iteration, there are $w^2$ memory addresses that each of $w$ threads may be accessing. Therefore, by the pigeonhole principle, among these $w^2$ addresses, there exists a subset of $w$ *distinct* memory addresses that reside in the same memory bank. We can choose $\mathcal{Q}$ so that these $w$ distinct memory addresses are accessed by the $w$ threads in the $r$-th step, thus resulting in a $w$-way bank conflict.

Consider an arbitrary pair of threads $t_i$ and $t_j$ within the same warp, running on the above input $\mathcal{Q}$. Let them access memory addresses $i$ and $j$, respectively, in the $r$-th step. Since $i$ and $j$ are distinct, $|i - j| \geq N/2^r$. This implies that (1) we can set each query to one of $N/2^r$ possible entries of $\mathcal{N}$ and still cause $w$-way bank conflicts in iteration $r$; and (2) any choice of these values will result in each thread accessing distinct addresses in each of the final $\log(N) - r$ iterations (including iteration $r$). Of these $N/2^r$ choices, we choose each query so that in the rest of the algorithm all threads take the same branch of the `if` statement.

Let the addresses accessed by $t_i$ and $t_j$ in any iteration $k \geq r$ be $i + \delta_k$ and $j + \delta_k$, respectively. Since every pair of threads accesses the same bank in round $r$, $i \equiv j \pmod{w}$, and it follows that $i + \delta_k \equiv j + \delta_k \pmod{w}$. Thus, for $\log N - r + 1$ rounds, our input $\mathcal{Q}$ causes $w$-way bank

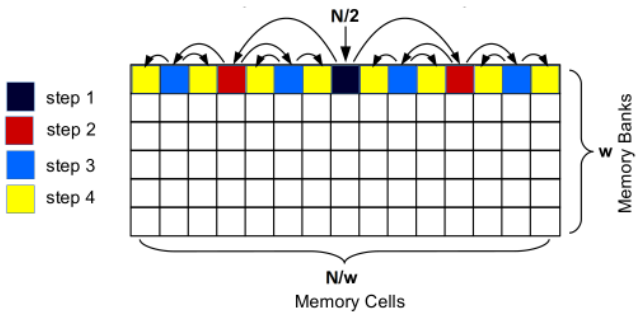


Figure 6.4: Worst-case example for the first $\log N - \log w$ iterations of the PBS algorithm, when $N$ is a multiple of $w^2$ (Corollary 6.2.1).

conflicts, resulting in a total number of shared memory accesses of at least $\Omega(w \cdot (\log N - r + 1)) = \Omega\left(w \cdot \log \frac{N}{w^2}\right)$. ∎

**Proof (of Theorem 6.2.1)** A typical (deterministic) GPU implementation breaks down $Q$ queries into $Q/w$ groups of $w$ queries each, and each group is processed in SIMD fashion by one of the $P/w$ warps. Note, that only $P/w$ out of $Q/w$ groups can be processed simultaneously. The PBS algorithm for the GPU assigns $w$ queries to each warp of $w$ threads, for a total of $\frac{Q}{w}$ warps. By Lemma 6.2.2, each warp performs $\Omega\left(w \cdot \log \frac{N}{w^2}\right)$ accesses, thus

$$\begin{aligned}\beta\mathcal{A}_s &= \Omega\left(\frac{Q/w}{P/w} \cdot \left(w \cdot \log \frac{N}{w^2}\right)\right) \\ &= \Omega\left(w \cdot \frac{Q}{P} \log \frac{N}{w^2}\right). \quad ∎\end{aligned}$$

It is well known that inputs that are multiples of $w$ tend to be very bad in terms of bank conflicts [50, 21, 11]. The following corollary shows a slightly stronger lower bound when $N$ is a multiple of $w^2$:

**Corollary 6.2.1** *For every sorted sequence $\mathcal{N}$, such that $|\mathcal{N}| = N \geq w^2$ and $N \equiv 0 \pmod{w^2}$, there exists a query set $\mathcal{Q}$, $|\mathcal{Q}| = Q \geq p$, for which*

$$\beta\mathcal{A}_s = \Omega\left(w \cdot \frac{Q}{P} \log \frac{N}{w}\right).$$

**Proof** If $N \equiv 0 \pmod{w^2}$, then all possible memory addresses that each thread might access in the first $r' = \log w + 1$ iterations reside in memory bank 0 (see Figure 6.4). Thus, there exists a set of queries that cause $w$-way bank conflicts starting from the iteration $r' = \log w + 1$ instead of iteration $r = 2 \log w + 1$ proven in Lemma 6.2.2. The rest of the proof is similar to the proofs of Lemma 6.2.2 and Theorem 6.2.1. ∎

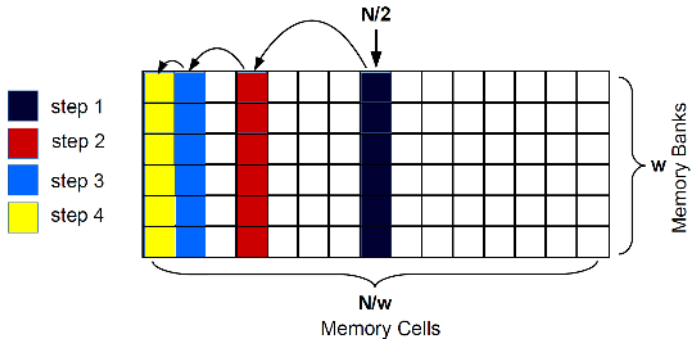


Figure 6.5: Illustration of the shared memory access pattern for stage 1 of the PBS-CF algorithm ($\delta \geq w$).

This tells us that, in the worst case, the access pattern of the PBS algorithm can result in a factor $w$ performance loss due to bank conflicts.

### 6.2.2 Conflict-Free PBS

To avoid the performance loss due to bank conflicts, we present a modified PBS algorithm, which we call *Parallel Binary Search - Conflict Free (PBS-CF)*, that is free of shared memory bank conflicts. To do so, we divide the PBS algorithm into two stages, and for each stage we design a solution to eliminate all bank conflicts. We define each stage based on the parameter $\delta$, which we define as the step distance at each iteration of the binary search. At each iteration, $\delta$ is decreased by half, until, after $\log N$ iterations, $\delta = 1$. *Stage 1* consists of the first $(\log N - \log w)$ iterations, during which $\delta \geq w$. *Stage 2* consists of the remaining iterations. The access pattern and reason for bank conflicts differ between the two stages, so we must develop a different conflict-free solution for each pattern.

For *Stage 1*, the PBS-CF algorithm assigns each thread a separate shared memory bank. This is accomplished by giving each thread an initial offset equal to its *threadID* within the warp. The resulting access pattern is illustrated in Figure 6.5. We note that to extend this approach to a general case for any size $N$, we must alter the $\delta$ calculation so that $\delta$ is a multiple of $w$ at every iteration of Stage 1. After the first $\log N - \log w$ iterations, however, it is impossible for $\delta$ to be a multiple of $w$, thus we must use a different method of avoid bank conflicts in Stage 2. To completely avoid bank conflicts, we perform a linear search of the final $w$ elements within the search space. While this is not work-efficient, we avoid all bank conflicts, making our entire PBS-CF algorithm bank conflict-fee. Thus, for this algorithm, the total number of shared memory accesses is

$$\mathcal{A}_s = \left( \frac{Q}{P} \cdot (\log N - \log w + w) \right).$$

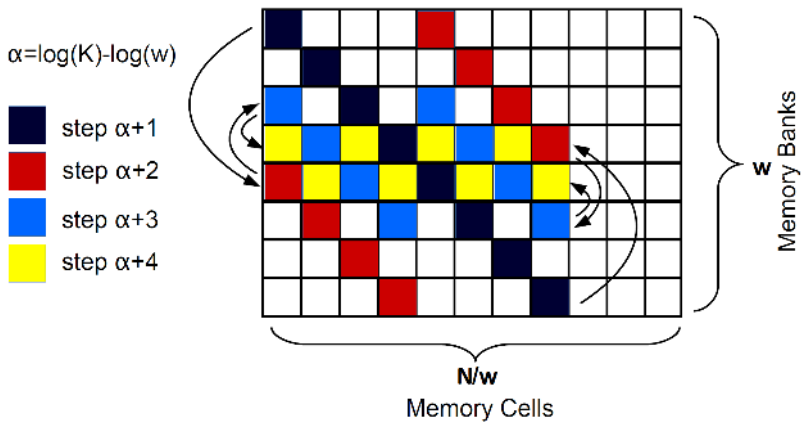


Figure 6.6: Illustration of the shared memory access pattern for stage 2 of the PBS-CL algorithm ($\delta < w$), for a worst-case example.

Since only shared memory accesses vary between different searching methods, we estimate the runtime of PBS-CF to be

$$\begin{aligned} \mathcal{T} &= \mathcal{T}_g + \mathcal{T}_s + L_{sync} \\ &= \frac{Q}{P} \cdot max\left(\frac{1}{\mathcal{B}_g}, \frac{L_g}{\mathcal{M}}\right) + \left(\frac{Q}{P} \cdot \left(\log \frac{N}{w} + w\right)\right) \cdot max\left(\frac{1}{\mathcal{B}_s}, \frac{L_g}{\mathcal{M}}\right) + L_{sync}, \end{aligned}$$

where $\mathcal{M} = \frac{M \cdot t}{N \cdot P}$.

### 6.2.3 Conflict-Limited PBS

Because the PBS-CF algorithm is completely conflict-free, it always requires the same number of memory accesses per query. However, during stage 2 it is not work optimal, requiring $O(w)$ memory accesses (and operations) to search $w$ elements. The PBS algorithm, on the other hand, is work optimal, requiring only $\Theta(\log w)$ iterations to search the last $w$ elements. In the worst case, however, it incurs $w$-way bank conflicts at each of these iterations, leading to $O(w \log w)$ total memory accesses, a factor $\Theta(\log w)$ more than PBS-CF. We present a hybrid algorithm *PBS-CL* (Parallel Binary Search - Conflict Limited) with the goal to minimize bank conflicts while remaining work optimal.

PBS-CL uses the same conflict-free stage 1 (the first $\log N - \log w$ iterations) as PBS-CF. However, it uses a hybrid approach for stage 2 that introduces few bank conflicts while keeping the number of iterations minimal $\Theta(\log w)$. Recall that, for the PBS-CF algorithm, each thread accesses a different memory bank during stage 1 (specifically, each thread accesses bank $i$ where $i \equiv threadId \pmod w$). Upon starting stage 2, the offset between each thread is preserved, causing

them to each access a different bank. If we then perform a binary search, with all threads utilizing the same $\delta$ value, we can preserve these offsets and reduce the number of bank conflicts while remaining work optimal, requiring only $\Theta(\log w)$ iterations.

Figure 6.6 illustrates which cells may be accessed at each iteration of stage 2 of the PBS-CL algorithm, in the *worst case*. For the first iteration of stage 2, $\delta = \frac{w}{2}$, and, since $w$ is a power of 2 and

$$(threadId + \frac{w}{2}) \equiv (threadId - \frac{w}{2}) \pmod{w},$$

there will be no conflicts (and thus only 1 parallel access). For the second iteration, however, there is potential for at most a 2-way conflict. For the third iteration, the worst case would have one 4-way conflict, and so on. Thus, as illustrated in Figure 6.6, the worst-case number of memory accesses for stage 2 is:

$$\sum_{i=0}^{\log w - 1} 2^i = w - 1$$

Thus, in the worst case, PBS-CL requires $O(\frac{Q}{P} \log \frac{N}{w} + \frac{Qw}{P})$ accesses, which is equivalent to PBS-CF. However, in the best case, PBS-CL will experience *no* bank conflicts, resulting in $O(\frac{Q}{P} \log N)$ accesses.

### 6.2.4 Empirical Performance Comparison

We implement the PBS, PBS-CF, and PBS-CL algorithms and evaluate their performance empirically on our three hardware platforms (details in Table 4.1). Since our GPUs have limited shared memory per SM, we use a relatively small search list ($N$), though we can use a large number of queries ($Q$). Figure 6.7 contains the execution times of our three implementations for varying $N$ sizes and 500 million random queries on the GIBSON platform. These results indicate that, for most $N$ sizes, PBS-CF has the highest execution time. However, as we increase $N$, the naive PBS performance degrades more quickly than our other two implementations. We attribute this to an increasing number of bank conflicts. PBS-CL attains the benefits of both of the other algorithms: it performs well when $N$ is small, yet it does not suffer from many bank conflicts and so performs well when $N$ is large as well. Across all of our experiments, PBS-CL achieves a maximum speedup of 2.98 over the naive PBS algorithm.

## 6.3 Conclusion

In this Chapter we considered the predecessor search problem on GPU architectures. Specifically, we looked at the batched predecessor search: given $N$ keys and $Q$ queries, find the predecessor of each query $q$ in the list of $N$ keys. We considered the cases when (i) $N$ is large and the keys

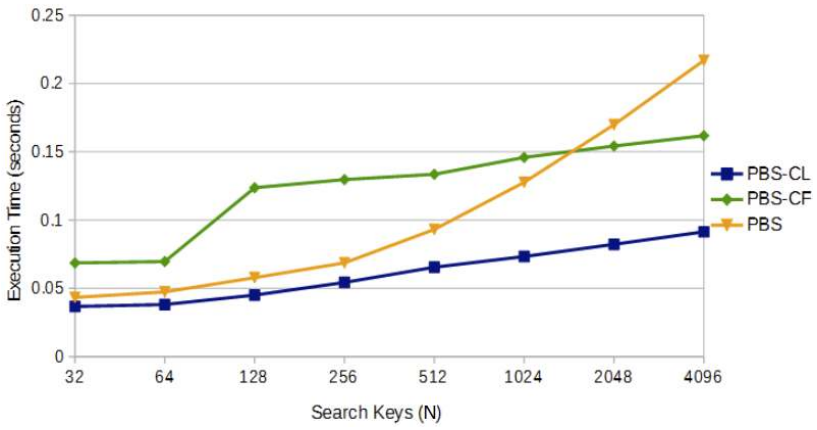


Figure 6.7: Execution time results of our three batched predecessor search implementations for varying numbers of search keys and $Q = 500M$ on GIBSON.

are stored in global memory and (ii) $N$ is small and the keys are stored in shared memory. In global memory, we looked at different memory layouts (sorted, BST, and B-tree) and analyzed how they impact the total number of memory accesses performed. Results indicate that the SPT model accurately estimate overall runtime, indicating that the major performance factors of GPUs are captured by the model.

Searching constitutes a broad class of problems, each with different difficulties and potential solutions. Many of these solutions rely on data structures, some of which are difficult to apply to many-core architectures. Thus, an interesting area of future work would be to consider other searching problems and, using the SPT model, design GPU-efficient data structures. We discuss specific data structures and search problems, as well as other directions of future work in Chapter 8.

# CHAPTER 7
# CASE STUDY: SORTING

Sorting is a primitive operation that is a building block of countless other algorithms. Many algorithms assume a sorted input and thus rely on sorting to provide consistent input to other algorithms. Sorting can be accomplished by a wide range of algorithms, however, for this work we focus on comparison-based sorting algorithms. In this section, we consider several state-of-the-art sorting algorithms for the GPU and analyze them with the SPT model, defined in Chapter 3 and instantiated for our GPU platforms in Chapter 4. First, in Section 7.1 we analyze state-of-the-art GPU-efficient library implementations of pairwise mergesort. We identify performance bottlenecks inherent to these algorithms and conclude that a multiway mergesort algorithm can alleviate these bottlenecks. Next, in Section 7.2 we look at the multiway mergesort algorithm developed with the AGPU model (discussed in Section 3.2.4) and presented by Koike and Sadakane [47]. While we do not have access to their code, our analysis indicates that their algorithm also suffers from performance bottlenecks. Thus, in Section 7.3 we introduce GPU-MMS, our GPU-efficient multiway mergesort algorithm that provides a series of improvements over the algorithm presented by Koike and Sadakane. Finally, in Section 7.4 we compare the empirical performance of GPU-MMS with several sorting implementations available with state-of-the-art GPU libraries.

## 7.1 Pairwise Mergesort

Thrust [34] and ModernGPU [11] (MGPU) currently provide the fastest comparison-based sorting implementations and are both based on the pairwise mergesort algorithm [17]. We focus our analysis on the the MGPU mergesort, although the analysis of the Thrust mergesort is similar, as these algorithms are designed with the same goals in mind: 1) achieving as much parallelism as possible, 2) performing only coalesced global memory accesses, and 3) reducing bank conflicts with heuristics. We note that both MGPU and Thrust are highly optimized with a likely significant impact on performance. However, since our analysis identifies both asymptotically dominating terms as well as their associated constant factors, we expect it to predict performance accurately.

### 7.1.1 MGPU algorithm overview

The latest MGPU (version 2.10) mergesort assigns a fixed number of elements, $E$, to each thread, regardless of input size $N$. MGPU uses $E = 11$ or $E = 15$, depending on the specific GPU hardware. Threads are grouped into tread-blocks of $b = 128$ threads each. The algorithm begins with the "base-case" by sorting $b \cdot E$ elements in shared memory using $b$ threads. This is accomplished by a pairwise mergesort in shared memory, using $\log(bE)$ merge rounds. At all subsequent merge

rounds, an increasing number of TBs work together on each pair of merge lists, while $E$ remains constant. At the final merge round, $\frac{N}{E}$ threads all work together on the same pair of merge lists, each of size $\frac{N}{2}$. To enable all threads to work together, MGPU relies on partitioning, requiring that the following three *phases* are performed at each merge round:

- Block-partition - for each TB, find partitions in two merge lists using the Mergepath [29] method,
- Thread-partition - load block partitions into shared memory and find partitions within it for each of the $b$ threads,
- Merge - each thread merges the elements in its respective partition of shared memory and writes the result back to global memory.

### 7.1.2 Algorithm Analysis

We first consider the number of kernels $|\mathcal{K}|$ that this algorithm uses, and see if there are any ways we can simplify analysis to avoid analyzing each kernel individually. As discussed above, the algorithm begins with a single kernel that sorts the *base case* of $bE$ elements. We call this first kernel $\mathcal{K}_{base}$. Each of the subsequent $\lceil \log NbE \rceil$ kernels perform the same series of steps: 1) find block-partitions for each TB, 2) load partition into shared memory, 3) find thread-partitions for each thread, 4) merge, and 5) write results back to global memory. Since each of these kernels performs the same series of steps and uses the same number of threads and $TBs$, we can consider them all as a single kernel by simply defining

$$\mathcal{A}_\phi = \sum_{i=1}^{\lceil \log \frac{N}{bE} \rceil} \mathcal{A}_{i,\phi}$$

for each $\phi \in \Phi$. This results in two kernels to analyze: the base case, $\mathcal{K}_{base}$, and the combined merge kernels, $\mathcal{K}_{merge}$. We analyze each kernel in the context of each operation $\phi \in \Phi$ separately. Recall from Chapter 4 that, for our GPU platforms, we say that $\Phi = \{g, s, r, y\}$, where $g$, $s$, $r$, and $y$ represent accessing global memory, accessing shared memory, performing register operations, and calling SYNCTHREADS, respectively.

#### Global memory accesses

For the base case kernel, $\mathcal{K}_{base}$, we can very simply determine the number of global memory accesses. Each element is read from global memory, sorted in blocks of $bE$ in shared memory, and written back to global memory. All accesses are coalesced and, since each element is read once and written

once,

$$\mathcal{A}_{base,g} = \frac{2N}{P}$$

For the merge kernel, $\mathcal{K}_{merge}$, we have to consider two phases of each merge round: finding block-partitions and merging. First, each TB (of which there are $\frac{N}{bE}$) performs a binary search global memory to find an independent partition using the method detailed by Green et al. [29]. Since the merge list sizes grow with each merge round, the number of accesses to find partitions increases. Thus, the total accesses required for partitioning for each TB, across all merge rounds, is:

$$\begin{aligned}\sum_{i=\lceil \log bE \rceil}^{\lceil \log N \rceil} i &= \frac{(\lceil \log^2 N \rceil - \lceil \log N \rceil) - (\lceil \log^2 bE \rceil - \lceil \log bE \rceil)}{2} \\ &\approx \frac{\lceil \log^2 \frac{N}{bE} \rceil - \lceil \log NbE \rceil}{2}.\end{aligned}$$

There are $\frac{N}{bE}$ thread-blocks, and accesses are not coalesced, resulting in $\frac{P}{w}$-way parallelism. Thus, the number of parallel global memory accesses for block-partition, across all merge rounds, is

$$\frac{1}{P/w} \cdot \frac{N}{bE} \cdot \left( \frac{\lceil \log^2 \frac{N}{bE} \rceil - \lceil \log NbE \rceil}{2} \right) = \frac{Nw \left( \lceil \log^2 \frac{N}{bE} \rceil - \lceil \log NbE \rceil \right)}{2bEP}$$

Once block-partitions are found, each thread-block loads its partition into shared memory, merges, and writes it back. These accesses are coalesced, for a total of $\frac{2N}{P}$ parallel accesses per merge round. There are $\log \frac{N}{bE}$ merge rounds, so the total accesses to merge, across all rounds, is

$$\frac{2N \lceil \log \frac{N}{bE} \rceil}{P}$$

Combining this with the block-partition phase, we get

$$\mathcal{A}_{merge,g} = \frac{2N \lceil \log \frac{N}{bE} \rceil}{P} + \frac{Nw \left( \lceil \log^2 \frac{N}{bE} \rceil - \lceil \log NbE \rceil \right)}{2bEP}.$$

**Shared memory accesses**

As with our analysis of global memory accesses, we analyze $\mathcal{K}_{base}$ and $\mathcal{K}_{merge}$ separately. At the base case, MGPU sorts blocks of $E$ elements in registers and merges them in shared memory to create sorted blocks of $bE$ elements. For each merged element, MGPU reads once and writes once. The associated access pattern is also data-dependent, so we include the bank conflict parameter,

i.e.,

$$\mathcal{A}_{base,s} = \beta\left(\frac{2N}{P}\lceil\log(b)\rceil\right)$$

At each round of $\mathcal{K}_{merge}$, $bE$ elements are loaded into shared memory and each of the $b$ threads finds an independent partition by performing binary search in shared memory. The search list is always size $bE$, requiring $\lceil\log bE\rceil$ shared memory accesses per thread. There are a total of $\frac{N}{E}$ threads, and each access is only a read, so the thread-partition step takes $\beta\frac{N}{PE}\lceil\log(bE)\rceil$ shared memory accesses. Each thread then merges its partitions in shared memory, spending 2 shared memory accesses per element (read and write), for a total of $\beta\frac{2N}{P}\left\lceil\log\frac{N}{bE}\right\rceil$ accesses. Combining these phases, we have

$$\mathcal{A}_{merge,s} \quad = \quad \beta_1\frac{N}{PE}\lceil\log(bE)\rceil + \beta_2\frac{2N}{P}\left\lceil\log\frac{N}{bE}\right\rceil$$

Note that we define separate $\beta$ values for each phase, as the access patterns differ. We empirically measure $\beta_1$ and $\beta_2$ in Section 7.1.3 and incorporate the resulting values into our SPT model to develop a runtime estimate.

**Register operations**

The exact number of register operations performed by an algorithm is highly dependent on compiler details and is difficult to determine. However, for MGPU, most work is performed in shared memory. We assume that every operation performed on elements in shared memory will be dominated shared memory access time, so we do not include them in $\mathcal{A}_r$. The only additional work performed in registers is the sorting of blocks of $E$ elements at the base case. This is done using a odd-even sorting network [46], requiring $\sim\frac{E^2}{2}$ "swaps," per block. Each swap takes 3 register operations, so we estimate that

$$\mathcal{A}_r = \frac{N}{PE}\cdot\frac{3E^2}{2} = \frac{3NE}{2P}$$

**Thread-block synchronizations**

One strength of the MGPU algorithm is the relatively few synchronizations that it performs. It is designed to maximize parallelism and minimize communication between threads. At each merge round, there are only two points that intra-TB synchronization is required: before and after loading shared memory. Therefore, we can easily determine that

$$\mathcal{A}_y = \frac{N}{P}\left(\left\lceil\log\frac{N}{bE}\right\rceil + 1\right)$$

### 7.1.3 Estimating runtime

To compute the runtime estimate for each of our hardware platforms, we first determine multiplicity, $\mathcal{M}$. Since MGPU mergesort uses few registers, multiplicity $\mathcal{M}$ is limited only by shared memory usage. Each thread stores $E$ elements in shared memory, and the GPU has a total shared memory of size $M$, thus $\mathcal{M} = \frac{M}{PE}$. ILP ($\mathcal{M}_I$), however, depends on optimizations and varies for each phase. MGPU mergesort employs a global memory read/write optimization proposed by [58] that effectively doubles $\mathcal{M}_I$. For all other operations, we say that $\mathcal{M}_I = 1$. All $\mathcal{M}$ values remain the same throughout MGPU execution, so we combine the $\mathcal{K}_{base}$ and $\mathcal{K}_{merge}$ calculations to get the following runtime estimate

$$
\begin{aligned}
\mathcal{T} &= \mathcal{T}_g + \mathcal{T}_s + \mathcal{T}_r + \mathcal{T}_y + |\mathcal{K}| \cdot L_{sync} \ , \\
\mathcal{T}_g &= \frac{2N}{P}\left(\left\lceil \log \frac{N}{bE} \right\rceil + \frac{w\left(\lceil \log^2 \frac{N}{bE} \rceil - \lceil \log NbE \rceil\right)}{4bE} + 1\right) max\left(\frac{1}{\mathcal{B}_g}, \frac{L_g}{2\mathcal{M}}\right) \ , \\
\mathcal{T}_s &= \frac{2N}{P}\left(\frac{\beta_1 \lceil \log (bE) \rceil}{E} + \beta_2 \left\lceil \log \frac{N}{E} \right\rceil\right) max\left(\frac{1}{\mathcal{B}_s}, \frac{L_s}{\mathcal{M}}\right) \ , \\
\mathcal{T}_r &= \frac{3NE}{2P} \cdot max\left(\frac{1}{\mathcal{B}_r}, \frac{L_r}{\mathcal{M}}\right) \ , \\
\mathcal{T}_y &= \frac{N}{P}\left(\left\lceil \log \frac{N}{bE} \right\rceil + 1\right) max\left(\frac{1}{\mathcal{B}_y}, \frac{L_y}{\mathcal{M}_o}\right) \ .
\end{aligned}
$$

where $|\mathcal{K}| = \lceil \log \frac{N}{bE} \rceil + 1$.

We determine $\beta_1 = 3.1$ and $\beta_2 = 2.2$ by using the nvprof tool [65] to empirically measure the average number of bank conflicts. These are simply estimates obtained with random inputs and certain permutations may result in more or fewer bank conflicts, though they are independent of hardware platform. We investigate this further in Section 7.4 by generating inputs that result in many bank conflicts.

### 7.1.4 Experimental Performance

Combining the above analysis with the parameters that we measured with our microbenchmarks, listed in Table 4.2, we can predict the execution time of MGPU mergesort on our three hardware platforms, estimate optimal parameter values (e.g., $E$), and identify bottlenecks. Figure 7.1 shows our throughput estimate as $E$ varies, on the ALGOPARC platform, along with the average throughput measured when running MGPU. These results indicate that our estimate correctly predicts the best value for $E$ to be 31, despite MGPU using $E = 15$ for Maxwell generation GPUs, like that on ALGOPARC. Our overall execution time estimate is also accurate, with an average error of only 7.48%.

Our performance analysis also allows us to determine the relative impact of each level of the

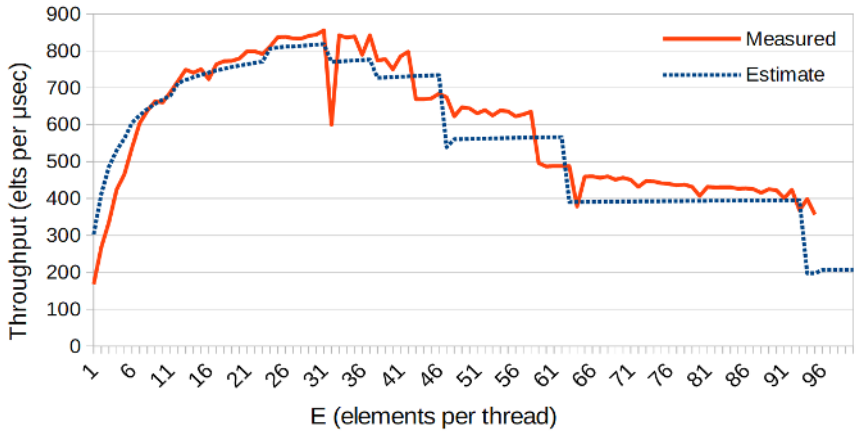


Figure 7.1: Estimated and measured MGPU mergesort throughput when varying $E$ (elements per thread) on ALGOPARC, for $N = 100M$.

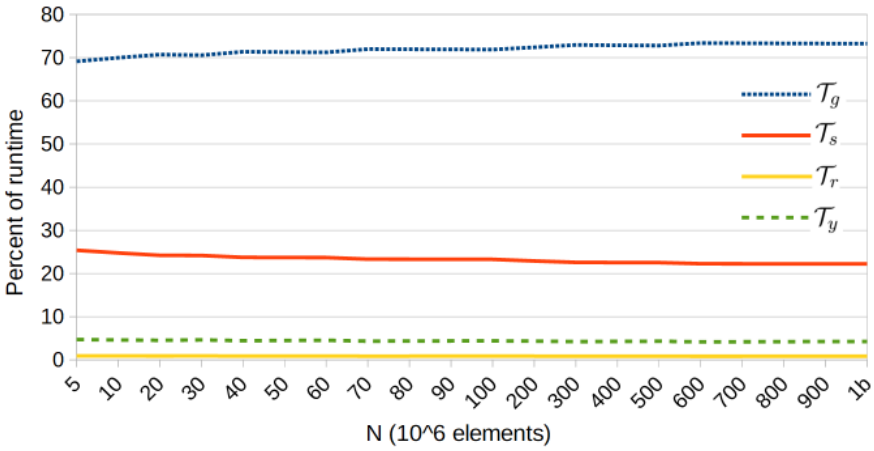


Figure 7.2: Estimated percentage of MGPU mergesort runtime due to each type of operation on ALGOPARC, for varying $N$ and $E = 31$.

memory hierarchy on overall performance. Figure 7.2 shows the estimated percentage of the total execution time due to global memory accesses, shared memory access, register operations, and synchronization, on ALGOPARC with $E = 31$. On all three platforms, our estimate indicates that between 65% and 80% of execution time is due to global memory accesses, which is not surprising since, for our hardware, $\mathcal{B}_s \approx 5\mathcal{B}_g$. This said, the remaining portion of execution time is attributed to shared memory accesses, indicating that improving shared memory usage can significantly improve performance. We measure $\beta$ for merging and partitioning phases to be 3.1 and 2.2, respectively, indicating that bank conflicts contribute roughly 15-20% to overall execution time. We conclude that MGPU suffers from two primary performance bottlenecks: global memory bandwidth and shared memory bank conflicts.

Using both $E = 15$ (used by MGPU) and $E = 31$ (found to be optimal for ALGOPARC), we evaluate performance as we vary $N$. Figure 7.3 contains both measured and predicted throughput values on ALGOPARC. These results indicate that, for inputs smaller than 20M elements, our model estimate is higher than the measured throughput. We attribute this to the fact that our performance estimate is determined by terms that asymptotically dominate and therefore ignores constants associated with low-order terms, which may have a significant impact on performance when $N$ is small. For large inputs, however, our performance model accurately estimates average MGPU sorting throughput. Additionally, as estimated, MGPU gains a significant performance increase when using $E = 31$ on ALGOPARC. We note that the "stepwise" performance profile is due to the ceiling function around the number of merge rounds. The model estimate incorporates ceilings functions and thus accurately predict this behavior.

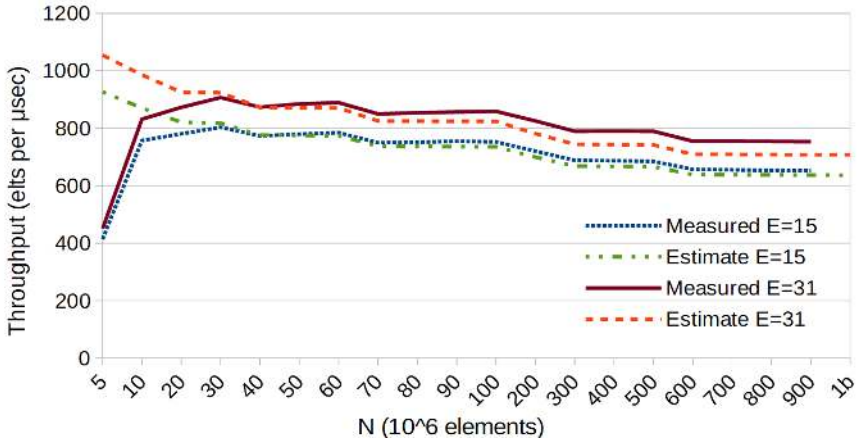


Figure 7.3: Estimated and measured MGPU mergesort throughput when varying $N$ on ALGOPARC, for both $E = 15$ and $E = 31$.

## 7.2 Koike and Sadakane's multiway mergesort

The results in the previous section show that the performance of a state-of-the-art comparison-based sorting algorithm for the GPU, MGPU, is limited in part by global memory bandwidth. The use of pairwise mergesort leads to $O(\log_2 N)$ merge rounds, where $N$ elements must be read from (and written to) global memory at each round. This naturally suggests the use of a *multiway* mergesort algorithm to reduce the number of merge rounds.

Like standard (pairwise) mergesort, multiway mergesort relies on repeated merge rounds. However, at each round, $K$ sorted lists are merged into a single list. In the external memory model [4], multiway mergesort achieves optimal I/O complexity when $K = \frac{M}{B}$, where $M$ is the size of internal memory and $B$ is the access block size [46]. To achieve this I/O efficiency, multiway mergesort must access only blocks of $B$ elements while storing only a limited number of elements in internal mem-

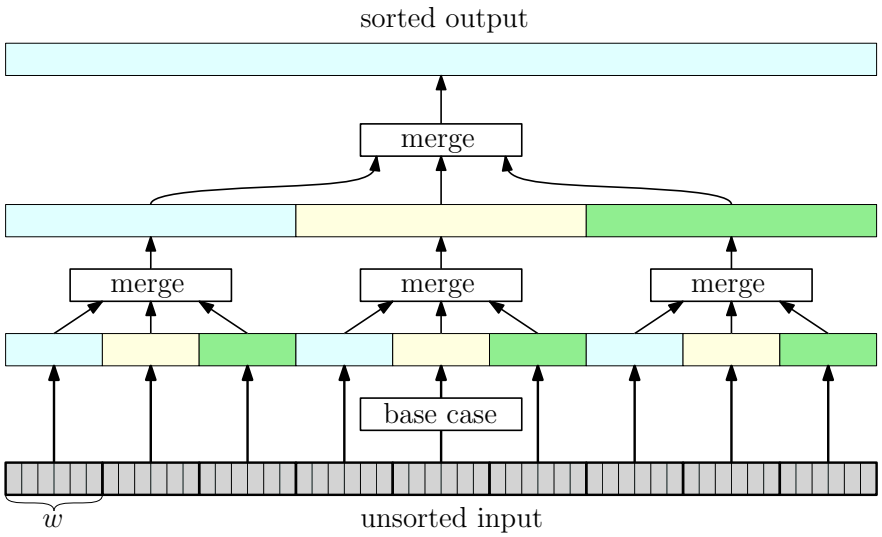


Figure 7.4: High-level illustration of the multiway mergesort algorithm used by Koike and Sadakane [47]. Example depicts multiway mergesort process if $K = 3$.

ory. Sequentially, this is accomplished by using a minHeap and an output buffer [46]. Multiway merging in parallel, however, is difficult and may require increased internal computation. Koike and Sadakane [47] present a multiway mergesort designed for GPUs. We start with an overview of their algorithm and a brief analysis to identify its potential performance bottlenecks.

### 7.2.1 Algorithm Overview

Using their AGPU model (discussed in Section 2.1.3, Koike and Sadakane [47] propose a GPU-efficient multiway mergesort to reduce global memory accesses. While we do not have access to their code, we analyze their multiway mergesort algorithm using the SPT model to evaluate the overall performance and identify if their approach overcomes the bottlenecks seen with MGPU. starts by sorting blocks of $w$ elements in internal memory ("base case"). Groups of $K$ sorted lists are then merged until the entire input is sorted. At each merge round, partitions are needed if there are not enough independent merge lists to satisfy all thread-blocks, where each thread-block contains $w = 32$ threads. Figure 7.4 illustrates a high-level view of this multiway mergesort algorithm. Since groups of $K$ lists are merged at each round, partitions are found across all $K$ lists using a search method based on the technique proposed by Hayashi et al. [32] that requires $O(K \log n)$ memory accesses per partition, where $n$ is the size of each list.

The merging of $K$ lists by $w$ threads is accomplished with the use of a variation of a minHeap structure. This structure stores $2w$ elements in sorted order at each node, where all elements satisfy the heap property (every element in a node $u$ is smaller than every element in its children $v$ and $w$). This guarantees that the smallest elements are stored in the root, allowing the $w$ threads to write

a block of $w$ elements to global memory. Re-filling the root is done by merging the elements of its children and placing the smallest resulting elements in the root. This leaves a non-full child node, and the process is repeated until a leaf node is reached. Each leaf node corresponds to one of the $K$ input lists, so $w$ new elements are read from the corresponding list. Koike and Sadakane store $2w$ elements in each node to guarantee correctness of their methods that empty and refill nodes.

### 7.2.2 Algorithm Analysis

As with our analysis of MGPU, we can simplify our analysis by noting that the multiplicity, $\mathcal{M}$, remains the same throughout each kernel call. Unlike MGPU, however, we now have $|\mathcal{K}| = \left\lceil \log_K \frac{N}{w} \right\rceil + 1$ kernel calls. As with MGPU, we analyze the base case $\mathcal{K}_{base}$ separate, while combining all merge kernels into $\mathcal{K}_{merge}$. We compute $\mathcal{A}_\phi$, for each $\phi \in \{g, s, r, y\}$.

#### Global memory accesses

As with MGPU mergesort, for the base case, each element is read once and written once, so

$$\mathcal{A}_{base,g} = \frac{2N}{P}.$$

For each merge round, thread-blocks find partitions (if there are fewer lists than thread-blocks) and then merge groups of $K$ lists, for a total of $\left\lceil \log_K \left(\frac{N}{w}\right) \right\rceil$ rounds. Koike and Sadakane use $w$ threads per thread-block, so the total number of thread-blocks is $\frac{P \cdot \mathcal{M}}{w}$. Thus, partitions are only needed when there are fewer than $\frac{KP\mathcal{M}}{w}$ lists, and only $\frac{P\mathcal{M}}{w}$ We note that this is only dependent on small hardware constants, and not $N$. We assume that this is negligible for a large $N$ and, to simplify analysis, we ignore the number of memory accesses due to partitioning. Merging requires that each element be read from and written to global memory once (i.e., $\frac{2N}{P}$ accesses) for each merge round, so

$$\mathcal{A}_{merge,g} = \frac{2N}{P} \left\lceil \log_K \frac{N}{w} \right\rceil$$

#### Shared memory accesses

Koike and Sadakane [47] use bitonic mergesort [46] to sort blocks of $w$ elements in this base case. While this avoids bank conflicts, is not work-efficient, requiring $2w \log^2 w$ shared memory accesses to sort a block of size $w$. There are no bank conflicts (i.e., $\beta = 1$), so

$$\mathcal{A}_{base,s} = \frac{N}{Pw} \cdot 2w \log^2 w = \frac{2N}{P} \log^2 w$$

Each thread-block merges groups of $K$ lists in shared memory using the variation of the minHeap described in the algorithm overview above. Every time a block of $w$ smallest elements is extracted,

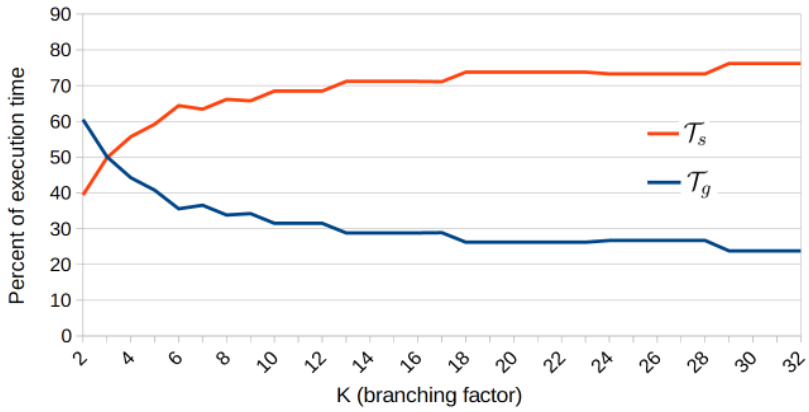


Figure 7.5: Estimated percentage of Koike and Sadakane's multiway mergesort is due to global memory and shared memory accesses, on ALGOPARC with $N = 2^{29}$ for varying $K$ values.

two nodes of $2w$ elements each are merged at each level of the heap using bitonic merge [46]. Thus, for every $w$ elements output from the heap, $(4w \log w \log K)$ shared memory accesses are performed, for a total of $\left(\frac{4N}{P} \log w \lceil \log K \rceil\right)$ accesses per merge round. All accesses are bank-conflict free and there are $\left\lceil \log_K \frac{N}{w} \right\rceil$ merge rounds, thus

$$\mathcal{A}_{merge,s} = \frac{4N}{P} \log w \lceil \log K \rceil \left\lceil \log_K \frac{N}{w} \right\rceil \approx \frac{4N}{P} (\log w \lceil \log N \rceil - 1) \ .$$

**Register operations**

Unlike MGPU, Koike and Sadakane [47] perform the base case in shared memory, rather than registers, so all work is done in global or shared memory. We, therefore, assume that all operations are dominated by memory access time and ignore register operation time, i.e., $\mathcal{T}_r = 0$.

**Thread-block synchronizations**

For all kernel calls, Koike and Sadakane [47] use $b = 32$ (i.e., each TB has 32 threads). Since each TB has only one warp, all threads are in SIMT lock-step. This removes the need for any calls to SYNCTHREADS, so $\mathcal{T}_y = 0$.

### 7.2.3 Estimating runtime

We are unable to obtain an implementation of the algorithm presented in [47], so we cannot accurately determine ILP ($\mathcal{M}_I$). We, therefore, assume $\mathcal{M}_I = 1$, since each operation of the bitonic merge network is dependent on the result of the previous operation. We can, however, estimate oversubscription ($\mathcal{M}_o$), since it is based on shared memory usage. Each warp works on its own heap structure, with each heap having $2K - 1$ nodes. Each node contains $2w$ elements,

so oversubscription is limited to $\mathcal{M}_o = \frac{M}{4KP-2P}$. If we assume that $\mathcal{M}_o$ and $\mathcal{M}_I$ are the same for $\mathcal{K}_{base}$ and $\mathcal{K}_{merge}$, we estimate the total runtime as

$$\begin{aligned}
\mathcal{T} &\approx \mathcal{T}_g + \mathcal{T}_s + |\mathcal{K}| L_{sync} \ , \\
\mathcal{T}_g &\approx \frac{2N}{P} \left( \left\lceil \log_K \frac{N}{w} \right\rceil + 1 \right) max \left( \frac{1}{\mathcal{B}_g}, \left\lceil \frac{L_g}{\frac{M}{4KP-2P}} \right\rceil \right) \ , \\
\mathcal{T}_s &\approx \frac{4N}{P} \left( \log w \lceil \log N \rceil + \frac{\log^2 w}{2} \right) max \left( \frac{1}{\mathcal{B}_s}, \left\lceil \frac{L_s}{\frac{M}{4KP-2P}} \right\rceil \right) \ .
\end{aligned}$$

where $M$ is the total shared memory available on the GPU.

While, without the code, we cannot measure the empirical performance of this algorithm, we can use the result of our performance estimate to determine potential bottlenecks. Using the hardware parameters in Table 4.2, we compare $\mathcal{T}_g$ and $\mathcal{T}_s$ estimates for each of our hardware platforms to determine which operations most impact performance. Figure 7.5 contains the estimated cost of $T_g$ and $T_s$ for varying $K$, using $N = 2^{29}$ 4-byte elements and the hardware parameters corresponding to ALGOPARC. Our performance estimate indicates that, when $K$ is very small, global and shared memory have similar contributions. However, as $K$ grows, the relative impact of $\mathcal{T}_s$ quickly increases. When $K = 32$, more than 75% of total execution time is due to shared memory accesses. Furthermore, on ALGOPARC, when $K \geq 3$ and $K \geq 6$, oversubscription decreases and access time to shared memory and global memory become bound by latency, respectively. Thus, in order to obtain the I/O-efficient benefits of multiway mergesort (i.e., large $K$), oversubscription (and therefore multiplicity) becomes reduced and performance degrades.

We conclude that, while the multiway mergesort of Koike and Sadakane can significantly reduce global memory accesses, the performance bottleneck then becomes the shared memory, in spite of the algorithm not incurring any shared memory bank conflicts. This is due to the use of a lot of shared memory per thread, which in turn limits multiplicity when $K$ becomes large, negating any benefits of a large value of $K$.

## 7.3 Improved multiway mergesort: GPU-MMS

Using the SPT model, we have identified performance bottlenecks that degrade performance of each of the GPU sorting algorithms analyzed thus far. The main bottlenecks for the MGPU mergesort are global memory bandwidth and shared memory bank conflicts. The multiway mergesort proposed by Koike and Sadakane [47] addresses these bottlenecks, however, its performance is limited by the large number of shared memory accesses and low multiplicity. In this section, we present GPU-MMS, our GPU-efficient multiway mergesort algorithm that avoids the performance bottlenecks of both of these algorithms.

### 7.3.1 Algorithm overview

We use the algorithm by Koike and Sadakane as a starting point, but present several improvements that address its performance bottlenecks. In a nutshell, GPU-MMS maximizes multiplicity and ILP while reducing both shared memory and global memory accesses. We present the following improvements over the algorithm of Koike and Sadakane to out-perform MGPU mergesort.

#### Improved heap structure

We first focus our design on reducing the shared memory usage to increase $\mathcal{M}$ and thereby improve performance for larger values of $K$. Recall that the heap used by Koike and Sadakane stores $2w$ elements at each node, requiring $4Kw - 2w$ elements in shared memory per heap. We present an improved heap structure, which we call a minBlockHeap, that requires half the shared memory, while still allowing all $w$ threads to work cooperatively and access only blocks of $w$ elements at a time from global memory. Unlike the heap of Koike and Sadakane, the minBlockHeap stores only $w$ elements at each node. Let $v$ be a node in our minBlockHeap, we denote the $i$-th element stored in $v$ as $v[i]$ (i.e., $v$ contains elements from $v[0]$ to $v[w-1]$). The elements in each node is sorted and all elements satisfy the heap property: for nodes $v$ and $u$, if $v$ is the parent of $u$, then $v[w-1] < u[0]$. We note that, since the elements in each node is sorted, this requirement implies that every element in $v$ is smaller than every element in $u$. Furthermore, if this property is satisfied, the root node contains the $w$ smallest elements in the minBlockHeap.

When merging, we write all elements of the root node out to global memory in a coalesced manner, leaving the root empty. We define the *fillEmptyNode* operation that fills an empty node (i.e., a node without any elements in its list). Consider $v$ to be an empty node with non-empty children $u$ and $x$. W.l.o.g. assume that $u[w-1] > x[w-1]$. The $fillEmptyNode(v)$ operation is performed as follows: merge the lists of $u$ and $x$, fill $v$ with the $w$ smallest elements, fill $u$ with the $w$ largest elements, and set $x$ as empty. Since, prior to merging, $u$ had the largest element ($u[w-1]$), its new largest element has not changed and the heap property holds for $u$. We continue down the tree by calling $fillEmptyNode(x)$ until we reach a leaf, which we fill by loading $w$ new elements from global memory. Figure 7.6 provides an example of this process on a minBlockHeap with $w = 4$.

This structure provides two benefits over the heap used by Koike and Sadakane. First, it reduces the shared memory used by each heap to $2Kw - w$ elements, effectively doubling the number of threads that can be used (i.e., $\mathcal{M}_o$), compared with the heap of Koike and Sadakane. Second, the number of elements merged is reduced from $4w$ and $2w$ at each level, cutting the amount of work to merge nodes in half.

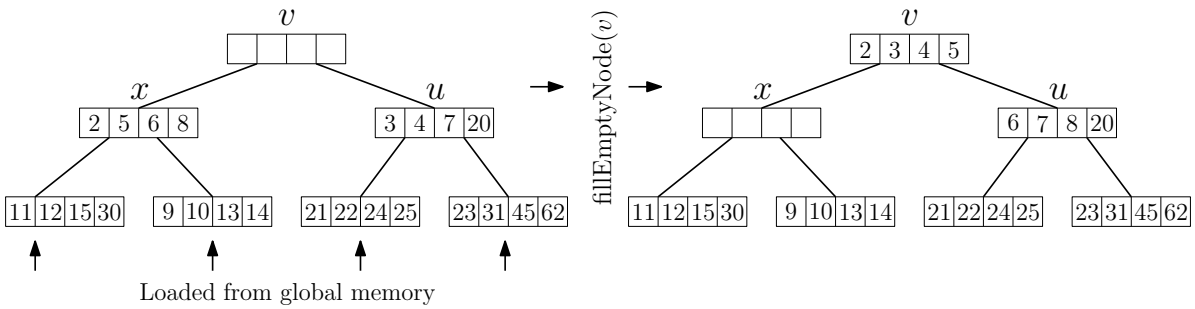


Figure 7.6: Example a minBlockHeap structure with $w = 4$. We perform a fillEmptyNode operation on node $v$ by merging its children, leaving $x$ empty.

### Merging in registers

A primary drawback of using a minBlockHeap type structure is that, since $w$ threads are merging pairs of $w$ elements, a work-efficient sequential merge cannot be used. [47] employ a bank-conflict-free bitonic merge network in shared memory, increasing shared memory accesses by a factor $\log w$. To reduce the cost of this work-inefficient merge, we develop a merge step for GPU-MMS that operates in *registers*. While this means that GPU-MMS is also work-inefficient, the extra work is done in low-latency registers that have higher peak bandwidth. We accomplish this register merge with the use of `__shfl()`, a hardware instruction that lets threads within a warp access each others' registers. CUDA [65] provides a *__shflxor()* command that lets thread $i$ access the register of thread $j$, where $j = i \oplus x$, for some bitmask $x$. We use this access pattern to efficiently implement a bitonic merge network.

### Independent merging

In addition to increasing $\mathcal{M}_o$ we increase $\mathcal{M}_I$ with the following optimization to the minBlockHeap. After we remove the smallest $w$ elements from the root node, all $w$ threads working on the heap identify the path of merge nodes from the root to the leaf. To do this, we simply descend through the heap, comparing the $(w-1)$-th element at each pair of nodes, and following the path of the node that contains the smallest. This gives us the path of $2 \cdot \log K$ nodes that need to be merged. While identifying this path, each thread loads 1 element each from each node on the path and its sibling, for a total of $2 \log K$ elements. Since each merge operation is independent, we interleave the operations, thereby increasing ILP ($\mathcal{M}_I$) by a factor $\log K$. This comes at the cost of using additional registers. However, this is not an issue on our hardware, so shared memory remains the primary factor limiting multiplicity.

#### Increased base case

MGPU mergesort sorts its base case of $bE$ elements using pairwise mergesort, though this results in bank conflicts, reducing performance. Koike and Sadakane [47] avoid bank conflicts in the base case by letting each thread sort $w$ elements within its own memory bank, resulting in smaller base case. We increase the base case by sorting $w^2$ items in shared memory by using the bank conflict-free algorithm based on the work of Afshani and Sitchinava [3]. We have each thread load $w$ elements into shared memory, thus creating a $w \times w$ matrix that is shared by $w$ threads. We then employ a variant of Shearsort [3] to sort this matrix as follows: each thread loads a row of $w$ elements into registers, sorts it internally (with a bitonic sorting network), and writes it back into shared memory in ascending or descending order, based on thread id. Each thread then loads a "column" of data into registers, sorts it, and writes it back. We repeat this $\log w$ times, sort rows once more, and then write the $w \times w$ matrix back into global memory. By carefully defining indices, we can avoid shared memory bank conflicts when accessing *both* rows and columns. While this requires more internal work than the base case in [47], our base case is larger ($w^2$ elts), thus reducing the number of merge rounds.

With this method, our base case sorts blocks of $w^2$ elements and has complexities:

$$\begin{aligned}
\mathcal{A}_{base,g} &= \frac{2N}{P} \\
\mathcal{A}_{base,s} &= \frac{N}{Pw} 4w \log w = \frac{4N}{P} \log w \\
\mathcal{A}_{base,r} &= \frac{N}{Pw} 2w \log^2 w \log w = \frac{2N}{P} \log^3 w
\end{aligned}$$

### 7.3.2 Performance Analysis

Aside from shifting $O(\log w)$ work from shared memory to registers, GPU-MMS does not significantly improve asymptotic performance over Koike and Sadakane's mergesort. However, GPU-MMS improves many of the constant factors that have a practical impact on execution time.

#### Global memory accesses

The improvements that GPU-MMS provides over Koike and Sadakane's multiway mergesort results in a significant reduction in the number of shared memory accesses, while increasing $\mathcal{M}_o$ and $\mathcal{M}_I$, which we identified as bottlenecks. However, we do not reduce the number of global memory accesses. Thus, for GPU-MMS,

$$\begin{aligned}
\mathcal{A}_{base,g} &= \frac{2N}{P} \\
\mathcal{A}_{merge,g} &= \frac{2N}{P} \left\lceil \log_K \frac{N}{w} \right\rceil .
\end{aligned}$$

Since $\mathcal{M}$ remains the same for both kernels,

$$\mathcal{A}_g = \mathcal{A}_{base,g} + \mathcal{A}_{merge,g} = \frac{2N}{P}\left(\left\lceil \log_K \frac{N}{w} \right\rceil + 1\right)$$

**Shared memory accesses**

The new base case that GPU-MMS uses is presented in Section 7.3.1, along with a discussion of the number of shared memory accesses it requires, i.e.,

$$\mathcal{A}_{base,s} = \frac{N}{Pw} 4w \log w = \frac{4N}{P} \log w.$$

In addition to the new base case, the GPU-MMS algorithm uses the minBlockHeap structure and performing merging in registers, significantly reducing the number of shared memory accesses, compared with Koike and Sadakane's mergesort. As discussed in Section 7.3.1, during merging, for every block of $w$ elements written out from the root of the minBlockHeap, 2 nodes from every level of the tree are loaded into registers from shared memory, merged in registers, and written back to the heap. There are $\log K$ levels in the heap, each node contains $w$ elements, and each element is read once and written once, so, for every $w$ elements merged, we perform $4w \log K$ shared memory accesses. There are a total of $\lceil \log_K \frac{N}{w^2} \rceil$ merge rounds, thus

$$\mathcal{A}_{merge,s} = \frac{N}{Pw} 4w \lceil \log K \rceil \left\lceil \log_K \frac{N}{w^2} \right\rceil \approx \frac{4N}{P} \left\lceil \log \frac{N}{w^2} \right\rceil,$$

and, as we have no bank conflicts and $\mathcal{M}$ remains constant,

$$\mathcal{A}_s = \mathcal{A}_{base,s} + \mathcal{A}_{merge,s} = \frac{4N}{P} \log w + \frac{4N}{P} \left\lceil \log \frac{N}{w^2} \right\rceil \approx \frac{4N}{P} \left\lceil \log \frac{N}{w} \right\rceil,$$

which we note is $O(\frac{N}{P} \log N)$, which is the lower bound for sorting in the CREW PRAM model [17].

**Register operations**

From Section 7.3.1, we have that

$$\mathcal{A}_{base,r} \quad = \quad \frac{N}{Pw} 2w \log^2 w \log w = \frac{2N}{P} \log^3 w.$$

Furthermore, by performing merging in registers using the $\_\_shfl$ operations, we reduce the number of shared memory accesses, while increasing the number of register operations. For each block of $w$ elements merged (i.e., written from the root of a minBlockHeap), we merge pairs of nodes in registers using a bitonic merge network. Each node contains $w$ elements, so the bitonic merge network performs a total of $2w \log w$ swaps. We use $w$ threads, and each swap consists of 3

operations: `__shfl()`, `min()`, and `max()`, for a total of $6 \log w$ register operations per thread. For every $w$ elements written to global memory, we merge $\log K$ pairs of nodes, so across all $\left\lceil \log_K \frac{N}{w^2} \right\rceil$ merge rounds,

$$\mathcal{A}_{merge,r} = \frac{6N}{P} \left\lceil \log \frac{N}{w^2} \right\rceil \log w$$

**Thread-block synchronizations**

The GPU-MMS algorithm uses only 32 threads per TB, so all threads in a TB are implicitly synchronized after every operation. Therefore, SYNCTHREADS is not necessary and we do not use it, i.e., $\mathcal{A}_y = 0$.

### 7.3.3 Estimating runtime

Since our heap uses less memory, $\mathcal{M} = \frac{M}{2KP-P}$ (a factor 2 increase compared with Koike and Sadakane [47]). Furthermore, by pipelining merges, we are able to increase $\mathcal{M}_I$ for both shared memory and register accesses by a factor $\log K$. Thus, combining the analysis of each type of operation for the GPU-MMS algorithm, we have

$$\begin{aligned}
\mathcal{T} &= \mathcal{T}_g + \mathcal{T}_s + \mathcal{T}_r + |\mathcal{K}| L_{sync} \\
\mathcal{T}_g &= \frac{2N}{P} \left( \left\lceil \log_K \frac{N}{w^2} \right\rceil + 1 \right) \cdot max \left( \frac{1}{\mathcal{B}_g}, \frac{L_g}{\mathcal{M}} \right), \\
\mathcal{T}_s &= \frac{4N}{P} \left\lceil \log \frac{N}{w} \right\rceil \cdot max \left( \frac{1}{\mathcal{B}_s}, \frac{L_s}{\mathcal{M} \log K} \right), \\
\mathcal{T}_r &= \frac{6N}{P} \left( \left\lceil \log \frac{N}{w^2} \right\rceil \log w + \frac{\log^3 w}{3} \right) \cdot max \left( \frac{1}{\mathcal{B}_r}, \frac{L_r}{\mathcal{M} \log K} \right).
\end{aligned}$$

where $|\mathcal{K}| = \left\lceil \log_K \frac{N}{w^2} \right\rceil + 1$.

Recall from Section 7.2 that we estimated that Koike and Sadakane's multiway mergesort saw performance loss when $K > 4$ on our hardware platforms. This was because operations because latency-bound due to multiplicity falling too low. The GPU-MMS algorithm, however, has improved both $\mathcal{M}_o$ and $\mathcal{M}_I$. Therefore, we compare our runtime estimate for each algorithm for varying $K$ values. Figure 7.7 shows our performance estimate and measured execution time of our GPU-MMS implementation, on ALGOPARC, for varying $K$ values and $N = 2^{28}$. Results indicate that, as expected, GPU-MMS achieves significantly increase overall performance, especially when $K$ is larger. However, even with the increased $\mathcal{M}_o$ and $\mathcal{M}_I$, the ideal $K$ value is still either 8 or 16 on ALGOPARC. On our other two hardware platforms, both our analysis and empirical results indicate that $K = 8$ leads to the best performance. Thus, in all experiments that follow we use $K = 16$, $K = 8$, and $K = 8$ on ALGOPARC, GIBSON, and UHHPC, respectively, unless otherwise noted.

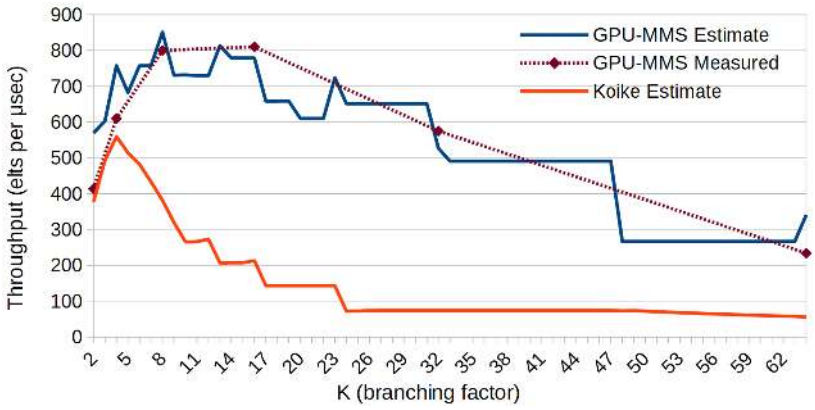


Figure 7.7: Estimated throughput of GPU-MMS, compared with the measured performance on ALGOPARC with $N = 2^{28}$ for a range of $K$ values. Estimated throughput also shown for Koike and Sadakane' multiway mergesort [47].

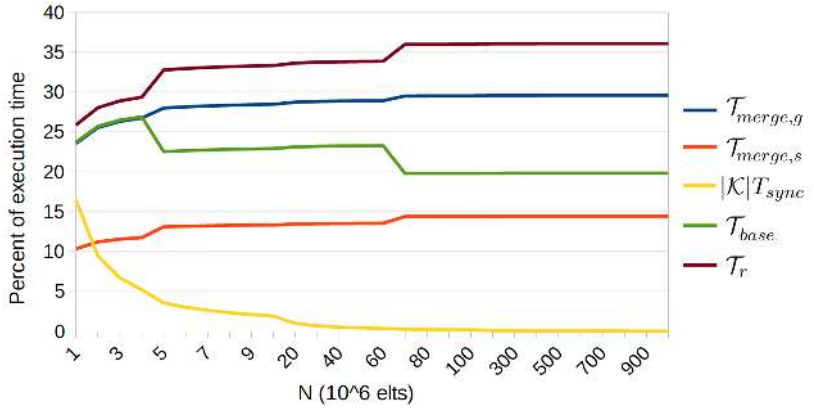


Figure 7.8: Estimated percentage of GPU-MMS execution time that is due each aspect of the algorithm, on ALGOPARC with $K = 16$, for varying input sizes $N$.

To verify that our GPU-MMS algorithm does not suffer from the performance bottlenecks that we see with MGPU mergesort and the Koike and Sadakane's multiway mergesort [47], we look at the estimated execution time of each component of GPU-MMS. Figure 7.8 shows the percentage of estimated GPU-MMS execution time of several different types of operations. We see that, unlike the MGPU mergesort and Koike and Sadakane's mergesort, no single type of operation dominates execution time. Furthermore, register operations make up a significant portion of overall execution time, indicating that, unlike MGPU and Koike and Sadakane's mergesort, GPU-MMS performance is not bound memory accesses.

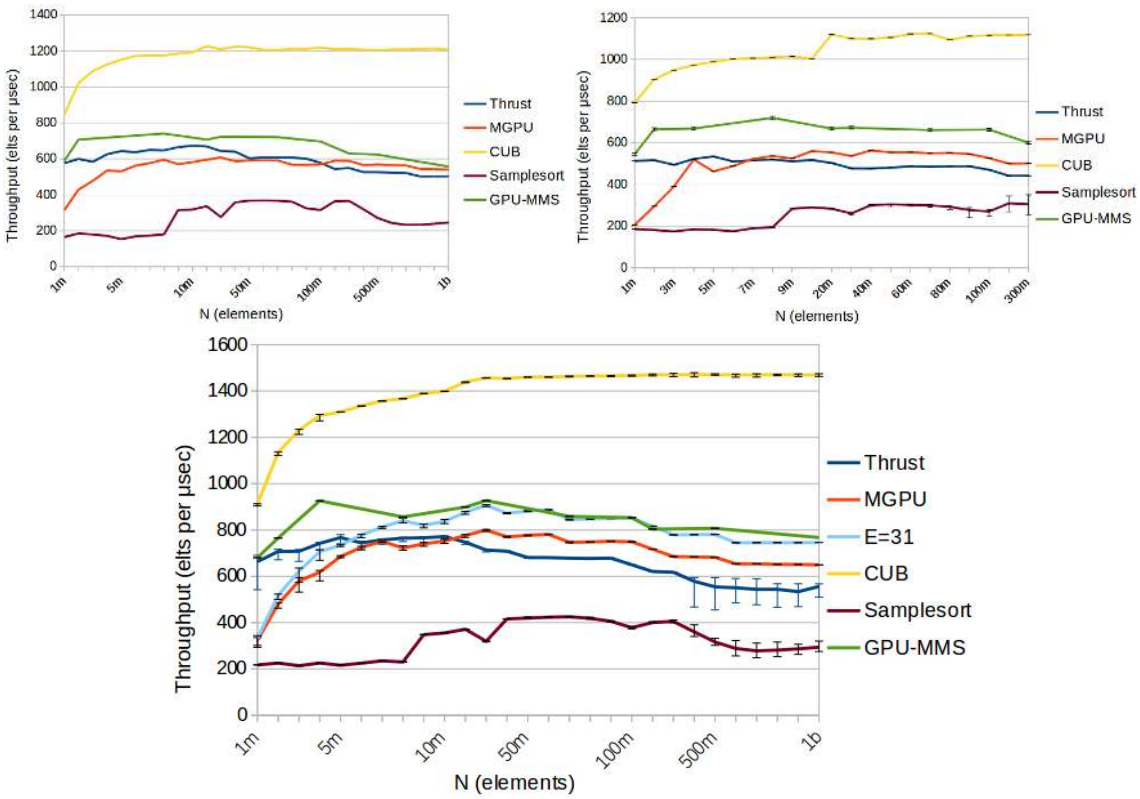


Figure 7.9: Comparison of average throughput for each sorting algorithm on inputs of random integers on ALGOPARC (middle), GIBSON (left), and UHHPC (right).

## 7.4 Comparison of Empirical Performance

The analysis in the previous section indicates that our GPU-MMS algorithm provides key advantages over the Thrust and MGPU pairwise mergesorts, as well as the multiway mergesort presented by Koike and Sadakane [47]. In this section, we compare GPU-MMS performance with three leading GPU sorting libraries: Thrust 1.8.1 [34], MGPU 2.10 [11], and CUB 1.6.4 [57]. As discussed in Section 7.1, Thrust and MGPU provide two of the fastest comparison-based sorts available for the GPU. CUB provides the highest-performing radix sort. Although CUB is not a comparison-based sort, and is therefore limited to sorting keys that can be represented by small integers, we include it in some of our experiments for completeness. We also include results from an I/O-efficient samplesort implementation [50] in some of our experiments.

### Sorting random inputs

Figure 7.9 shows the throughput achieved by each sorting algorithm when applied to fully random inputs of increasing sizes, using 4-byte integers, on each of our hardware platforms. These results show that GPU-MMS out-performs all other comparison-based sorting algorithms for most input

sizes. As expected, CUB, being a radix sort, achieves much higher throughput across all input sizes. The samplesort implementation performs significantly worse than all its competitors across the board. On ALGOPARC, we see that, even when we use the improved $E = 31$, GPU-MMS out-performs MGPU for most input sets and when $E = 15$, it out-performs MGPU across all input sizes by 32.27% on average. We obtain similar results on our other hardware platforms, with GPU-MMS out-performing the next-fastest comparison-based sort across input sizes on GIBSON and UHHPC by 23.26% and 11.48% on average, respectively.

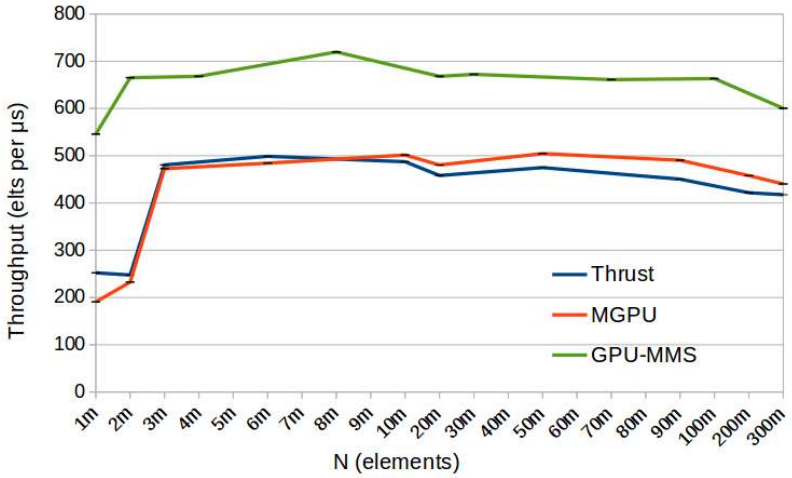


Figure 7.10: Average throughput vs. input size of *conflict-heavy* inputs on the GIBSON platform.

### Impact of Bank Conflicts

One additional feature of GPU-MMS is that it is free of shared memory bank conflicts, for any input. MGPU and Thrust, however, have memory access patterns that depend on the input. Since the memory access patterns of MGPU and Thrust are deterministic, we are able to generate an input permutation that will cause these algorithms to incur large numbers of bank conflicts. Figure 7.10 contains the results of running GPU-MMS, Thrust, and MGPU on such a *conflict-heavy* input on the GIBSON platform. These results indicate that our conflict-heavy inputs significantly degrade Thrust and MGPU performance, while GPU-MMS (being conflict-free) exhibits worst-case performance regardless of input. We repeat this experiment across all of our hardware platforms and find that GPU-MMS out-performs Thrust and MGPU by an average of 67.14% on conflict heavy inputs.

## 7.5 Conclusion

In this Chapter, we applied the SPT model to the analysis of comparison-based sorting algorithms for the GPU. We considered several state-of-the-art GPU sorting algorithms and, using the SPT model, identified key performance bottlenecks. We developed GPU-MMS, a GPU-efficient multiway mergesort algorithm that mitigates or avoids the performance bottlenecks that we identified for other algorithms. Empirical results indicate that GPU-MMS outperforms the fastest current library comparison-based sorting implementations by up to 67% on our GPU platforms.

An important remaining open problem is to be able to analytically determine the number of bank conflicts ($\beta$) in algorithms with data-dependent access patterns. This problem is a very challenging one. Our current approach is to estimate this quantity by running an implementation of an algorithm and measuring the number of bank conflicts using the profiler. Observe that this approach does not undermine the above application, because the number of bank conflicts is a function of the algorithm, the input, and the number of memory banks in the hardware. Since the latter changes only every decade or so (and is currently the same for all current GPUs) the algorithm can be run on a single hardware to estimate $\beta$ and will remain the same on other hardware. We further discuss this and other areas of future work in Chapter 8.

# CHAPTER 8
# CONCLUSIONS

It is clear that, as the push toward computer systems that deliver higher computational throughput continues, parallelism is becoming increasingly important. Even today, many supercomputers [6, 26] and community clusters use many-core coprocessors to meet the processing needs of users. Unfortunately, many users of such systems lack the expertise to make efficient use of them, causing computational resources to be wasted. This underscores the importance of developing parallel algorithms and easy-to-use libraries that can efficiently run on a wide range of architectures. Currently, however, there is simply too much variance among architectures and they are too complex to fall under a unified model that can be used to develop algorithms that suit them all.

In this work, we attempted to address this issue, although we concede that the model we present is necessarily complex due to the intricacies of these architectures. Nevertheless, in this work we have shown that our general model can accurately estimate the runtime of a given algorithm on any of our GPU hardware platforms. Furthermore, the unifying features that we identify among a range of high-performance architectures, is a step in the direction of better understanding these systems. While these complex processing architectures may never be accurately modeled by something as simple as RAM or PRAM models, the SPT model, presented in this dissertation, reduces the optimization space and underscores just a few parameters that may help us develop algorithms that make more efficient use of them. We demonstrated this fact on three of the most fundamental problems in computer science.

## 8.1 Many-core Architectures

In this dissertation, we instantiated the SPT model on our three GPU platforms, with hardware details in Table 4.2. However, the general SPT model that we defined in Chapter 3 is general and applicable to a range of many-core architectures. While GPUs are some of the most common many-core systems in use today, there exist other such architectures and more variations may be available in the future. Processing units such as Intel Xeon Phi [38], AMD GPUs, and custom many-core units in modern supercomputers are all architectures that differ significantly from the NVIDIA GPUs considered in this dissertation, though the SPT model may be applicable. Furthermore, different generations of NVIDIA GPUs have significant architectural changes, especially to the memory systems. For example, the Pascal generation architectures introduced significant changes to the global and shared memory latency and bandwidth values. Thus, the trade-off between different types of operations (e.g., memory accesses) depends on the architecture itself. Different architectures may also provide new hardware primitives for specific operations. For example, the new Volta generation of NVIDIA GPUs provides a hardware unit that performs a $4 \times 4$ matrix-

matrix multiplication operation in a single clock cycle. This type of hardware unit can have a major impact on the performance of many algorithms and it may be beneficial to develop algorithms that make efficient use of these units. Thus, when we instantiate the SPT model on such a hardware platform, we may include this matrix multiplication operation in the list of relevant operations, $\Phi$, that we consider during algorithm analysis.

The generality of the SPT model makes it able to incorporate unique or unusual features of a particular architecture without having to develop new models. To demonstrate this, we would like to apply the SPT model to a range of other architectures and evaluate its accuracy in estimating runtime for a series of algorithms. In particular, it would be interesting to evaluate the efficacy of the model on architectures with unique features, such as the $4 \times 4$ matrix multiplication hardware unit available with Volta GPUs.

## 8.2 Developing Efficient Algorithms

The case studies we considered for this work range from simple, compute bound problems such as matrix-matrix multiplication to more complex, data-dependent algorithms such as pairwise merge-sort. However, there are a broad range of other problems that lack algorithms that are efficient for many-core architectures, such as GPUs. Many existing algorithms to these problems are not easily applicable to many-core architectures because they (i) are inherently sequential, (ii) require a high degree of communication between processors, or (iii) rely on data structures that use a large amount of memory.

Many problems have efficiently solutions that are inherently sequential. For example, the single-source shortest path (SSSP) problem can be efficiently solved with Dijkstra's algorithm [17] in $O(E + V \log V)$ on a graph with $V$ vertices and $E$ edges. However, this algorithm is inherently sequential due to its use of a priority queue and the requirement of $V$ synchronization steps. A series of other algorithms that solve SSSP are more applicable to many-core architectures [60, 80, 18], though there are complex performance tradeoffs that must be considered. For future work, we propose using the SPT model to analyze these algorithms and identify the interplay between these performance tradeoffs.

Communication-bound algorithms include a number of solutions to fundamental problems. The fast Fourier transform (FFT) [17] is one such communication-bound algorithm that is one of the most frequently used algorithms today. While libraries, such as CUFFT [64], provide efficient many-core implementations of these algorithms, the large number of synchronizations may significantly degrade performance. As demonstrated in the case studies of this dissertation, the SPT model identifies such performance bottlenecks and aides us in developing algorithms that avoid them. Therefore, a promising avenue of future work would be to analyze such communication-bound algorithms to identify improvements or develop new algorithms that are better suited to many-core architectures.

As discussed in Chapter 3, using too many resources (e.g., memory) per thread can reduce multiplicity and degrade performance on many-core architectures. Therefore, algorithms that use large data structures or require large amounts of additional memory may not achieve peak performance. This makes it difficult to efficiently use techniques such as dynamic programming, which may be essential to efficiently solving a given problem. As a result, many algorithms developed for many-core systems avoid more complex structures and instead use more brute-force techniques that may result in sub-optimal performance. It is important to identify such instances and provide alternative solutions using analytical tools such as the SPT model.

## 8.3 Future Work

The SPT model is a general performance model for many-core architectures and therefore has many applications. In this section, we identify some particularly interesting areas of future work. In the previous sections we discussed more broad directions of future work, so we now focus on specific algorithms or applications that are particularly interesting or promising.

### 8.3.1 Linear Algebra

While our analysis showed that the state-of-the-art matrix multiplication library implementation is well-suited for our GPU platforms, it relies on the naive GEMM algorithm. To our knowledge, thus far only one work has attempted to implement a fast matrix multiplication algorithm on GPUs [51], and they use the simple variant that achieves $O(n^{2.81})$ work complexity [75]. Thus, an interesting direction of future work would be to use our SPT model to analyze more advanced fast GEMM algorithms to determine if they can be efficiently implemented on many-core systems. This poses a particular challenge, as it is not known how to perform these "fast" matrix multiplication algorithms I/O efficiently.

As discussed in Section 8.1, the NVIDIA Volta generation GPUs have a $4 \times 4$ matrix multiplication operation as a hardware primitive. Thus, an interesting area of future work would be analyze the impact of such an operation on overall GEMM performance, and determine if our SPT model accurate captures such an impact. Furthermore, many other linear algebra algorithms make use of matrix multiplication. Thus, a potentially fruitful direction of future work may be to identify other linear algebra algorithms that can benefit from this hardware primitive, or to develop new algorithms that can make use of it.

### 8.3.2 Searching

Searching is a broad area and many potentially useful data structures have been developed to facilitate efficient searching. Persistent search trees [22, 10, 8] are one class of data structures that are useful in solving a range of problems. However, persistent structures typically require a large

amount of additional space [10], making them potentially unsuitable for many-core architectures. Thus, identifying methods of reducing the space complexity of such structures may be useful in developing data structures that are efficient for many-core architectures.

In the case studies presented in this dissertation, we determined that global memory accesses are frequently a factor that dominates overall runtime. Therefore, I/O-efficient algorithms may perform well on many-core architectures like GPUs. Searching can be performed I/O-efficiently using buffer tree structures [7, 72], although they have yet to be efficiently implemented on GPUs or other many-core architectures.

### 8.3.3 Sorting

Sorting has been extensively studied and the state-of-the-art library implementations perform well, as seen in Chapter 7. However, in this dissertation we only considered comparison-based sorting algorithms. Thus, we would like to use our SPT model to analyze the CUB [57] and Thrust [34] radix sort implementation to determine if there are any ways of improving performance. Additionally, analyzing GPU-MMS performance on other many-core architectures may lead to further improvements.

# BIBLIOGRAPHY


[1] clBLAS. `https://github.com/clMathLibraries/clBLAS`. Accessed: 2018-02-10.

[2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack J. Dongarra. Performance, design, and autotuning of batched GEMM for gpus. In *Proceedings of the 31st International Conference, ISC High Performance*, pages 21–38, 2016.

[3] Peyman Afshani and Nodari Sitchinava. Sorting and permuting without bank conflicts on gpus. In *Proc. of ESA*, pages 13–25, 2015.

[4] A. Aggarwal and J.S. Vitter. The input/output coplexity of sorting and related problems. *Commun. ACM*, 31(11), 1988.

[5] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11, 1990.

[6] Tatsumi Aoyama, Ken-Ichi Ishikawa, Yasuyuki Kimura, Hideo Matsufuru, Atsushi Sato, Tomohiro Suzuki, and Sunao Torii. First application of lattice qcd to pezy-sc processor. *Procedia Comput. Sci.*, 80(C):1418–1427, June 2016.

[7] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, September 2003.

[8] Lars Arge, Andrew Danner, and Sha-Mayn Teh. I/o-efficient point location using persistent b-trees. *J. Exp. Algorithmics*, 8, December 2003.

[9] Lars Arge, Michael Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proc. of SPAA*, pages 235–246, Munich, Germany, 6 2008.

[10] Mikhail J. Atallah, Michael T. Goodrich, and S. Rao Kosaraju. Parallel algorithms for evaluating sequences of set-manipulation operations. *J. ACM*, 41(6):1049–1088, 1994.

[11] Sean Baxter. Modern GPU, 2013.

[12] Kyle Berney, Henri Casanova, Alyssa Higuchi, Ben Karsin, and Nodari Sitchinava. Beyond binary search: parallel in-place construction of implicit search tree layouts. In *Proc. of IPDPS*, 2018.

[13] Nicola Bombieri, Federico Busato, and Franco Fummi. A fine-grained performance model for gpu architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

[14] Bryan Catanzaro, Alexander Keller, and Michael Garland. A decomposition for in-place matrix transposition. In *Proc. of PPoPP*, 2 2014.

[15] Douglas Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11:121–137, 1979.

[16] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 1–6, 1987.

[17] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[18] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 349–359, 2014.

[19] Frank Dehne and Hamidreza Zaboli. Deterministic sample sort for GPUs. volume abs/1002.4464, 2010.

[20] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. Accelerating numerical dense linear algebra calculations with gpus. *Numerical Computations with GPUs*, pages 1–26, 2014.

[21] Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manfedelli. Fast scan algorithms on graphics processors. In *ICS*, 2008.

[22] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 109–121, 1986.

[23] P. Enfedaque, F. Auli-Llinas, and J.C. Moure. Implementation of the DWT in a GPU through a register-based strategy. *IEEE Trans. PDS*, PP(99), 2014.

[24] N. Fauzia, L. N. Pouchet, and P. Sadayappan. Characterizing and enhancing global memory data coalescing on GPUs. In *Proc. of CGO*, pages 12–22, 2015.

[25] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[26] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fang-Li Qiao, Wei Zhao, Xunqiang Yin, Chaofeng Hou, Chenglong Zhang, Wei Ge, Jian Zhang, Yangang Wang, Chunbo Zhou, and Guangwen Yang. The sunway taihulight supercomputer: system and applications. *SCIENCE CHINA Information Sciences*, 59(7):072001:1–072001:16, 2016.

[27] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The queue-read queue-write pram model: Accounting for contention in parallel algorithms. *SIAM J. Comput.*, 28(2):733–769, February 1999.

[28] Oded Green, Robert McColl, and David A. Bader. GPU merge path: a GPU merging algorithm. In *Proc. of ICS*, pages 331–340, 2012.

[29] Oded Green, Saher Odeh, and Yitzhak Birk. Merge path - A visually intuitive approach to parallel merging. *CoRR*, abs/1406.2628, 2014.

[30] Azzam Haidar, Chongxiao Cao, Asim Yarkhan, Piotr Luszczek, Stanimire Tomov, Khairul Kabir, and Jack Dongarra. Unified Development for Mixed Multi-GPU and Multi-coprocessor Environments Using a Lightweight Runtime Environment. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 491–500, 2014.

[31] Azzam Haidar, Jack Dongarra, Khairul Kabir, Mark Gates, Piotr Luszczek, Stanimire Tomov, and Yulu Jia. Hpc programming on intel many-integrated-core hardware with magma port to xeon phi. *Scientific Programming*, 23, 2015.

[32] Tatsuya Hayashi, Koji Nakano, and Stephan Olariu. Weighted and unweighted selection algorithms for k sorted sequences. In *Proceedings of the 8th International Symposium on Algorithms and Computation*, pages 52–61, London, UK, UK, 1997. Springer-Verlag.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[34] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.7.0.

[35] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proc. of the 36th Intl. Symp. on Computer Architecture (ISCA)*, pages 152–153, 2009.

[36] C. Iancu, S. Hofmeyr, F. Blagojevic, and Y. Zheng. Oversubscription on multicore processors. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.

[37] Joseph JaJa. *Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.

[38] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[39] K. Kaczmarski. Experimental $B^+$-tree for GPU. In *Proc. of ADBIS*, volume 2, pages 232–241, Rome, Italy, 2011.

[40] K. Kaczmarski. B-tree optimized for GPGPU. In *Proc. of OTM 2012*, pages 843–854, Rome, Italy, 9 2012.

[41] Ben Karsin, Henri Casanova, and Nodari Sitchinava. Efficient batched predecessor search in shared memory on GPUs. In *Proc. of HiPC*, pages 335–344, 2015.

[42] Paul-Virak Khuong and Pat Morin. Array layouts for comparison-based searching. *ACM Journal of Experimental Algorithmics*, 22(1), 2017.

[43] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldeway, V. Lee, S. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proc. of SIGMOD*, Indianapolis, Indiana, USA, 6 2010.

[44] David B. Kirk. *Programming Massively Parallel Processors*. Elsevier Science, 2012.

[45] JS. Kirtzic, O. Daescu, and TX. Richardson. A parallel algorithm development model for the GPU architecture. In *Proc. of Intl Conf. on Parallel and Distributed Processing Techniques and Applications*, 2012.

[46] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[47] A. Koike and K. Sadakane. A novel computational model for GPUs with applications to efficient algorithms. *International Journal of Networking and Computing*, 5(1):26–60, 2015.

[48] K. Kothapalli, R. Mukherjee, S. Rehman, S. Patidar, P. Narayanan, and K. Srinathan. A performance prediction model for the cuda gpgpu. In *Proc. of the Intl. Conf. on High-Performance Computing (HiPC)*, 2009.

[49] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. Autotuning GEMM kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, November 2012.

[50] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *Proc. of IPDPS*, pages 1–10, April 2010.

[51] Junjie Li, Sanjay Ranka, and Sartaj Sahni. Strassen's matrix multiplication on gpus. In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, pages 157–164, 2011.

[52] Yinan Li, Jack Dongarra, and Stanimire Tomov. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 2009 International Conference on Computational Science, ICCS'09*, Baton Roube, LA, May 25-27 2009. Springer.

[53] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28, 2008.

[54] L. Ma, K. Agarwal, and R.D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202–215, 2014.

[55] L. Ma, R.d. Chamberlain, and K. Agarwal. Analysis of classic algorithms on GPUs. In *Proc. of HPCS*, 2014.

[56] L. Ma, R.D. Chamberlain, and K. Agarwal. Performance modeling for highly-threaded many-core GPUs. In *Proc. of ASAP*, 2014.

[57] Duane Merrill. Cub: Cuda unbound, 2015.

[58] Duane Merrill and Andrew Grimshaw. Parallel Scan for Stream Architectures. Technical Report CS2009-14, Department of Computer Science, University of Virginia, 2009.

[59] Bruce Merry. A performance comparison of sort and scan libraries for GPUs. *Parallel Processing Letters*, 4, 2016.

[60] Ulrich Meyer and Peter Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, pages 393–404, 1998.

[61] K. Nakano. Simple memory machine models for GPUs. In *Proc. of IPDPSW*, pages 794–803, May 2012.

[62] K. Nakano. The hierarchical memory machine model for GPUs. In *Proc. of IPDPSW*, pages 591–600, 5 2013.

[63] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for fermi graphics processing units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, November 2010.

[64] NVIDIA. CUDA CUFFT library, 2007.

[65] NVIDIA. CUDA programming guide 7.0, 2015.

[66] NVIDIA. Nsight, 2015.

[67] NVIDIA. CUDA cuBLAS library, 2018.

[68] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. of PPoPP*, pages 73–82. ACM, 2008.

[69] Shubhabrata Sengupta, Mark Harris, and Michael Garland. Efficient parallel scan algorithms for GPUs. NVIDIA Technical Report NVR-2008-003, 12 2008.

[70] A. Shekhar. Parallel binary search trees for rapid IP lookup using graphic processors. In *Proc. of IMKE*, pages 176–179, 12 2013.

[71] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 11–22, 2012.

[72] Nodari Sitchinava and Norbert Zeh. A parallel buffer tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 214–223, 2012.

[73] Jyothish Soman, Kishore Kothapalli, and P. J. Narayanan. Discrete range searching primitive for the GPU and its applications. *J. Exp. Algorithmics*, 17:4.5:4.1–4.5:4.17, October 2012.

[74] Gilbert Strang. *Linear algebra and its applications*. Thomson, Brooks/Cole, Belmont, CA, 2006.

[75] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13(4):354–356, August 1969.

[76] T. Abdelrahman T. Han. hiCUDA: High-level GPGPU programming. In *Proc. of TPDS*, volume 22, pages 78–90, 2010.

[77] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, pages 1–8. IEEE Computer Society, 2010.

[78] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[79] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 31:1–31:11, 2008.

[80] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. *SIGPLAN Not.*, 51(8):11:1–11:12, February 2016.

[81] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 887–898, 2012.

[82] H. Wong. Demystifying GPU microarchitecture through microbenchmarking. In *Proc. of ISPASS*, pages 235–246, 3 2010.

[83] Yao Zhan and J.D. Owens. A quantitative performance analysis mdoel for gpu architectures. In *Proc. of HPCA*, pages 382–393, 2011.