Winter 12-15-2014

Modeling Algorithm Performance on Highly-threaded Many-core Architectures

Lin Ma
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science
Department of Computer Science and Engineering

Dissertation Examination Committee:
Kunal Agrawal, Chair
Roger Chamberlain, Co-Chair
James Buckley
Jeremy Buhler
Tao Ju

Modeling Algorithm Performance on Highly-threaded Many-core Architectures

by
Lin Ma

A dissertation presented to the
Graduate School of Arts and Sciences
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

December 2014
Saint Louis, Missouri

Table of Contents

List of Figures

List of Tables

Acknowledgments

The fact that I survived the past 6 years, defended this dissertation and finally graduated as a computer science Ph.D. would not be true without the many people for whom I am grateful.

First of all, I would like to express my sincere gratitude to the two great advisors: Dr. Roger Chamberlain and Dr. Kunal Agrawal. Roger is the most patient and friendly person I have ever met. Since picked up into his research group, I was greatly impacted by his research advice, systematic insight, and optimistic attitude towards both work and life. Thank him for giving me so much freedom and flexibility so that I can manage my time with a good balance, and finish a successful internship in NVIDIA Research. I was kept being moved by his trust and kindness on me, as well as his warm smiles. Every tiny little piece of my achievement would not have been achieved without his strong support and encouragement. I would also thank Kunal to lead me and shape my thinking in theoretical parallel computing. She impressed and influenced me by her critical thinking, straightforward writing, and hardworking. I'm a firm believer that these influences, as life-long treasure, would significantly benefit and train me as a capable researcher. It's my great honor to be her first graduated Ph.D. in her academic life.

I would like to thank my committee James Buckley, Jeremy Buhler, and Tao Ju for their incisive comments and suggestions, and my colleagues Peng Li, Chengjie Wu, Jonathan Beard, Michael Hall, Shobana Padmanabhan, Joseph M. Lancaster, Joseph G. Wingbermuehle, and

Arpith C. Jacob for their kind help and insightful discussion. I truly appreciate the patience and help from all the staff members in the Department of Computer Science and Engineering over these years. Myrna Harbison, Lauren Huffman, Kelli Eckman, Jayme Moehle, Madeline Hawkins, Andrea Levy, and Sharon Matlock make my Ph.D. study and life here really easier and more colorful than it is supposed to be. Special thanks also go out to Dr. Ken Wong and Dr. Jonathan Turner for their kind encouragement and strong endorsement.

I would also like to acknowledge the NSF Grant CNS-0905368, CNS-0931693, the NIH Award R42 HG003225, and Exegy, Inc. for financially funding my research.

Finally on a personal note, I take this opportunity to express my deepest gratitude to my parents for their unconditional love and consistent support. It is their thoughtful care and encouragement that helped me going through those hard times over all these years. Part of my life goal is to make their lives happier and happier than ever before. It is to them that I dedicate this work and degree.

Lin Ma

Washington University in Saint Louis
December 2014

ABSTRACT OF THE DISSERTATION

Modeling Algorithm Performance on Highly-threaded Many-core Architectures

by
Lin Ma
Doctor of Philosophy in Computer Science
Washington University in St. Louis, 2014
Professor Kunal Agrawal, Chair
Professor Roger Chamberlain, Co-Chair

The rapid growth of data processing required in various arenas of computation over the past decades necessitates extensive use of parallel computing engines. Among those, highly-threaded many-core machines, such as GPUs have become increasingly popular for accelerating a diverse range of data-intensive applications. They feature a large number of hardware threads with low-overhead context switches to hide the memory access latencies and therefore provide high computational throughput. However, understanding and harnessing such machines places great challenges on algorithm designers and performance tuners due to the complex interaction of threads and hierarchical memory subsystems of these machines. The achieved performance jointly depends on the parallelism exploited by the algorithm, the effectiveness of latency hiding, and the utilization of multiprocessors (occupancy). Contemporary work tries to model the performance of GPUs from various aspects with different emphasis and granularity. However, no model considers all of these factors together at the same time.

This dissertation presents an analytical framework that jointly addresses parallelism, latency-hiding, and occupancy for both theoretical and empirical performance analysis of algorithms on highly-threaded many-core machines so that it can guide both algorithm design and performance tuning. In particular, this framework not only helps to explore and reduce

the configuration space for tuning kernel execution on GPUs, but also reflects performance bottlenecks and predicts how the runtime will trend as the problem and other parameters scale. The main contribution of the dissertation is a pair of analytical models with one focusing on higher-level asymptotic algorithm performance on GPUs and the other one emphasizing lower-level details about scheduling and runtime configuration. Based on the two models, we have conducted extensive analysis of a large set of algorithms. Two analysis provides interesting results and explains previously unexplained data. In addition, the two models are further bridged and combined as a consistent framework. The framework is able to provide an end-to-end methodology for algorithm design, evaluation, comparison, implementation, and prediction of real runtime on GPUs fairly accurately.

To demonstrate the viability of our methods, the models are validated through data from implementations of a variety of classic algorithms, including hashing, Bloom filters, all-pairs shortest path, matrix multiplication, FFT, merge sort, list ranking, string matching via suffix tree/array, etc. We evaluate the models' performance across a wide spectrum of parameters, data values, and machines. The results indicate that the models can be effectively used for algorithm performance analysis and runtime prediction on highly-threaded many-core machines.

Chapter 1

Introduction

General-purpose computation has irreversibly stepped into the parallel era, along with radically increased volumes of data, new algorithms, and emerging powerful parallel hardware platforms. Thanks to the development of new algorithms and computer systems, parallel processing not only can solve problems faster, it can also handle larger problems than is possible for sequential machines. However, as is commonly accepted, not every algorithm runs well on every parallel machine. We attribute the performance to the joint effect of algorithms and underlying architectures.

Abreast with traditional parallel architectures, such as shared memory machines and multi-cores, highly-threaded, many-core devices such as GPUs have gained popularity in the last decade; both NVIDIA and AMD manufacture general purpose GPUs that fall in this category. The important distinction between these machines and traditional multi-core machines is that these devices provide a large number of cores that support a massive numbers of hardware threads with low-overhead context switching between them; this fast context-switch mechanism is used to hide the memory access latency of transferring data from slow large (and often global) memory to fast, small (and typically local) memory.

Over the years, various models have been designed to capture the most important aspects of architectures and algorithms that dominate the effect on performance for a diverse range of sequential and parallel machines. The most fundamental model that is used to analyze sequential algorithms is the Random Access Machine (RAM) model [4], which we teach undergraduates in their first algorithms class. This model assumes that all operations, including memory accesses, take unit time. While this model is a good predictor of performance on computationally intensive programs, it does not properly capture the important characteristics of the memory hierarchy of modern machines and the nonuniform costs of accessing memory. There are a number of other models that consider the memory access costs of sequential algorithms in different ways [1, 2, 7, 8, 52, 126, 153]. For parallel computing, the analogue for the RAM model is the Parallel Random Access Machine (PRAM) model [51], and there is a large body of work describing and analyzing algorithms in the PRAM model [79, 152]. In the PRAM model, the algorithm's complexity is analyzed in terms of its *work* — the time taken by the algorithm on 1 processor, and *span* (also called *depth* and *critical-path length*) — the time taken by the algorithm on an infinite number of processors. Given a machine with P processors, a PRAM algorithm with work W and span S completes in $\max(W/P, S)$ time. The PRAM model also ignores the vagaries of the memory hierarchy and assumes that each memory access takes unit time. For modern machines, however, this assumption seldom holds. All these models fail to capture the nature of highly-threaded many-core machines.

There is no generally accepted theoretical model suitable for all architectures. Algorithm design and analysis tends to be agnostic to architectures and neutral to the number of processors so as to be general and universal. Unfortunately, this approach is insufficient for complex parallel machines, because while these machines provide substantial parallelism and efficiency, they impose important constraints on programs running on them. Failure

to respect those constraints will likely result in downgraded performance. Attempts to understand algorithms' performance on modern machines have been extensively made during the past decades. Researchers have designed various models that capture memory hierarchies for various types of modern machines such as distributed memory machines [44, 147, 155], shared memory machines and multi-cores [11, 19, 20, 36, 41], or the combination of the two [38, 39]. However, there is limited literature unveiling this relation on highly-threaded many-core machines such as GPUs [64, 84, 116]. This becomes our motivation of successfully bridging algorithms and highly-threaded many-core machines, so that people can be enabled to predict and control the tradeoffs towards overall optimal performance.

We consider the general problem of how to understand, analyze, and ultimately optimize algorithm performance on highly-threaded many-core machines and hopefully guide designers or programmers to develop solutions with limited pitfalls and high performance. This is a broad problem, in part because not every type of application manifests similar performance patterns on many-core machines. Performance of some applications heavily relies on the behavior of the memory subsystem. While other applications require fewer memory transactions but more computation, in which case the performance is mostly subject to the capability of the computing engine. In this dissertation, distinct from existing approaches, we formalize a set of analytical models to reflect the performance of arbitrary algorithms. A performance analysis framework is designed to bridge the gap of understanding between algorithms and performance on such machines, consisting of a theoretical asymptotic model — *Threaded Many-core Memory (TMM)* model and an empirical prediction model — *Calibrated Performance Model*. Based on the insight gained from the performance analysis framework, we also explore an end-to-end methodology for algorithm design, evaluation, comparison, implementation, and prediction of real runtime on many-core GPUs accurately.

1.1 Examples of Highly-threaded Many-core Architectures

The most common instantiation of highly-threaded many-core architectures include NVIDIA GPUs, AMD/ATI GPUs, and YarcData uRiKA system. Although each of these machines has its own distinctive features, there are some common properties that categorize them as highly-threaded many-core machines. Those architectures typically consist of a number of *core groups*, each containing a number of processors (or cores) [1], a fixed number of registers, and a fixed quantity of fast local on-chip memory shared within a core group. A large slow global memory is shared by all the core groups. Registers and local on-chip memory are the fastest to access, while accessing the global memory many potentially take 100s of cycles. Data is transferred from slow to fast memory in *chunks* in order to reduce long-latency memory transfers and achieve high bandwidth; instead of just transferring one word at a time, the hardware tries to transfer a large number of words during a memory transfer. The chunk can either be a cache line from hardware managed caches, or an explicitly-managed combined read from multiple threads.

On NVIDIA GPUs, a number of streaming multiprocessors (core groups in our terminology) share the same global memory residing on the device. On each of these streaming multiprocessors, there are a number of CUDA cores[2] that share a fixed number of registers and on-chip (fast) memory shared among the cores of the streaming multiprocessor. Accessing the off-chip global memory usually takes 20 to 40 times more clock cycles than accessing the on-chip shared memory/L1 cache [118]. The streaming multiprocessor creates, manages,

[1] A core group can also have a single core.

[2] CUDA (aka Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA.

schedules, and executes threads in groups of 32 parallel threads called *warps*. When a warp executes an instruction that accesses global memory, it *coalesces* the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. Accesses are fully coalesced as long as all threads in a warp are organized such that consecutive 32-bit words are accessed by consecutive thread IDs. A fast hardware-supported context-switching mechanism enables a large number of threads simultaneously in flight. The threads can be manually organized and managed in thread blocks. Each thread block consists of a number of warps. One or more thread blocks will be scheduled on a streaming multiprocessor, depending on the usage of on-chip resources per block.

AMD/ATI GPUs consist of multiple Single-Instruction-Multiple-Data (SIMD) computation engines as core groups. Each of these compute engines accommodates a number of Thread Processors (TP) and Local Data Store (LDS) shared by all the thread processors associated with the SIMD engine. Each thread processor possesses ALUs called Stream Cores (SC), and is arranged as a five-way or four-way Very Long Instruction Word (VLIW) processor depending on different device families. Each of these thread processors executes a single instruction across each lane for each of a block of 16 work-items. The instruction is repeated over four cycles to make the 64-element vector called a *wavefront*. Taking Cypress, the codename of Radeon HD5800 series GPUs, as an example, the architecture is composed of 20 SIMD computation engines. In each SIMD engine, there are 16 thread processors and a 32 KB local data store. Low context-switch threading is well supported, and every 64 threads are grouped into a wavefront executing the same instruction in lockstep.

The uRiKA system from YarcData is also a good example of such machines. Based on fundamental assumption from Smith et al. [10] about the nature of the computations this

processor was going to run, it is a purpose-built appliance for real-time graph analytics featuring graph-optimized hardware that provides up to 512 terabytes of global shared memory, massively-multithreaded graph processors (named Threadstorm) supporting 128 threads/processor, and highly scalable I/O. There can be up to 65,000 threads in a 512 processor system and over 1 million threads at the maximum system size of 8192 processors, so that the latencies are hidden by accommodating so many remote memory references in flight. The processor's instruction execution hardware essentially does a context switch every instruction cycle, finding the next thread that is ready to issue an instruction into the execution pipeline. This suggests that the memory access width is 1 on these machines. Threads do not share anything, as the Threadstorm processor has 128 hardware copies of the register set, program counter, stack pointer, etc., necessary to hold the current state of one software thread that is executing on the processor. But different than the two GPU architectures above, it has only one core on-chip.

1.2 Research Questions

While superficially, highly-threaded many-core machines are shared memory machines, their characteristics are very different from traditional multicore or multiprocessor shared memory machines. The most important distinction between multi-cores and highly-threaded many-core machines is the number of threads per core. On multi-core machines, context switch cost is high, and most models nominally assume that only one (or a small constant number of) thread(s) are running on each machine and this thread blocks when there is a memory access. Therefore, many existing models consider the number of memory transfers from slow memory to fast memory as a performance measure, and algorithms are designed to minimize

these, since memory transfers take a significant amount of time. In contrast, highly-threaded many-core machines are explicitly designed to have a large number of threads per core and a fast context switching mechanism. They are explicitly designed to hide memory latency; if a thread stalls on a memory operation, some other thread can be scheduled in its place. In principle, the number of memory transfers does not matter as long as there are enough threads to hide their latency. Therefore, if there are enough threads, we should, in principle, be able to use PRAM algorithms on GPUs, since we can ignore the effect of memory transfers which is exactly what the PRAM model does. However, the number of threads required to reach the point where one gets PRAM performance depends on both the algorithm and the hardware.

To motivate this enterprise and to understand the importance of high thread counts on many-core machines, let us consider a simple application that performs Bloom filter set membership tests on an input stream of biosequence data [105] on GPUs. The problem is embarrassingly parallel, each set membership test is independent of every other membership test. Fig. 1.1 shows the performance of this application, varying the number of threads per processor core, for two distinct GPUs. For both GPUs, the pattern is quite similar, at low thread counts, the performance increases (roughly linearly) with the number of threads, up until a transition region, after which the performance no longer increases with increasing thread count. While the location of the transition region is different for distinct GPU models, this general pattern is found in many applications. Once sufficient threads are present, the PRAM model adequately describes the performance of the application and increasing the number of threads no longer helps.

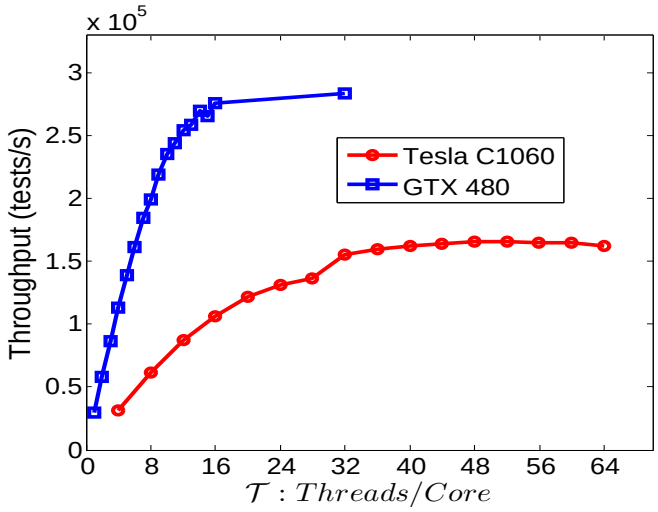


Figure 1.1: Throughput of Bloom filter algorithm for set membership testing on biosequence data. Performance (in membership tests per second) is plotted vs. number of threads per processor both for a Tesla C1060 and a GTX 480 GPU.

Since no highly-threaded many-core machine allows an infinite number of threads, and different thread/block counts may result in a huge search space of diversified performance, it is important to understand:

1. How many threads does a particular algorithm need to achieve PRAM performance asymptotically?

2. How does an algorithm perform and compare when it has fewer threads than required to get PRAM performance?

3. How does the variation of thread counts and thread block counts influence the scheduling, and therefore the real performance?

4. How can the real performance be predicted or calibrated for arbitrary runtime configurations?

For the first two problems, we design a high-level theoretical model (TMM model), which should abstract away the details of particular implementations so as to be applicable to many instantiations of these machines, while still being particular enough to model the performance of algorithms on these machines with reasonable accuracy. Basically it will characterize properties of algorithms and machines, capture how well the latencies are hidden by a given number of threads for a particular algorithm, and reveal how the performance of algorithms compares with asymptotically changing problem size and hardware parameters. For the last two problems, we develop a calibrated performance model, which, quite differently, plays with real throughput rather than asymptotic performance by incorporating the impact of hardware resources, real scheduling, and execution configuration. It will not tell you which algorithm is more efficient among candidates, but it will reflect the calibrated performance curve or pattern however you twist the execution configuration for the selected algorithm (or application) and machine being used.

1.3 Methodology for Performance Modeling

Performance analysis relies upon models that represent underlying assumptions; if a model does not capture the important aspects of target machines and programs, then the analysis is not predictive of real performance. Therefore, failure to capture unique features of highly-threaded many-core machines, especially the interaction between memory and the number of threads, will lead to misleading model. Given the heterogeneity and complexity of emerging many-core architectures relative to previous architectures, most well-known models for parallel architectures typically do not fit highly-threaded many-core machines. Researchers and developers need a way to bridge between algorithms and their possible performance on

such machines so as to find key factors, understand their correlation, define a unified metric to represent, measure, and predict the performance accurately and make decisions wisely. Conceptually, our modeling process consists of 3 steps as followed.

1.3.1 Find Key Factors of Performance

It's not too hard to implement an algorithm on a highly-threaded many-core machine, like a GPU, but it is much more difficult to get it to run efficiently. The crux of the problem is that few models or tools can precisely pinpoint the weakness of an algorithm in terms of latency hiding and the efficiency of the program execution in terms of processor scheduling, figure out the cause of the under-valued performance, and meticulously track the cause back to various key factors as a systematic methodology. Our performance analysis framework is designed to achieve this goal in a bottom-up approach. Basically, one needs to encompass the following aspects that are pertinent to overall algorithm performance:

1. **Architecture Parameters**

   Each highly-threaded many-core machine being used has its own hardware parameter settings among machines from different manufacturers and of different generations along the production road-map. Those parameters may include the number of cores, core groups, registers, the maximal number of threads and thread blocks supported, the memory access width, the size of local memory, and the latency between fast local memory and global memory, etc. Some of these parameters are even configurable.

2. **Algorithm Efficiency**

   For previous parallel machines, algorithm efficiency is mainly weighted by work and parallelism, in terms of the number of computational instructions. For highly-threaded

many-core machines, this still holds, but is insufficient, due to unique memory subsystem. Some algorithms that are efficient for previous machines, may exhibit irregular memory accesses, and incur too much remote memory traffic on many-core machines. Memory locality and cost plays a more critical role on such machines than other machines, and should not be ignored.

3. **Design/Tune Settings**

   In the process of design or performance tuning, there are a number of aspects that can be controlled and varied: runtime configuration for scheduling, choice of access pattern, choice of memory spaces, and choice of data structures. The *frequency*, *pattern*, *usage*, and *cost* of memory interactions are largely the determinant factors to be considered.

   (a) **Thread Access Patterns**

   In highly-threaded many-core machines, threads are typically organized and scheduled in batches, each executing the same instruction for all the inclusive threads in lock-step. If a batch of threads access a contiguous block of memory, those accesses will be *coalesced* into just one memory transaction. This is an important pattern for reducing memory delays. The access pattern most of the time is determined by the algorithm itself, but sometimes also affected by the data structures. For example, Structure of Arrays (SoA) and Array of Structures (AoS) refer to two different ways of laying out data in memory, and they can significantly impact the data access patterns and therefore perform distinctively.

   (b) **Choice of Data Structures**

   Various algorithms have preferred data structures on different machines in terms of performance, depending on computation patterns. For example, if we look

at two example algorithms for all-pairs shortest path problem, the Floyd Warshall algorithm uses an adjacency matrix for the vector-wise computation; while Johnson's algorithm would prefer an adjacency list to save space. On the other hand, due to coalescing and divergence, not all data structures are friendly to many-core machines. Poor choice of data structures results in more long-latency memory accesses and downgrades performance dramatically.

(c) **Choice of Memory Space**

Which memory space to use can make significant difference in performance even for cases with the same work or access pattern. We cannot avoid global memory transactions in general, but some important classes of computation can be designed to have the working sets almost entirely fit on fast local memory and hence benefit from that memory's advantages. For example, if an algorithm unavoidably requires data accesses in a purely random way, designers may wish to partition the random access range and deposit it on to fast local memory to radically reduce the long-latency memory transactions.

(d) **Efficiency of Scheduling**

Scheduling has great impact on performance. On many-core machines, peak performance only comes when all the cores are fully scheduled in use (i.e. full *occupancy*). The same algorithm may behave distinctively given different scheduling by altering the runtime configuration (i.e. threads per block, number of thread blocks). Finally, the occupancy is determined by hardware resource requested and real usage including registers, local memory, etc.

1.3.2 Correlate 3 Spaces of Parameters

Cognizant of the impact of various factors described above, we attempt to build performance models by folding them into 3 spaces and correlating them as illustrated in Fig. 1.2 below.

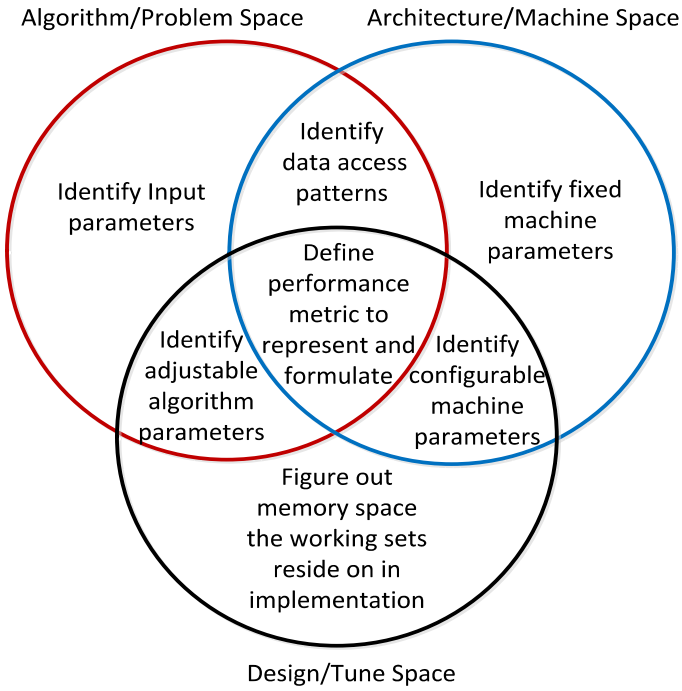


Figure 1.2: Approach to bridge the problem space, architecture space, and design space.

For the problem space, we need to identify both input parameters and adjustable parameters in the algorithm; for the architecture space, we need to identify both fixed (e.g. memory sizes and cores) and configurable parameters (e.g. threads and thread blocks) for the machine. Data access patterns stipulated by the algorithms should also be recognized considering the memory spaces the machine can provide. If the pattern is somehow random or heavily revisited, in design space, the fast local memory would be preferred for working sets to reside on in order to reduce latency in the real implementation, as accessing fast local memory would be 100s times faster than accessing the global memory for strided accesses. If the fast

local memory size is not big enough to accommodate the whole working set, designers might either want to decompose the problem into smaller but more working sets, or just use the slow global memory with sufficient amount of threads to hide latency. The strategy of choice in the design space depends on all these parameters and factors, which are actually intertwined and affected by some of the others. They jointly determine the overall performance of the algorithm running on the underlying machines.

1.3.3 Define Performance Metric

To count in the effect of individual parameters, a performance metric needs to be defined to encode, correlate, and formulate the information obtained above. Then the relationships between input parameters and predicted performance can be explored both theoretically and empirically. We will expose runtime as a unified metric for performance across all 3 spaces/domains presented in Fig. 1.2 above, with expression of parameters from individual spaces positively contributing to it. From the theoretical side, we can effectively use the number of instructions to measure runtime, assuming instructions take constant time. This holds for both computation and memory instructions, although they differ from each other by a factor. From the empirical side, we will measure real runtime with varied parameter settings.

1.4 Contribution and Dissertation Structure

This dissertation makes contributions in the following aspects:

1. We design a high-level theoretical performance model named the Threaded Many-core Memory (TMM) model to capture the performance characteristics of highly-threaded many-core systems, and analyze algorithm performance using it. This is the first formalized asymptotic model for algorithm design, analysis, and comparison in any system which has fast context switching and large number of threads to hide memory latency. It provides a more fine-grained and accurate performance prediction than the PRAM analysis. This model is designed in [99, 101] and will be introduced in Chapter 3.

2. A wide range of classic algorithms are analyzed through the TMM model with sufficient details and empirical results to examine and highlight the power of the TMM model. The algorithms involved include 4 all-pair shortest path algorithms (dynamic programming via a adjacency matrix, Johnson's algorithm via array/heap, Bellman-Ford), FFT, merge sort, string matching via suffix array/string, list ranking, etc. These analyses provide the detailed TMM results and extensively compare with PRAM results in [100, 102] and will be described in Chapter 4.

3. A low-level calibrated performance model is designed to quantitatively predict runtime for all possible runs with various configurations by only one real run. This is achieved by explicit inclusion of performance-impacting factors that are only important over some range of the model's input domain (especially, for example, for smaller input sizes) in addition to the scale factors that enable the calibrated model to make specific quantitative performance predictions throughout the entire configuration space. Those critical factors include application algorithmic complexity, caching factor, and scheduling factor. This model is designed in [104, 105] and will be expatiated in Chapter 5.

4. A number of empirical validations towards the calibrated performance model have been done to demonstrate the effectiveness of this model. A parallel Bloom filter algorithm is designed and implemented on GPUs with 35-fold speedup in [105]. A synthetic micro-benchmark for hashing is presented [104], allowing us to quantitatively explore impact of various access patterns and memory spaces of GPUs on application throughput. Bloom filters in BLAST (the most widely used tool for biosequence similarity search) and another application that exploits hashing in DNA classification are implemented. Empirical results from all applications above line up closely with the model prediction and are presented in Chapter 5.

5. An analytical performance framework is defined in [103] by exploring the coordinated use of the TMM model and the calibrated model and confirming their consistency of prediction for algorithm performance analysis. The framework is able to address parallelism exploited by the algorithm, effectiveness of latency-hiding, and utilization of multiprocessors (occupancy) all together. In particular, it not only helps to explore and reduce the configuration space for tuning kernel execution on highly-threaded many-core machines, but also reflects performance bottlenecks and predicts how the runtime will trend as the problem and other parameters scale. The framework is mainly described in Chapter 6.

In addition to those chapters mentioned above, Chapter 1 briefly introduces the architectures, research questions and methodologies, and highlights the contributions of this dissertation. Chapter 2 describes background information, lists and compares related works for machine models, calibrated models, and GPU algorithms from the literature. At the end, the conclusion and future work are discussed in Chapter 7.

Chapter 2

Background and Related Work

In this chapter, we briefly describe GPU architectures and programming models. Then, we review the abstract machine models for sequential machines, parallel machines, and particular GPU machines as related work. After that, we also review recent work on algorithms and performance analysis of GPUs which are the most common current instantiations of highly-threaded, many-core machines.

2.1 GPU Architectures and Programming Model

Generally, GPUs consist of a number of ***core groups***, each of which can be a *streaming multiprocessor* in NVIDIA GPUs or an *SIMD compute engine* in AMD/ATI GPUs. Those core groups share the same global device memory for inter-core-group communications. On each of the core groups, there are a certain number of processors, named *CUDA cores* in NVIDIA GPUs, and *Thread Processors* in AMD/ATI GPUs. Those processors have their own local storage for individual thread such as registers, but they do share the same on-chip shared memory and caches of different features for inter-core and inter-thread communications.

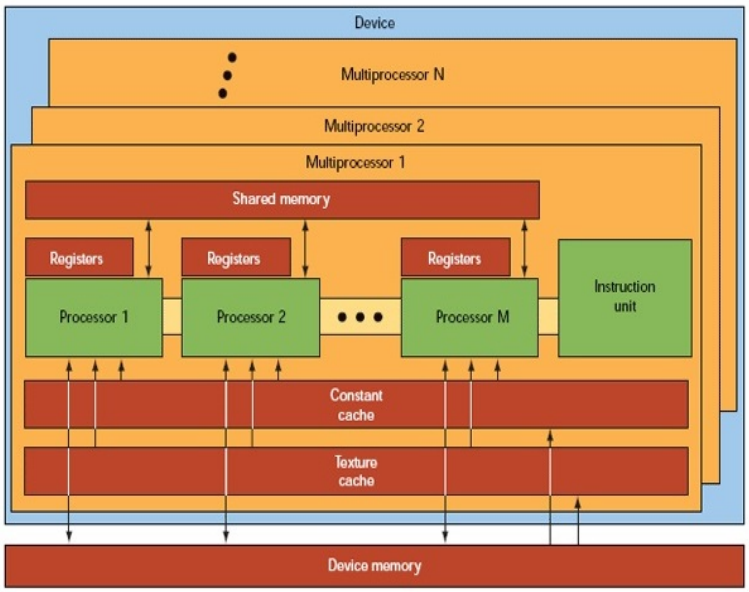


Figure 2.1: NVIDIA GPU Architecture [118]

While AMD GPUs are similar, we will focus the discussion on NVIDIA GPUs since the empirical results we present are all on various NVIDIA platforms. Fig. 2.1 illustrates the general architecture of NVIDIA GPUs. On GPU device, there are a number of multiprocessors sharing the same device (global) memory. On each of the multiprocessors, there are a number of processors (cores) sharing the same shared memory and other specialized memory space such as constant memory and texture memory. For each of the processors, there is also local memory such as registers associated with it.

To harness such architecture, the ***Compute Unified Device Architecture (CUDA)***, a parallel computing platform and programming model, is designed by NVIDIA to efficiently solve many complex computational problems and transparently scale their GPU parallelism to the ever-increasing number of processor cores and threads, while maintaining a stable structure for programs and low learning curve for programmers familiar with the C programming language [118]. As shown in Fig. 2.2, it mainly has 3 key components exposed to

the programmer – a hierarchy of thread groups, a hierarchy of shared memories, and barrier synchronizations. Utilizing these components, people can easily manipulate fine-grained data parallelism and thread parallelism, nested with coarse-grained data parallelism and task parallelism. A CUDA program invokes parallel functions called ***kernels*** that, when called,

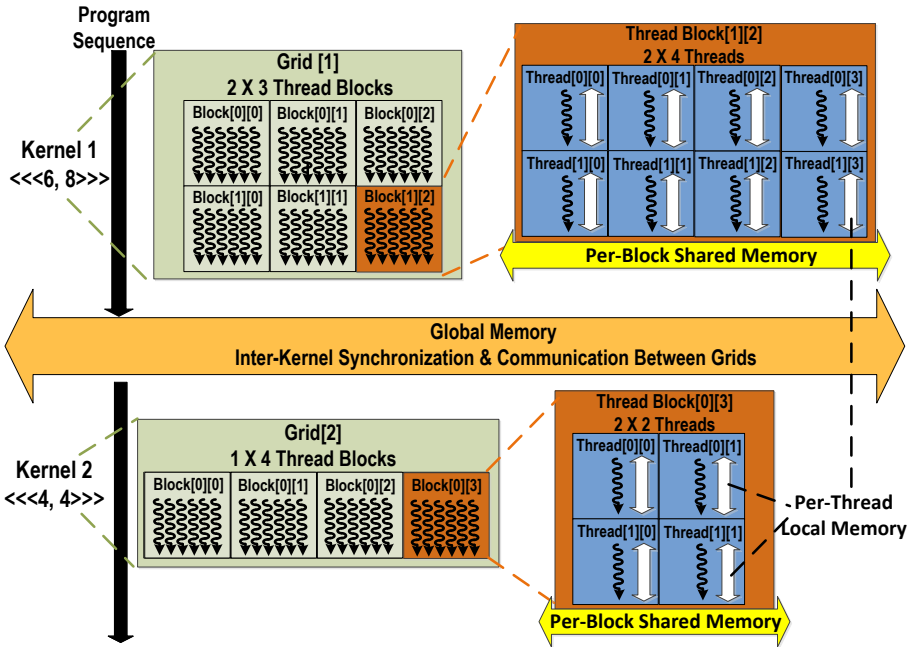


Figure 2.2: NVIDIA GPU thread hierarchy and programming model

are executed N times in parallel by N different CUDA threads. A ***thread block*** is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A ***grid*** is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls.

Each thread has a per-thread ***local memory*** space used for register spills, function calls, and C automatic array variables. Each thread block has a per-block ***shared memory***

space used for inter-thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in ***global memory*** space after kernel-wide global synchronization.

2.2 Abstract Machine Models

Theoretical analysis relies upon models that represent underlying assumptions; if a model does not capture the important aspects of target machines and programs, then the analysis is not predictive of real performance. Over the years, computer scientists have designed various models to capture important aspects of the machines that we use. Many machine and memory models have been designed for various types of sequential and parallel machines.

2.2.1 Sequential Machine Models

The most fundamental model that is used to analyze sequential algorithms is the Random Access Machine (RAM) model [4], which we teach undergraduates in their first algorithms class. This model assumes that all operations, including memory accesses, take unit time. While this model is a good predictor of performance on computationally intensive programs, it does not properly capture the important characteristics of the memory hierarchy of modern machines. Aggarwal and Vitter proposed the Disk Access Machine (DAM) model [3] which counts the number of memory transfers from slow to fast memory instead of simply counting the number of memory accesses by the program. Therefore, it better captures the fact that modern machines have memory hierarchies and exploiting spatial and temporal locality on these machines can lead to better performance. Aggarwal et al. [1] present the Hierarchical

Memory Model (HMM) and use it for a theoretical investigation of the inherent complexity of solving problems in RAM with a memory hierarchy of multiple levels. It differs from the RAM model by defining that access to location x takes $\log x$ time, but it does not consider the concept of block transfers, which collects data into blocks to utilize spatial locality of reference in algorithms. The Block Transfer model (BT) [2] addresses this deficiency by defining that a block of consecutive locations can be copied from memory to memory, taking one unit of time per element after the initial access time. Alpern et al. propose the Memory Hierarchy (MH) Framework [8] that reflects important practical considerations that are hidden by the RAM and HMM models: data are moved in fixed size blocks simultaneously at different levels in the hierarchy, and the memory capacity as well as bus bandwidth are limited at each level. But there are too many parameters in this model that can obscure algorithm analysis. Thus, they simplified and reduced the MH parameters by putting forward a new Uniform Memory Hierarchy (UMH) model [7, 153]. Later, an 'ideal-cache' model was introduced in [52, 126] allowing analysis of cache-oblivious algorithms that use asymptotically optimal amounts of work and move data asymptotically optimally among multiple levels of cache without the necessity of tuning program variables according to hardware configuration parameters.

2.2.2 Parallel Machine Models

For parallel computing, the analogue for the RAM model is the Parallel Random Access Machine (PRAM) model [51], and there is a large body of work describing and analyzing algorithms in the PRAM model [79, 152]. In the PRAM model, the algorithm's complexity is analyzed in terms of its *work* — the time taken by the algorithm on 1 processor, and *span* (also called *depth* and *critical-path length*) — the time taken by the algorithm on an infinite

number of processors. Given a machine with P processors, a PRAM algorithm with work W and span S completes in $\max(W/P, S)$ time.

The PRAM model, although widely used, also unrealistically ignores the vagaries of the memory hierarchy and assumes that all processors work synchronously with uniform cost per memory access, and that interprocessor communication is free. For modern machines, however, this assumption seldom holds. Quite different to PRAM, the Bulk-Synchronous Parallel (BSP) model [147] attempts to bridge theory and practice by allowing processors to work asynchronously, and it models latency and limited bandwidth for distributed memory machines without shared memory. Culler et al. [44] offer a new parallel machine model called LogP based on BSP, characterizing a parallel machine by four parameters: number of processors, communication bandwidth, delay, and overhead. It reflects the convergence towards systems formed by a collection of computers connected by a communication network via message passing. Vitter et al. [155] present a two-level memory model and give a realistic treatment of parallel block transfers in parallel machines. But this model assumes processors are interconnected via sharing of internal memory.

More recently, several models have been proposed emphasizing the use of private-cache chip multiprocessors (CMPs). Arge et al. [11] present the Parallel External Memory (PEM) model with P processors and a two-level memory hierarchy, consisting of the main memory as external memory shared by all processors and caches as internal memory exclusive to each of the P processors. Blelloch et al. [19] present a multicore-cache model capturing the fact that multi-core machines have both per-processor private caches and a large shared cache on-chip. Bender et al. [17] present a concurrent cache-oblivious model. Blelloch et al. [20] also propose a parallel cache-oblivious (PCO) model to account for costs of a wide range of cache hierarchies. Chowdhury et al. [36] present a hierarchical multi-level caching model

(HM), consisting of a collection of cores sharing an arbitrarily large main memory through a hierarchy of caches of finite but increasing sizes that are successively shared by larger groups of cores. They in [39] consider three types of caching systems for CMPs: D-CMP with a private cache for each core, S-CMP with a single cache shared by all cores, and multi-core with private L_1 caches and a shared L_2 cache. All the models above do not accurately describe highly-threaded, many-core systems, due to their distinctive architectures, i.e. the explicit use of many threads for the purpose of hiding memory latency.

2.2.3 GPU Machine Models

More recently, there are a number of machine models proposed particularly for GPUs. Kirtzic et al. [84] proposed the Parallel GPU Model (PGM), which is essentially an adaption of the Bulk-Synchronous Parallel (BSP) model [147], and equates a super-step in BSP with a function unit of a GPU program. This model does not explicitly model the memory subsystem and assumes uniform cost access to all levels of memory. Nakano [116] proposed the Hierarchical Memory Machine (HMM) model, which consists of multiple Discrete Memory Machines (DMMs) representing shared memory and a single Unified Memory Machine (UMM) representing global memory. The HMM model does consider both shared memory accesses and the grouping of global memory accesses. Haque et al. [64] proposed a Many-core Machine Model (MMM) based on the Graham-Brent theorem, which is quite similar to the TMM model, but does not model the impact of threading for hiding memory latency.

2.3 Calibrated Performance Models

While there hasn't been much work on abstract machine models for highly-threaded, many-core machines, there has been a lot of recent work on designing calibrated performance models for particular instantiations of these machines such as NVIDIA GPUs. We review some of that work here.

He et al. [67] focus on the access patterns of gather and scatter operations, which can suffer from low memory bandwidth utilization, and design a probabilistic cache model to predict cache misses. Govindaraju et al. [59] propose a cache model for efficiently implementing three memory intensive scientific applications with nested loops. It is helpful for applications with 2D-block representations while choosing an appropriate block size by estimating cache misses, but is not completely general. Liu et al. [92] describe a general performance model that predicts the performance of a biosequence database scanning application fairly precisely. Their model incorporates the relationship between problem size and performance, but only targets their biosequence application. Ryoo et al. [133] summarize five categories of optimization mechanisms, and use two metrics to prune the GPU performance optimization space by 98% via computing the utilization and efficiency of GPU applications. They do not, however, consider memory latency and multiple conflicting performance indicators. Kothapalli et al. are the first to define a general GPU analytical performance model in [85]. They propose a simple yet efficient solution combining several well-known parallel computation models: PRAM, BSP, QRQW, but they do not model global memory coalescing. Using a different approach, Hong et al. [73] propose another analytical model to capture an estimate of the cost of memory operations by counting the number of parallel memory requests in terms of memory-warp parallelism (MWP) and computation-warp parallelism (CWP). However, their assumption of no cache misses is not always realistic. Sim et al. [138] extend this MWP-CWP model

and present the GPUPerf framework. This framework quantitatively estimates performance along four dimensions: inter-thread instruction-level parallelism, memory-level parallelism, computing efficiency, and serialization effects. These four metrics help to identify performance bottlenecks and suggest what types of optimizations should be done. Baghsorkhi et al. [13] measure performance factors in isolation and later combine them to model the overall performance via workflow graphs so that the interactive effects between different performance factors are modeled correctly. The model can determine data access patterns, branch divergence, and control flow patterns only for a restricted class of kernels on traditional GPU architectures. Zhang and Owens [170] present a quantitative performance model that characterizes an application's performance as being primarily bounded by one of three potential limits: instruction pipeline, shared memory accesses, and global memory accesses. More recently, Kim et al. [82] also design a tool to estimate GPU memory performance by collecting performance-critical parameters. Parakh et al. [120] present a model to estimate both computation time by precisely counting instructions and memory access time by a method to generate address traces.

2.4 Algorithms for Memory Constrained Applications

There has been a rich body of work on the design of parallel algorithms to solve various memory constrained problems on many-core machines, primarily GPUs. Graph exploration is one of the important classes of such problems due to the irregularity of the underlying graph and random nature of memory access patterns. Attempts at accelerating graph processing on GPUs have been extensively made in the past several years for many algorithms, such as breadth-first search (BFS) [65, 71, 72, 97, 111], shortest

paths [28, 65, 81, 107, 108, 112], maximum flow/min cut [66, 69, 74, 98, 140, 149], minimum spanning trees [117, 132, 150, 159], inclusion-based points-to analysis [110], list ranking [46, 130], and connected components [14, 46, 142]. In addition to graph algorithms, a wide range of other memory constrained algorithms have also been attempted on GPUs, including hashing [5, 43, 94], sorting [29, 58, 61, 83, 88, 134, 139, 167], matrix multiplication [34, 55, 61, 63, 76, 88], FFT [61, 88, 113], and dynamic programming [91, 93, 95].

Other than those classic algorithms above, there are also a large set of complicated algorithms for real-world memory constrained problems that can potentially be accelerated on GPUs, such as reducing prediction and test time in machine learning [32, 54, 164, 165, 166], reducing search time in artificial intelligence [33], boosting the efficiency in real-time systems [161] and streaming systems [27, 75, 90], parallelly solving game theory algorithms for optimization [162] and 3D modeling algorithms for imaging [171, 172].

Chapter 3

Threaded Many-core Memory (TMM) Model

The *Threaded Many-core Memory (TMM)* model is meant to model the asymptotic performance of algorithms on highly-threaded, many-core machines. The model abstracts away the details of particular implementations so as to be applicable to many instantiations of these machines, while being particular enough to model the performance of algorithms on these machines with reasonable accuracy. In this chapter, we will describe the important characteristics of these highly-threaded, many-core architectures and our model for analyzing algorithms for these architectures.

3.1 Abstraction of Highly-threaded Many-core Machines

This model is a high-level model meant to be generally applicable to a number of machines which allow a large number of threads with fast context switching. Therefore, it abstracts away many implementation details of both the machine and the algorithm. We also assume

that the hardware provides 0-cost, perfect scheduling between threads. In addition, it also models the machine as having only 2 levels of memory. In particular, we model a slow global memory shared by all processors and fast local memory shared by one *core group*. In practice, these machines may have many levels of memory. However, we are interested in the interplay between the farthest level, since the latencies are the largest at that level, and therefore have the biggest impact on the performance. We expect that the model can be extended to also model other levels of the memory hierarchy.

3.1.1 Architectures

The most important high-level characteristic of highly-threaded, many-core architectures is that they provide a large number of hardware threads and use fast and low-overhead context-switching in order to hide the memory access latency from slow global memory.

Highly-threaded, many-core architectures typically consist of a number of *core groups*, each containing a number of processors (or cores), a fixed number of registers, and a fixed quantity of fast local on-chip memory shared within a core group. A large slow global memory is shared by all the core groups. Registers and local on-chip memory are the fastest to access, while accessing the global memory may potentially take 100s of cycles. The TMM model models these machines as having a memory hierarchy with two levels of memory: slow global memory and fast local memory. In addition, on most highly-threaded, many-core machines, data is transferred from slow to fast memory in *chunks*; instead of just transferring one word at a time, the hardware tries to transfer a large number of words during a memory transfer. The chunk can either be a cache line from hardware managed caches, or an explicitly-managed combined read from multiple threads. Since this characteristic of using high-bandwidth

transfers in order to counter high latencies is common to most many-core machines (and most multi-core machines), the TMM model captures the chunk size as one of its parameters.

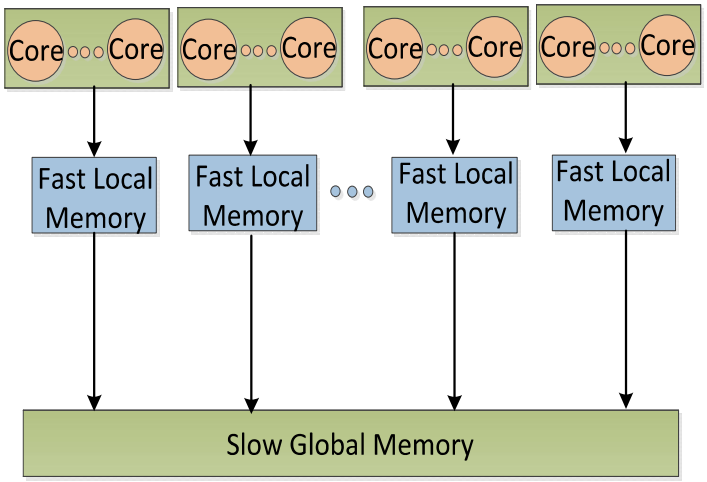


Figure 3.1: Abstracted highly-threaded, many-core architecture. The short arrows from the cores to the local memory symbolize low latency, while the long arrows to the global memory symbolize high latency.

These architectures support a large number of hardware threads, much larger than the number of cores. Cores on a single core group execute in synchronous style where groups of threads execute in lock-step. When a thread group executing on a core group stalls on a slow memory access, in theory, a context switch occurs and another thread group is scheduled on that core group. The abstract architecture is shown in Fig. 3.1. Note that this architecture abstraction ignores a number of details about the physical machine, including thread grouping, scheduling, etc.

3.1.2 Parameters

The TMM model captures the important characteristics of a highly-threaded, many-core architecture by using six parameters shown in Table 3.1. L is the latency for accessing the

slow memory (in our case, the global memory which is shared by all the core groups). P is the total number of cores (or processors) in the machine. C is the maximum chunk size; the number of words that can be read from slow memory to fast memory in one memory transfer. The parameter Z represents the size of fast local memory per core group and Q represents the number of cores per core group. As mentioned earlier, in some instantiations, a core group can have a single core. In this case, a many-core machine looks very much like a multi-core machine with a large number of low-overhead hardware threads. Note that we do not have a parameter for the number of core groups, that quantity is simply P/Q. Finally X is the hardware limit on the number of threads an algorithm is allowed to generate per core. This limit is enforced due to many different constraints, such as constraints on the number of registers each thread uses and an explicit constraint on the number of threads. We unify these constraints into one parameter.

Table 3.1: Architecture parameters.

Parameter	Description
L	Time for a global memory access
P	Number of processors (cores)
C	Memory access width
Z	Size of fast local memory per core group
Q	Number of cores per core group
X	Hardware limit on number of threads per core

In addition to the architecture parameters, we must also consider parameters which are determined by the algorithm. We assume that the programmer has written a correct synchronous program and taken care to balance the workload across the core groups. These program parameters are shown in Table 3.2. T_1 represents the work of the algorithm, that is, the total number of operations that the program must perform (including fast memory accesses). T_∞ represents the span of the algorithm, that is, the total number of operations

on the critical path. These are similar to the analogous PRAM parameters of work and time (or depth or critical-path length).

Table 3.2: Program parameters.

Parameter	Description
T_1	The work or total number of operations
T_∞	The span or the number of operations on the critical path
M	Number of global memory operations
$\mathcal{T}$	Number of threads per core
S	Amount of local memory used per thread

Next we come to program parameters that are specific to many-core programs. M represents the total number of global memory operations performed by the algorithm. Note that this is the total number of operations, not total number of accesses. Since many-core machines often transfer data in large chunks, multiple memory accesses can combine into one memory transfer. For instance, if the many-core machine has a hardware managed cache, and the program accesses data sequentially, then there is only one memory operation for C memory accesses; these will count as one when accounting for M. $\mathcal{T}$ is the number of threads created by the program per core. We assume that the work is perfectly distributed among cores. Therefore, the total number of threads in the system is $\mathcal{T}P$. On highly-threaded, many-core architectures, thread switching is used to hide memory latency. Therefore, it is beneficial to create as many threads as possible. However, the maximum number of threads is limited by both the hardware and the program. The software limitation has to do with parallelism, the number of threads per core is limited by $\mathcal{T} \leq T_1/(T_\infty \cdot P)$. The hardware limits $\mathcal{T} \leq X$. Finally, we have a parameter S, which is local memory used per thread. S and $\mathcal{T}$ are related parameters, since there is a limited amount of local memory in the system. The number of threads per core is at most $\mathcal{T} \leq Z/(QS)$.

3.1.3 Applicability

This TMM model explicitly models the large number of threads per processor and the memory latency to slow memory. Note that while we motivate this model for highly-threaded many-core machines with synchronous computations, in principle, as a high-level abstract model, it can be applicable to many instantiations of hardware platforms that feature a large number of threads with fast context switching for latency-hiding and a hierarchical memory subsystem of at least two levels with a large memory latency gap in between. Typical examples of this set include NVIDIA GPUs, AMD/ATI GPUs, and the uRiKA machine from YarcData. We do not try to model the Intel Xeon Phi, due to its limited use of threading for latency hiding. In contrast, its approach to hide memory latency is primarily based on strided memory access patterns associated with vector computation.

For NVIDIA GPUs, a number of streaming multiprocessors share the same global memory. On each of these multiprocessors, there are a number of CUDA cores that share a fixed number of registers and on-chip (fast) memory A fast hardware-supported context-switching mechanism enables a large number of threads to execute concurrently. Transfers between slow global memory and fast local memory can occur in chunks of at most 32 words; these chunks can only be created if the memory accesses are within a specified range. Accessing the off-chip global memory usually takes 20 to 40 times more clock cycles than accessing the on-chip shared memory/L1 cache [118]. All these features are well captured in the TMM model. Streaming multiprocessors serve the same role as a core group, while CUDA cores are equivalent to the cores defined in TMM. The width of memory access C is 32 due to the coalescing of the threads in a warp. Global memory latency and size of on-chip shared memory/L1 cache are also depicted by L and Z respectively.

Considering AMD/ATI GPUs and taking Cypress, the codename for Radeon HD5800 series GPUs, as an example, the architecture is composed of 20 Single-Instruction-Multiple-Data (SIMD) computation engines. In each SIMD engine, there are 16 Thread Processors (TP) and a 32 KB Local Data Store (LDS). Every TP is arranged as a five-way or four-way Very Long Instruction Word (VLIW) processor, and consists of 5 Stream Cores (SC). Low context-switch threading is well supported, and every 64 threads are grouped into a wavefront executing the same instruction. Basically, the SIMD engine can naturally be modeled by core groups. Each SC is modeled as a core in TMM, summing up to 1600 cores totally. LDS is straightforwardly described by the fast local memory of TMM. The width of memory access C in TMM equals to the wavefront width of 64 for AMD/ATI GPUs.

The uRiKA system from YarcData is also a potential target for the TMM model. Their massively-multithreaded Threadstorm processors support 128 threads/processor. Therefore, 128 defines parameter X, the hard limit of number of threads per processor. The processor's instruction execution hardware essentially does a context switch every instruction cycle, finding the next thread that is ready to issue an instruction into the execution pipeline. This suggests that the memory access width or chunk size C is 1 on these machines. Conceptually, each of the Threadstorm processors is mapped to a core group in the TMM model but, different than the two GPU architectures, it has only one core on-chip, thus Q equals 1.

3.2 TMM Analysis Structure

In order to analyze program performance in the TMM model, we must first calculate the program parameters for the particular program. Once we have calculated these values, we can then try to understand the performance of the algorithm.

We first calculate the effective work of the algorithm T_E. Effective work should consider both work due to computation and work due to memory accesses. Total work due to memory accesses is $M \cdot L$, but since this work is hidden by using threads, the real effective work due to memory accesses is $(M \cdot L)/\mathcal{T}$ Therefore, we have

$$T_E = O\left(\max(T_1, \frac{M \cdot L}{\mathcal{T}})\right) \tag{3.1}$$

Note that this expression assumes perfect scheduling (the threads are context swapped with no overhead, as soon as they are stalled) and perfect load balance between threads.

The time to execute on P cores is represented by T_P and is defined as:

$$T_P = O\left(\max\left(\frac{T_E}{P}, T_\infty\right)\right) = O\left(\max\left(\frac{T_1}{P}, T_\infty, \frac{M \cdot L}{\mathcal{T} \cdot P}\right)\right) \tag{3.2}$$

Therefore, speedup on P cores, S_P, is

$$S_P = \frac{T_1}{T_P} = \Omega\left(\min(P, \frac{T_1}{T_\infty}, \frac{P \cdot T_1 \cdot \mathcal{T}}{M \cdot L})\right) \tag{3.3}$$

For linear speedup, S_P is P. More precisely, for PRAM algorithms, $S_P = \min(P, T_1/T_\infty)$. Therefore, if the first two terms in the min of Eq. (3.3) dominate, then a highly-threaded, many-core algorithm's performance is the same as the corresponding PRAM algorithm. On the other hand, if the last term dominates, then the algorithm's performance depends on other factors. If $\mathcal{T}$ could be unbounded, then the last term will never dominate. However, as we explained earlier, $\mathcal{T}$ is not an unlimited resource and has both hardware and algorithmic upper bounds. Therefore, based on the machine parameters, algorithms that have the same

PRAM performance can have different real performance on highly-threaded, many-core machines. Therefore, this model can help us pick algorithms that provide performance as close as possible to PRAM algorithms.

Chapter 4

Application of the TMM Model

A model is only useful if it can explain and predict empirical data. In this chapter, we investigate the effectiveness of the TMM model. We analyze algorithms for 5 classic problems — all-pairs shortest paths, suffix tree/array for string matching, fast Fourier transform, merge sort, and list ranking — under this model, and compare the results of the analysis with the experimental findings of ours and other researchers who have implemented and measured the performance of these algorithms on an spectrum of diverse GPUs. We find that the TMM model is able to predict important and sometimes previously unexplained trends and artifacts in the experimental data.

4.1 All-pairs Shortest Path (APSP)

In this section, we demonstrate the usefulness of our model by using it to analyze 4 different algorithms for calculating all pairs shortest paths in graphs. All pairs shortest paths is a classic problem for which there are many algorithms. Given a graph $G = (V, E)$ with n vertices and m edges, each edge e has a weight $w(e)$. We must calculate the shortest

weighted path from every vertex to every other vertex. We are interested in asymptotic insights, therefore, we assume that the graphs are large graphs, in particular $n > Z$.

4.1.1 Dynamic Programming via Matrix Multiplication

Our first algorithm is a dynamic programming algorithm [42] that uses repeated matrix multiplication to calculate all pairs shortest paths. The graph is represented as an adjacency matrix A where A_{ij} represents the weight of edge (i, j).

A^l is a transitive matrix where A^l_{ij} represents the shortest path from vertex i to vertex j using at most l intermediate edges. A^1 is the same as the adjacency matrix A and we want to calculate A^{n-1} to calculate all pairs shortest paths.

A^2 can be calculated from A^1 as follows:

$$A^2_{ij} = \min_{0 \leq k < n} (A^1_{ij}, A^1_{ik} + A^1_{kj}). \tag{4.1}$$

Note that the structure of this equation is the same as the structure of a matrix multiplication operation where the sum is replaced by a min operation and the multiplication is replaced by an addition operation. Therefore, we can use repeated matrix multiplication which calculates A^n using $O(\lg n)$ matrix multiplications.

PRAM Algorithm and Analysis

Parallelizing this algorithm for the PRAM model simply involves parallelizing the matrix multiplication algorithm such that each element in the matrix is calculated in parallel. The

total work of $\lg n$ matrix multiplications using a PRAM algorithm is $T_1 = O(n^3 \lg n)$.[3] The span of a single matrix multiplication algorithm is $O(n)$. Therefore, the total span of the algorithm is $T_\infty = O(n \lg n)$.

The time and speedup using P processors is

$$T_P = O\left(\max(\frac{n^3 \lg n}{P}, n \lg n)\right) \tag{4.2}$$

$$S_P = \Omega\left(\min(P, n^2)\right) \tag{4.3}$$

Therefore, the PRAM algorithm gets linear speedup as long as $P \leq n^2$.

TMM Algorithm and Analysis

TMM algorithms are tailored to highly-threaded, many-core architectures generally by using fast on-chip memory to avoid accesses to slow off-chip global memory, coalescing to diminish the time required to access slow memory, and threading to hide the latency of accesses to slow memory. Due to its large size, the adjacency matrix is stored in off-chip global memory. Following traditional block-decomposition techniques, the matrix multiplication is performed by dividing the matrix into sub-blocks with dimension S_D such that the total number of sub-blocks is $(n/S_D)^2$, and allowing each thread block on a core group to operate on an individual sub-block of the data. Individual threads read in the required input sub-blocks, perform the computation of Eq. (6.4) for their assigned sub-block, and write the sub-block out to global memory. This happens $\lg n$ times by repeated squaring.

[3]This can be done faster using Strassen's algorithm. Using Strassen's algorithm will impact the PRAM and the TMM algorithms equally. Therefore, we demonstrate our point using the simpler algorithm.

The work and the span of this algorithm remain unchanged from the PRAM algorithm. However, we must also calculate M, the number of memory operations. Let us first consider a single matrix multiplication operation. There are a total of n^2 elements and each element is read for the calculation of $O(n/S_D)$ other sub-blocks. However, due to the regularity in memory accesses, each sub-block can be read fully coalesced. Therefore, the number of memory operations for one matrix multiply is $O((n^2/C) \cdot (n/S_D)) = O(n^3/(S_D C))$. Also note that since we must fit an $S_D \times S_D$ sub-block in a local memory of size Z on one core group, we get $S_D = \Theta(\sqrt{Z})$. Therefore, for $\lg n$ matrix multiplication operations, $M = O(n^3 \lg n/(S_D C)) = O(n^3 \lg n/(\sqrt{Z} \cdot C))$.

Now we are ready to calculate the time on P processors.

$$T_P = O\left(\max(\frac{T_1}{P}, T_\infty, \frac{M \cdot L}{\mathcal{T} \cdot P})\right) \tag{4.4}$$

$$= O\left(\max(\frac{n^3 \lg n}{P}, n \lg n, \frac{n^3 \lg n \cdot L}{\sqrt{Z} \cdot C \cdot \mathcal{T} \cdot P})\right) \tag{4.5}$$

Therefore, the speedup on P processors is

$$S_P = T_1/T_P \tag{4.6}$$

$$= \Omega\left(\min(P, n^2, \frac{\sqrt{Z} \cdot C \cdot \mathcal{T}}{L} \cdot P)\right) \tag{4.7}$$

We can now compare the PRAM and TMM analysis and note that the speedup is P as long as $\sqrt{Z}C\mathcal{T}/L \geq 1$. We also know that $\mathcal{T} \leq \min(X, Z/(QS))$, and $S = O(1)$, since each thread only needs constant memory. Therefore, we can conclude that the algorithm achieves linear speedup as long as $L \leq \min(\sqrt{Z}CX, Z^{3/2}C/Q)$.

4.1.2 Johnson's Algorithm: Dijkstra's Algorithm (Binary Heaps)

Johnson's algorithm [77] is an all pairs shortest paths algorithm that uses Dijkstra's single source algorithm as the subroutine and calls it n times, once from each source vertex. Dijkstra's algorithm is a greedy algorithm for calculating single source shortest paths. The pseudo-code for Dijkstra's algorithm is given in Algorithm 1 [47]. The single source algorithm consists of n insert operations, m decrease-key operations and n delete-min operations from a min-priority queue. The standard way of implementing Dijkstra's algorithm is to use a binary or a Fibonacci heap to store the array elements. We now consider a binary heap implementation so that each operation (insert, decrease-key, and delete-min) takes $O(\lg n)$ time. Note that Dijkstra's algorithm does not work when there are negative weight edges.

PRAM Algorithm and Analysis

A simple parallel implementation of Johnson's algorithm using Dijkstra's algorithm consists of doing each single-source shortest path calculation in parallel. The total work of a single-source computation is $O(m \lg n + n \lg n)$. For simplicity, we assume that the graph is connected, giving us $O(m \lg n)$. Therefore, the total work for all pairs shortest paths is $T_1 = O(mn \lg n)$. The span is $T_\infty = O(m \lg n)$ since each single source computation executes sequentially. The time and speedup using P processors is

$$T_P = O\left(\max(\frac{mn \lg n}{P}, m \lg n)\right) \tag{4.8}$$

$$S_P = \Omega\left(\min(P, n)\right) \tag{4.9}$$

Therefore, the PRAM algorithm gets linear speedup as long as $P \leq n$.

Algorithm 1 Dijkstra's Algorithm

```
1: Input: Graph G = (V, E), |V| = n, |E| = m
2: Input: W is weight of edges, |W| = m
3: Input: S is source vertex
4: Output: dist[n]
       {Initialize distance array}
5: for all u ∈ V do
6:    dist[u] = ∞
7: end for
8: dist[S] = 0
9: for all u ∈ V do
10:   Q ← dist[u]
11: end for
       {Propagate the distance update to all vertices}
12: while Q not empty do
13:   u = deletemin(Q)
14:   for each edge (u, v) ∈ E do
15:      if dist[v] > dist[u] + W[u, v] then
16:         dist[v] = dist[u] + W[u, v]
17:         decreasekey(Q, v)
18:      end if
19:   end for
20: end while
```

TMM Algorithm and Analysis

The TMM algorithm is very similar to the PRAM algorithm where each thread computes a single source shortest path. Therefore, each thread requires a min-heap of size n. Since n may be arbitrarily large compared to $Z/Q\mathcal{T}$ (the share of local memory for each thread), these heaps cannot fit in local memory and must be allocated on the slow global memory.

The work and span are the same as the PRAM algorithm. We must now compute M. Note that each time the thread does a heap operation, it must access global memory, since the heaps are stored in global memory. In addition, binary heap accesses are not predictable

and regular, so the heap accesses from different threads cannot be coalesced. Therefore, the total number of memory operations is $M = O(mn \lg n)$.[4]

Now we are ready to calculate the time on P processors.

$$T_P = O\left(\max(\frac{T_1}{P}, T_\infty, \frac{M \cdot L}{\mathcal{T} \cdot P})\right) \tag{4.10}$$

$$= O\left(\max(\frac{mn \lg n}{P}, m \lg n, \frac{mn \lg n \cdot L}{\mathcal{T} \cdot P})\right) \tag{4.11}$$

Therefore, the speedup on P processors is

$$S_P = \Omega\left(\min(P, n, \frac{\mathcal{T}}{L} \cdot P)\right) \tag{4.12}$$

Note that this algorithm gets linear speedup only if $\mathcal{T}/L \geq 1$. Therefore, the number of threads this algorithm needs to get linear speedup is very large. We know that $\mathcal{T} \leq \min(X, Z/(QS))$, and $S = O(1)$ for this algorithm. This allows us to conclude that this algorithm achieves linear speedup only if $L \leq \min(X, Z/Q)$, since each thread needs only constant memory. These conditions are much stricter than those imposed by the dynamic programming algorithm.

4.1.3 Johnson's Algorithm: Dijkstra's Algorithm (Arrays)

This algorithm is similar to the previous algorithm in that it still uses n single-source Dijkstra's algorithm calculations. However, instead of binary heaps, we use arrays to do delete-min and decrease-key operations.

[4]There are other accesses that are not heap accesses, but those are asymptotically fewer and can be ignored.

PRAM Algorithm and Analysis

The PRAM algorithm is very similar to the algorithm that uses binary heaps. Each single source shortest path is computed in parallel. However, in this algorithm, we simply store the current estimates of the shortest path of vertices in an array instead of a binary heap. Therefore, there are n arrays of size n, one for each single source shortest path calculation. Each decrease-key now takes $O(1)$ time, since one can simply decrease the key using random access. Each delete-min, however, takes $O(n)$ work, since one must look at the entire array to find the minimum element. Therefore, the work of the algorithm is $T_1 = O(n^3 + mn)$ and the span is $O(n^2 + m)$. We can improve the span by doing delete-min in parallel, since one can find the smallest element in an array in parallel using $O(n)$ work and $O(\lg n)$ time using a parallel prefix computation. This brings the total span to $T_\infty = O(n \lg n + m)$ while the work remains the same.

The time and speedup using P processors is

$$T_P = O\left(\max(\frac{n^3}{P}, n \lg n + m)\right) \tag{4.13}$$

$$= O\left(\max(\frac{n^3}{P}, n \lg n, m)\right) \tag{4.14}$$

$$S_P = \Omega\left(\min(P, \frac{n^2}{\lg n}, \frac{n^3}{m})\right) \tag{4.15}$$

TMM Algorithm and Analysis

The TMM algorithm is similar to the PRAM algorithm, except that each core group is responsible for a single-source shortest path calculation. Therefore, all the threads on a single core group ($Q\mathcal{T}$ in number) cooperate to calculate a single shortest path computation.

Since we assume that $n > Z$, the entire array does not fit in local memory and must be read with each delete-min operation. Therefore, the span of the delete-min operation changes. For each delete-min operation, elements are read into local memory in chunks of size Z. For each chunk, the minimum is computed in parallel in $O(\lg Z)$ time. Therefore, the span of each delete-min operation is $O((n/Z)\lg Z)$. Therefore, the total span is $T_\infty = O(n^2 \lg Z/Z)$. The work is the same as the PRAM work.

We must now compute the number of memory operations, M. There are n^2 delete-min operations in total, and each reads the array of size n coalesced. In addition, there are a total of mn decrease key operations, but these reads cannot be coalesced. Therefore, $M = O(n^3/C + mn)$.

$$\begin{aligned} T_P &= O\left(\max(\frac{T_1}{P}, T_\infty, \frac{M \cdot L}{\mathcal{T} \cdot P})\right) && (4.16)\\ &= O\left(\max(\frac{n^3}{P}, \frac{n^2 \lg Z}{Z}, \frac{(\frac{n^3}{C} + mn) \cdot L}{\mathcal{T} \cdot P})\right) && (4.17)\\ &= O\left(\max(\frac{n^3}{P}, \frac{n^2 \lg Z}{Z}, \frac{n^3 \cdot L}{C \cdot \mathcal{T} \cdot P}, \frac{mn \cdot L}{\mathcal{T} \cdot P})\right) && (4.18) \end{aligned}$$

Speedup is

$$S_P = \Omega\left(\min(P, \frac{nZ}{\lg Z}, \frac{C \cdot \mathcal{T}}{L} \cdot P, \frac{n^2 \cdot \mathcal{T}}{m \cdot L} \cdot P)\right) \quad (4.19)$$

Again, in this algorithm, $\mathcal{T} \leq \min(X, Z/(QS))$, and $S = O(1)$ since each thread needs only constant memory. Therefore, if $L \leq \min(CX, CZ/Q, n^2X/m, n^2Z/(mQ))$, then the PRAM performance dominates.

4.1.4 n Iterations of Bellman-Ford Algorithm

This is another all pairs shortest paths algorithm that uses a single-source Bellman-Ford algorithm as a subroutine. The algorithm is given in Algorithm 2 [16, 89].

Algorithm 2 Bellman-Ford

```
1: Input: Graph G = (V, E), |V| = n, |E| = m
2: Input: W is weight of edges, |W| = m
3: Input: S is source vertex
4: Output: dist[n]
       {Initialize distance array}
5: for all u in V do
6:    dist[u] = ∞
7: end for
8: dist[S] = 0
       {Update the distance for all vertices n − 1 times}
9: for i = 1 : (n − 1) do
10:   for each edge e(u, v) ∈ E do
11:      if dist[v] > dist[u] + W[u, v] then
12:         dist[v] = dist[u] + W[u, v]
13:      end if
14:   end for
15: end for
```

PRAM Algorithm and Analysis

Again, one can do each single source computation in parallel. Each single source computation takes $O(mn)$ work, making the total work of all pairs shortest paths $O(mn^2)$ and the total span $O(mn)$. One can improve the span by relaxing all edges in one iteration in parallel making the span $O(n)$.

$$T_P = O\left(\max(\frac{mn^2}{P}, n)\right). \tag{4.20}$$

$$S_P = \Omega\left(\min(P, mn)\right). \tag{4.21}$$

TMM Algorithm and Analysis

The TMM algorithm for this problem is more complicated and requires more data structure support. Each core group is responsible for one single-source shortest path calculation. For each single source calculation, we maintain three arrays, A, B and W, of size m, and one array D of size n. D contains the current guess of the shortest path to vertex i. B contains ending vertices of edges, sorted by vertex ID. Therefore B may contain multiple instances of the same vertex if that vertex has multiple incident edges. $A[i]$ contains the starting vertex of the edge that ends at $B[i]$ and $W[i]$ contains the weight of that edge. Therefore, both D and B are sorted.

Each thread deals with one index in the array and relaxes that edge in each iteration. All threads relax edges in parallel in order of B. The total work and span are the same as the PRAM algorithm. We can now calculate the time and speedup assuming threads can read all the arrays coalesced, $M = O(mn^2/C + n^3/C) = O(mn^2/C)$ for connected graphs.

$$T_P = O\left(\max(\frac{T_1}{P}, T_\infty, \frac{M \cdot L}{\mathcal{T} \cdot P})\right) \tag{4.22}$$

$$= O\left(\max(\frac{mn^2}{P}, n, \frac{mn^2 \cdot L}{C \cdot \mathcal{T} \cdot P})\right) \tag{4.23}$$

Therefore, the speedup on P processors is

$$S_P = \Omega\left(\min(P, mn, \frac{C \cdot \mathcal{T}}{L} \cdot P)\right) \tag{4.24}$$

In this case, we get linear speedup if $C\mathcal{T}/L \geq 1$. Subject to the limits on threads of $\mathcal{T} \leq \min(X, Z/(QS))$ and $S = O(1)$ for constant local memory usage per thread, this requires $L \leq \min(CX, CZ/Q)$.

4.1.5 Comparison of Various Algorithms

As our analysis of shortest paths algorithms indicates, the TMM model allows us to take the unique properties of highly-threaded, many-core architectures into consideration while analyzing the algorithms. Therefore, the model provides more nuance in the analysis of these algorithms for the highly-threaded, many-core machines than the PRAM model. In this section, we will compare the running times of the various algorithms and see what interesting things this analysis tells us.

Table 4.1 indicates the running times of the various algorithms in both the PRAM model and the TMM model, as well as the conditions under which TMM results are the same as the PRAM results. We have ignored the span term, since the span is small relative to work in all of these algorithms. As we can see, if L is small, then highly-threaded, many-core machines provide PRAM performance. However, the cut-off value for L is different for different algorithms where the performance in the TMM model differs from the PRAM model is different for different algorithms. Therefore, the TMM model can be informative when comparing between algorithms.

We will perform two types of comparison between these algorithms in this section. The first one considers the direct influence of machine parameters on asymptotic performance. Since machine parameters do not scale with problem size, in principle, machine parameters cannot change the asymptotic performance of algorithms in terms of problem size. That is, if the PRAM analysis indicates that some algorithm has a running time of $O(n)$ and another one has the running time of $O(n \lg n)$, for large enough n, the first algorithm is always *asymptotically* better since eventually $\lg n$ will dominate whatever machine parameter

Table 4.1: Algorithm running times and constraints for linear speedup.

Algorithm	Time (PRAM)	Time (TMM)	Constraints (for linear speedup)	
Dynamic Programming	$\frac{n^3 \lg n}{P}$	$\frac{n^3 \lg n \cdot L}{\sqrt{Z} C \mathcal{T} P}$	$L \leq \sqrt{Z} C X$	$L \leq Z^{3/2} C / Q$
Johnson's (Binary Heap)	$\frac{mn \lg n}{P}$	$\frac{mn \lg n \cdot L}{\mathcal{T} P}$	$L \leq X$	$L \leq Z / Q$
Johnson's (Array)	$\frac{n^3}{P}$	$\frac{n^3 L}{C \mathcal{T} P}$, $\frac{n^2}{m} \geq C$ $\frac{mnL}{\mathcal{T} P}$, $\frac{n^2}{m} < C$	$L \leq CX$ $L \leq n^2 X / m$	$L \leq Z / Q \cdot C$ $L \leq n^2 Z / (mQ)$
n iteration Bellman-Ford	$\frac{n^2 m}{P}$	$\frac{mn^2 L}{C \mathcal{T} P}$	$L \leq CX$	$L \leq CZ / Q$

advantage the second algorithm may have. Therefore, for this first comparison, we only compare algorithms which have the same asymptotic performance under the PRAM model.

Second, we will also do a non-asymptotic comparison where we compare algorithms when the problem size is relatively small, but not very small. In particular, we look at the case when $\lg n < \sqrt{Z}$. In this case, even algorithms that are asymptotically worse in the PRAM model can be better in the TMM model, for large latency L. In the next section, we will look at even smaller problem sizes where the effects are even more dramatic.

Influence of Machine Parameters

As the table shows, the limits on machine parameters to get linear speedup are different for different algorithms. Therefore, even when two algorithms have the same PRAM performance, their performance on highly-threaded, many-core machines may vary significantly. Let us consider a few examples:

- Dynamic Programming vs. Johnson's Algorithm using Binary Heaps when $m = O(n^2)$.

  If $m = O(n^2)$ (i.e., the graph is dense), the PRAM performance for both algorithms is the same. However when $Z/Q < L < Z^{3/2}C/Q$, Johnson's algorithm has a significantly worse running time. Take the example of $L = O(Z^{3/2}C/Q)$. The Johnson running time is $O(n^3 \lg n\sqrt{Z}C/P)$ while the running time of the dynamic programming algorithm is simply $O(n^3 \lg n/P)$.

- Johnson's Algorithm using Binary Heaps vs. Johnson's Algorithm using Arrays when $m = O(n^2/\lg n)$.

  If $m = O(n^2/\lg n)$ (i.e., a somewhat sparse graph), these two algorithms have the same PRAM performance, but if $Z/Q < L \leq ZC/Q$, then the array implementation is better. For $L = ZC/Q$, the binary heap implementation has a running time of $O(n^3C/P)$, while the array implementation has a running time of simply $O(n^3/P)$.

Influence of Graph Size

The previous section shows the asymptotic power of the model; the results there hold for large sizes of graphs asymptotically. However, the TMM model can also help decide on what algorithm to use based on the size of the graph. In particular for certain sizes of graphs,

algorithm A can be better than algorithm B even if it is asymptotically worse in the PRAM model. Therefore, the TMM model can give us information that the PRAM model cannot.

Consider the example of dynamic programming vs. Johnson's algorithm using arrays. In the PRAM model, the dynamic programming algorithm is unquestionably worse than Johnson's. However, if $\lg n < \sqrt{Z}$, we may have a different conclusion. In this case, dynamic programming has runtime:

$$\frac{n^3 \lg n \cdot L}{\sqrt{Z} C \mathcal{T} P} = \frac{n^2 L}{\mathcal{T} P} \cdot \frac{n \lg n}{\sqrt{Z} C} < \frac{n^2 L}{\mathcal{T} P} \cdot \frac{n}{C} \tag{4.25}$$

While Johnson's algorithm has runtime:

$$\min\left(\frac{n^3 L}{C \mathcal{T} P}, \frac{mnL}{\mathcal{T} P}\right) = \frac{n^2 L}{\mathcal{T} P} \cdot \min\left(\frac{n}{C}, \frac{m}{n}\right) \tag{4.26}$$

If $n^2/m < C$, i.e. dense graphs, $n/C < m/n$. Combine (4.25) and (4.26), we have

$$\frac{n^3 \lg n \cdot L}{\sqrt{Z} C \mathcal{T} P} < \frac{n^3 L}{C \mathcal{T} P} \quad , \qquad \text{if} \quad \frac{n^2}{m} < C \tag{4.27}$$

This indicates that when for small enough graphs where $\lg n < \sqrt{Z}$, there is a dichotomy. For dense graphs $n^2/m < C$, the dynamic programming algorithm should be preferred, while for sparse graphs, Johnson's algorithm with arrays is better. We illustrate this performance dependence on sparsity with experiments in Section 4.1.7.

We get a similar result when comparing the dynamic programming algorithm with Bellman-Ford when $m = O(n)$. In spite of being worse in the PRAM world, the dynamic programming algorithm is better when $\lg n < \sqrt{Z}$.

Our model therefore allows us to do two things. First, for a particular machine, given two algorithms which are asymptotically similar, we can pick the more appropriate algorithm for that particular machine given its machine parameters. Second, if we also consider the problem size, then we can do more. For small problem sizes, the asymptotically worse algorithm may in fact be better because it interacts better with the machine. We will draw more insights of this type in the next section.

4.1.6 Effect of Problem Size

In Section 4.1.5, we explored the asymptotic insights that can be drawn from the TMM model. However, the TMM model can also inform insights based on problem size. In particular, some algorithms can take advantage of smaller problems better than others, since they can use fast local memory more effectively. In this section, we explore the insights that the TMM model provides in these cases.

Vertices Fit in Local Memory

When $n < Z$, all the vertices fit in local memory. Note that this doesn't mean that the entire problem fits in local memory, since the number of edges can still be much larger than the number of vertices. In this scenario, the number of memory accesses by the first, second, and fourth algorithms is not affected at all. In the dynamic programming algorithm, we consider the array of size n^2 and being able to fit a row into local memory does not reduce the number of memory transfers. In Johnson's algorithm using binary heaps, each thread does its own single source shortest path. Since the local memory Z is shared among $Q\mathcal{T}$ threads, each

thread cannot hold its entire vertex array in local memory. In the Bellman-Ford algorithm, the cost is dominated by the cost of reading the edges. Therefore, the bounds do not change.

For Johnson's algorithm using arrays, the cost is lower. Each core group can store the vertex array and does not need to access it from slow memory. Therefore the bound on the number of memory operations changes to $M = O(n^2/C + mn) = O(mn)$ for connected graphs.

For these small problem sizes, the TMM model can provide even more insight. As an example, compare the two versions of Johnson's algorithm, the one that uses arrays and the one that uses heaps. When $m = O(n^2/\lg^2 n)$, the algorithm that uses heaps is better than the algorithm that uses arrays in the PRAM model. But in the TMM model, for large L, the algorithm that uses heaps has the running time of $O(Lmn \lg n/(\mathcal{T}P)) = O(Ln^3/(\mathcal{T}P \lg n))$, while the algorithm that uses arrays has the running time of $O(Ln^3/(\mathcal{T}P \lg^2 n))$. Therefore, the algorithm that uses arrays is better. Note that asymptotic analysis is a little dubious when we are talking about small problem sizes; therefore, this analysis should be considered skeptically. However, the analysis is rigorous when we consider the circumstance that local memory size grows with problem size (i.e., Z is asymptotic). Moreover, this type of analysis can still provide enough insight that it might guide implementation decisions under the more realistic circumstance of bounded (but potentially large) Z.

Edges Fit in the Combined Local Memories

When $m = O(PZ/Q)$, the edges fit in all the memories of the core groups combined. Again, the running time of the first, second, and third algorithms do not change, since they cannot take advantage of this property. However, the Bellman-Ford algorithm can take advantage of this property and each thread across all core groups is responsible for relaxing a single

edge. Now a portion of the arrays A, B and W fit in each core group's local memory and they never have to be read again. Therefore, the number of memory operations reduces to $M = O(n^3/C)$. The run time under the TMM model reduces to $O(n^3 L/(C\mathcal{T}P))$. Again, compare Bellman-Ford algorithm with Johnson's algorithm using binary heaps. When $m = O(n^2/\lg n)$, Johnson's algorithm is better than the Bellman-Ford algorithm in the PRAM model. However, in the TMM model, Johnson's has run time of $O(Lmn \lg n/(\mathcal{T}P)) = O(Ln^3/(\mathcal{T}P))$, while Bellman-Ford with a run time of $O(Ln^3/(C\mathcal{T}P))$ flips to be better.

4.1.7 Empirical Validation

In this section, we conduct experiments to understand the extent of the applicability of our model in explaining the performance of algorithms on real machines. This evaluation is a proof-of-concept that the model successfully predicts performance on examples of the highly-threaded, many-core machine. It is not meant to be an exhaustive empirical study of the model's applicability for all instances of highly-threaded, many-core machines. We implemented two all-pairs shortest paths algorithms: the dynamic programming using matrix multiplication and Johnson's algorithm using arrays, on NVIDIA GPUs.

In these experiments, we investigate the following aspects of the TMM model:

- **Effect of the number of threads:** The fact that the TMM model incorporates the number of threads per processor in the model is the primary differentiator between the PRAM and TMM models. The TMM model predicts that as the number of threads increases the performance increases, up to a certain point. After this point, the number of threads does not matter, and the TMM model behaves the same as the PRAM model.

In this set of experiments, we will use both the dynamic programming and Johnson's algorithms to demonstrate this dependence on the number of threads.

- **Effect of fast local memory size:** In some algorithms, including the dynamic programming via matrix multiplication, the size of the fast memory affects the performance of the algorithm in the TMM model. We investigate this dependence.

- **Effect of machine parameters:** Both the 2 investigations above use the same machine, therefore the machine parameters in the TMM model are constant and negligible. Next, we will change the machine parameters by using a different machine, investigate whether the observations above still hold, and demonstrate the effect of different machine parameters on real performance.

- **Power of providing more fine-grained and accurate results than the PRAM model for comparing algorithms:** For Johnson's algorithm using arrays, the PRAM performance does not depend on the graph's density. However, the TMM model predicts that performance can depend on the graph's density, when the number of threads is insufficient for the performance to be equivalent to the PRAM model. Therefore, even though Johnson's algorithm is always faster than the dynamic programming algorithm according to the PRAM model (since its work is n^3 while the dynamic programming algorithm has work $n^3 \lg n$), the TMM model predicts that when the number of threads is small, the dynamic programming algorithm may do better, especially for dense graphs. We demonstrate through experiments that this is a true indicator of performance.

Experimental Setup

The experiments are carried out on an NVIDIA *Fermi* architecture GTX 480 and a *Kepler* architecture GTX680. GTX 480 has 15 multiprocessors, each with 32 cores. GTX 680 has 8 multiprocessors, each with 192 cores. As typical highly-threaded, many-core machines, they also feature a big global memory and 16 KB/48 KB of configurable on-chip shared memory per multiprocessor, which can be accessed with latency significantly lower than the global memory.

Runtimes are measured across various configurations of each problem, including graph size, thread count, shared memory size, and graph density. When plotted as execution time, the performance units are in seconds. In many cases, however, the trends we wish to see are more readily apparent when performance is shown in terms of speedup rather than execution time. This poses a problem, however, as it is arguably meaningless to attempt to realistically measure the single-core execution time of an application deployed on a modern GPU. We address this issue using the following technique: all speedup plots compare the measured, empirical execution time on P cores to the theoretical, asymptotic execution time on 1 core using the PRAM model. As a result, the speedup axis does not represent a quantitatively meaningful scale, and the scale is labeled "arbitrary" on the graphs to reflect this fact; however, the shape of the curves are representative of the speedup achievable relative to a fixed serial execution time.

Effect of the Number of Threads

The TMM model indicates that when the number of threads is small, the performance of algorithms depends on the number of threads. With sufficient number of threads, the

performance converges to the PRAM performance and only depends on the problem size and the number of processors. We verify this result using both the dynamic programming and Johnson's algorithms.

For the dynamic programming algorithm, we generate random graphs with {2K, 4K, 8K, 16K} vertices. To better utilize fast local memory, the problem is decomposed into sub-blocks, and we must also pick a sub-block dimension S_D so that the sub-block size is ${S_D}^2$. Since we only care about the effect of threads and not the effect of shared memory (to be considered in the next subsection), here we show the results with a sub-block dimension S_D = 64, as it allows us to generate the maximum number of threads. We increase the number of threads until we reach either the hardware limit or the limit imposed by the algorithm.

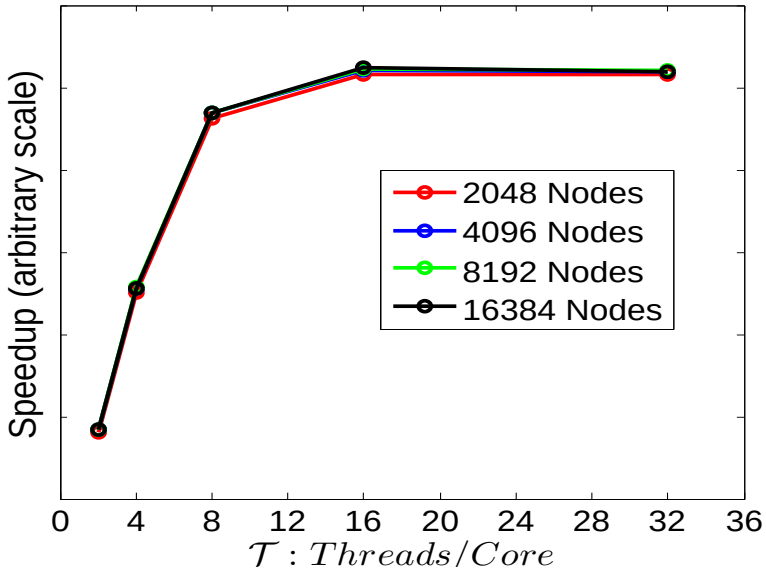


Figure 4.1: Speedup (theoretical T_1 via PRAM model over empirically measured T_P) of the dynamic programming algorithm, varying the number of threads per core from 2 to 32 (sub-block dimension $S_D = 64$).

Fig. 4.1 shows the speedup while varying the number of threads per core. We see that the speedup increases approximately linearly with the number of threads per core (as predicted by Eq. (4.7)) and then flattens out. This indicates that for this experiment, 8 is an estimated

threshold of threads/core where the TMM model switches to the "PRAM range" and the number of threads no longer matters. Note that the expression for this threshold does not depend on the graph size, as it is equal to $L/\sqrt{Z}C$. Also note that the speedup (both in and out of the PRAM range) is not impacted by the size of the graph (again as predicted by Eq. (4.7)).

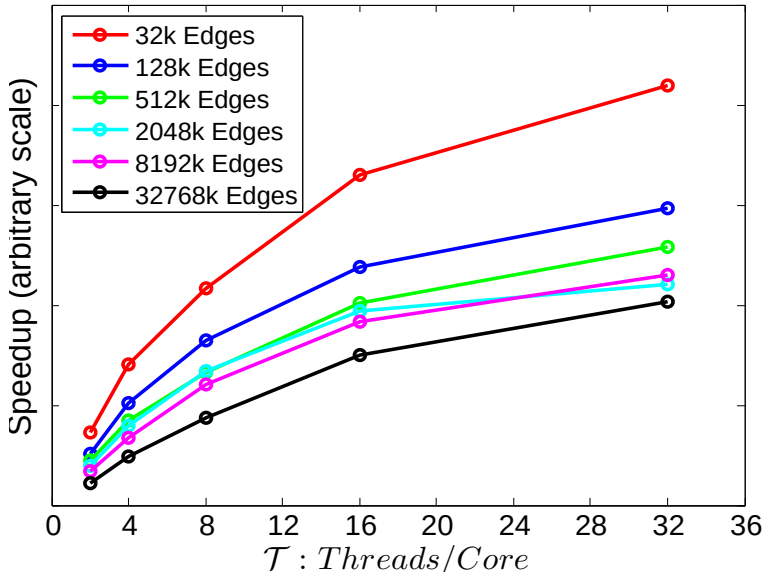


Figure 4.2: Speedup of Johnson's algorithm using arrays vs. threads/core for different graph densities. All curves are with 8K nodes. Again, speedup is theoretical T_1 divided by empirically measured T_P.

We see a similar performance dependence on the number of threads in Johnson's algorithm. Here we ran experiments with 8K vertices and varied the number of edges (ranging between 32K and 32M). The speedup graph is shown in Fig. 4.2. As we increase the number of threads, the speedup increases. We see two other interesting things, however. First, we never see the flattening of performance with increasing thread counts that is seen with the dynamic programming algorithm. Therefore, it appears that Johnson's algorithm requires more threads to reach the PRAM range where the performance no longer depends on the number of threads. This is also predicted by our model as the number of threads/core

required by the dynamic programming algorithm to reach PRAM range is $\mathcal{T} \geq L/\sqrt{Z}C$ while the corresponding number of threads required by Johnson's is $\mathcal{T} \geq L/C$, clearly a larger threshold. Johnson's algorithm is not taking advantage of the fast local memory, and this factor influences the number of threads required to hide the latency to global memory. Second, we see that the performance depends on the number of edges. This is consistent with the fact that we are in the TMM range where the runtime is $(mnL/\mathcal{T}P)$ and not in the PRAM range where the runtime only depends on the number of vertices.

The dependence on graph density is explored further in Fig. 4.3. Here, the runtime is plotted vs. number of graph edges for varying threads/core. The linear relationship predicted by the last term of Eq. (4.18) (for dense graphs) is illustrated clearly in the figure.

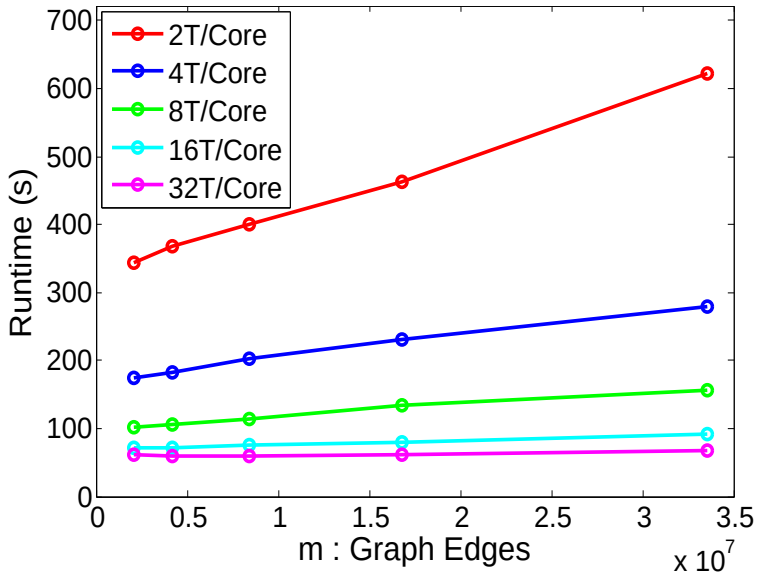


Figure 4.3: Runtime of Johnson's algorithm on graphs with constant 8K nodes and varying density by increasing edges. Threads/core varies from 2 to 32.

Effect of Fast Local Memory Size

In highly-threaded, many-core machines, access to local memory is faster than access to slow global memory. Among our shortest paths algorithms, only the dynamic programming algorithm makes use of the local memory and the running time depends on this fast memory size. In this experiment we verify the effect of this fast memory size on algorithm performance.

We set the fast memory size on our machine and measure its effect. Fig. 4.4 illustrates how this change has an impact on speedup across a range of threads/core. For a fixed Z (fast memory size), the maximum sub-block dimension S_D can be determined. Then, varying thread counts has the same effect as previously illustrated in Fig. 4.1, increasing threads/core increases performance until the PRAM range is reached. But as we can see from the figure, different sub-block dimensions have different performance for the same number of threads/core. This effect is predicted by Eq. (4.7). As we increase the size of local memory, the performance improves, since we can use bigger sub-blocks.

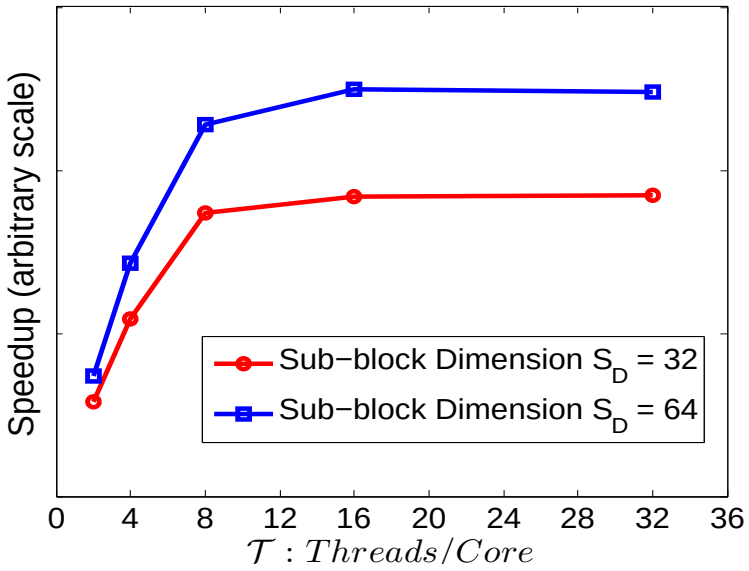
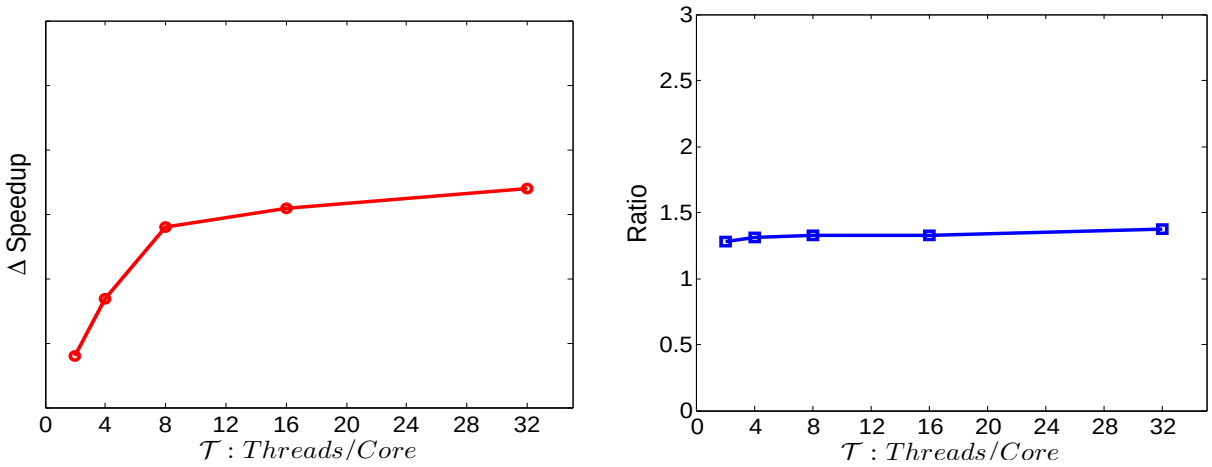


Figure 4.4: Speedup of the dynamic programming algorithm for different sub-block dimensions (S_D), varying the threads/core on graphs with 16K nodes.



(a) Spread (b) Ratio

Figure 4.5: Different format of data from the two curves in Fig. 4.4 with the same speedup scale in order to isolate the effect of sub-block size from the effects of other parameters. (a) Spread of performance between sub-block dimension 64 and sub-block dimension 32. (b) Ratio of performance between sub-block dimension 64 and sub-block dimension 32.

In order to isolate the effect of sub-block dimension from the effects of other parameters, we also plot this data in a pair of different formats in Fig. 4.5(a) and Fig. 4.5(b). The first curve shows the difference between the speedups for different sub-block dimensions. As the curve indicates, when setting the number of threads/core below the PRAM range (i.e.,the range where speedup is linear in threads/core $\mathcal{T}$), the delta speedup increases linearly with the number of threads/core, consistent with the model prediction of $(S_{D1} - S_{D2})\mathcal{T}$. However, when setting the number of threads/core into the PRAM range, the delta speedup keeps flat, consistent with the prediction that in PRAM range, the speedup no longer depends on $\mathcal{T}$, but only $S_{D1} - S_{D2}$ which is a constant. The second curve shows the ratio of the performance of sub-block dimension 64 to sub-block dimension 32, indicating a flat line all through the range of $\mathcal{T}$ being varied, since the thread term cancels out.

Effect of machine parameters

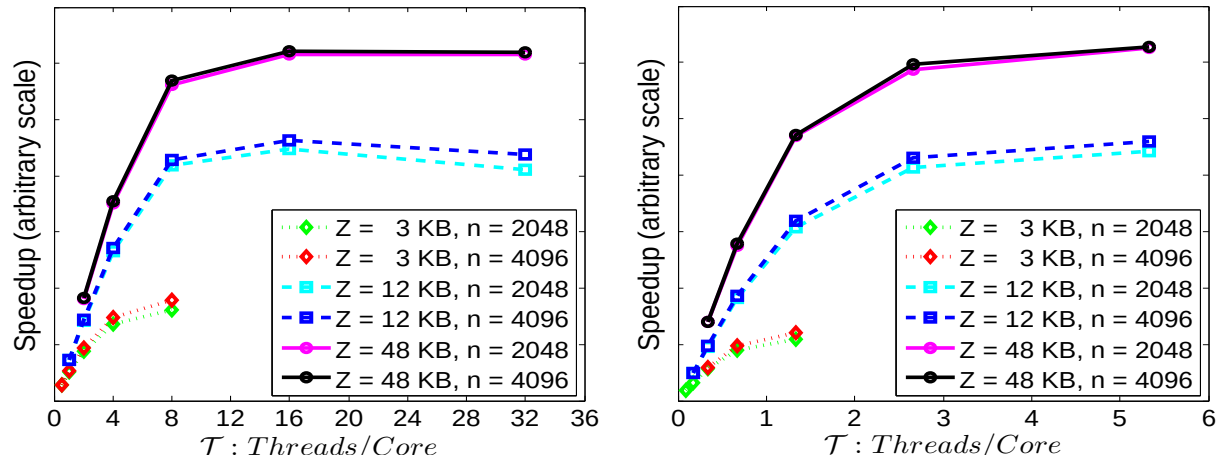
We conducted experiments on the latest NVIDIA GTX680 *Kepler* architecture (Fig. 4.6(b)) in addition to the *Fermi* architecture GTX480 (Fig. 4.6(a)). The figures are qualitatively similar in terms of the effect of threads variation — as $\mathcal{T}$ increases, the speedup increases for a while and then flattens out — as predicted by the TMM model.

We also conducted experiments to evaluate the effect of local memory size Z by artificially decreasing the memory by filling it with dummy data. The TMM model indicates that reducing Z should decrease speedup, which is validated by the experiments on both GPUs. The line for $Z = 3$ KB stops at $\mathcal{T} = 8$ and 1.3 respectively since we can only load one sub-block of size 16×16 in local memory. Each thread handles one location in this sub-block, for a total of 256 threads per multiprocessor. GTX480 and GTX680 have 32 and 192 cores per multiprocessor, respectively, leading to a maximum of 8 and 1.3 threads per core, respectively.

Quantitatively, the machines have different characteristics. GTX680 has a smaller hardware limit on the number of threads per core X.[5] However, it also requires fewer threads per core $\mathcal{T}$ to achieve almost flattened speedup, indicating that memory latency L is small compared to other parameters and can be hidden with fewer threads. However, it appears that the speedup curve is not completely flat even at the maximum number of threads, indicating that it may be beneficial to increase the hardware limit on the number of threads further.

This set of experiments indicates that the TMM model can be used to understand the importance of a variety of parameters on algorithm performance, not only $\mathcal{T}$ and Z, but

[5]On each multiprocessor, GTX480 supports up to 1536 threads on 32 cores; GTX680 supports up to 2048 threads on 192 cores.



(a) Speedup on NVIDIA GTX480 (Fermi). (b) Speedup on NVIDIA GTX680 (Kepler).

Figure 4.6: Speedup of dynamic programming using adjacency matrix for all-pairs shortest paths problem on two generations of NVIDIA GPUs. On GTX480, with memory size 12 KB and 48 KB, using more than 16 threads hides the memory latency completely; on GTX680, due to the hardware limit on $\mathcal{T}$, latencies are not fully hidden, and the speedup curve is still climbing slowly.

also X and L. More importantly, the TMM model can be informative with respect to understanding limits on performance that are due to constraints on machine parameter ranges. For example, the maximum threads per core, X, is lower in the Kepler architecture than in the Fermi architecture, while Fig. 4.6(b) indicates that a larger value of X for Kepler could provide a performance benefit for this all-pairs shortest paths algorithm.

Power of providing more fine-grained and accurate results than the PRAM model for comparing algorithms

It is interesting to compare the the dynamic programming algorithm and Johnson's algorithm with arrays, since the PRAM and the TMM model differ in predicting the relative performance of these algorithms. The PRAM model predicts that Johnson's algorithm should

always be better. However, from Eq. (4.27), for a small number of threads/core working on a dense graph, the TMM model predicts that dynamic programming may be better.

For the graphs with 8K vertices that we explored earlier, $\lg n < \sqrt{Z}$. Consequently, TMM predicts Johnson's algorithm is generally faster than dynamic programming for sparse graphs, but slower for relatively dense ones. Fig. 4.7 demonstrates this effect concretely.

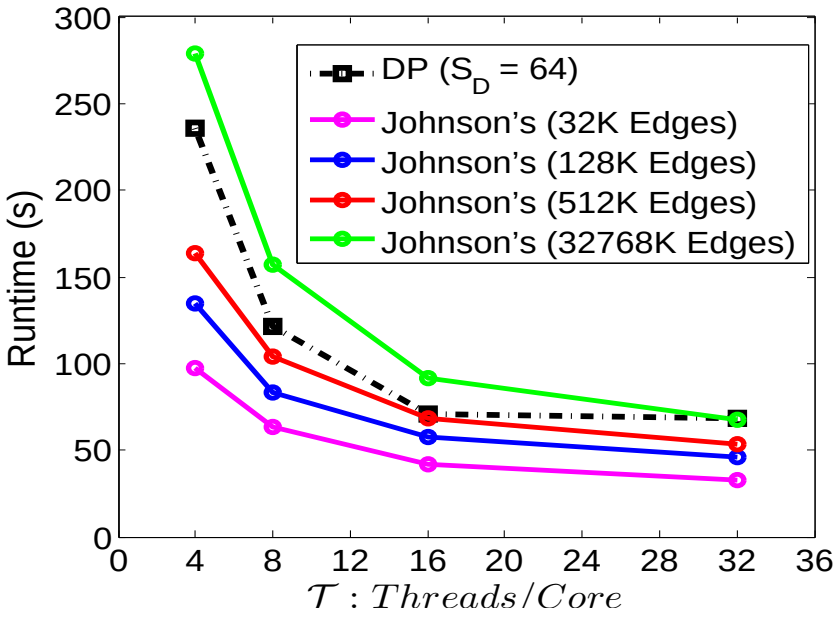


Figure 4.7: Runtime of the dynamic programming (DP) algorithm relative to Johnson's algorithm on a graph with 8K nodes, varying threads/core from 4 to 32 and edges from 32K to 32M.

In addition, for the dense graph, the figure also shows the intersection between the runtime curves of the two algorithms. At that point (32 threads/core), dynamic programming has already been in the PRAM range with stable performance since 16 threads/core, while Johnson's has not. Its runtime is still benefiting by increasing the threads/core. As a result, we predict that Johnson's runtime will flip to be the better one if given sufficient threads. The peak performance of Johnson's being better than that of dynamic programming is consistent with what the PRAM model predicts.

4.2 String Matching

We now consider the string matching problem. In general, the problem of string matching is to find all occurrences of the ***query string*** of length k in a given ***reference string*** of length m ($m \gg k$). Both strings consist of characters from the same alphabet of constant size. We consider ***batch string matching*** where we have a large number n of query strings and want to find matches to all of them in the reference string.

In this problem, an index is precomputed using the reference string for fast matching with query strings. We primarily focus on two types of indexing strategies: suffix tree — a compressed trie containing all the suffixes of the reference string as keys and the starting positions in the string as values, and suffix array — a sorted array of all indices of suffixes in a string. We only focus on batch string matching, assuming that the index has been built in advance.

4.2.1 Suffix Tree

Conceptually, each root-to-leaf path in the tree represents a suffix of the reference string. All leaves store the starting position of the suffix they represent in the reference string. Each edge represents a substring of the suffix along the path, and the outgoing edges for each node represent substrings that start with different letters from that node. Instead of storing entire strings at edges, generally, we only store the starting position and the length of the string for each edge.

In order to explain the search procedure, consider the following example. Say, our query string is '*issips*'. We start at the root, pick the first character i from the query, and follow

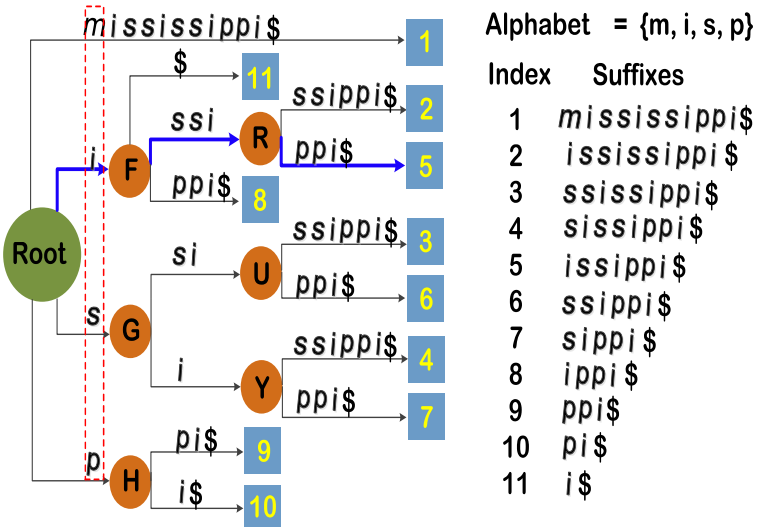


Figure 4.8: Suffix tree for string '*mississippi*'. Each suffix is terminated by the special character \$. Leaves appear immediately after \$, represented by squares and labeled with suffix indices. Circles represent the internal nodes.

the edge that has a string that starts with an i i.e. the second edge leading out of the root in the figure. We have now matched i. Next, we pick the second character s, and follow the edge corresponding to s, which is the second outgoing edge of node F. At this point, we match the string on that edge, '*ssi*', character by character with the query string. We have now matched '*issi*' from the query string and the next character is p. Therefore, we follow the edge corresponding to p and start matching the remaining query string with that edge. We reach a mismatch when we don't find s, and declare that the query string is not a substring of the reference string. If we had reached the end of the query string while continuing to successfully match it, then we would declare a success, and each leaf in the subtree below this node represents an occurrence of the query in the reference. Considering only computational complexity, the suffix tree is optimal. Given any query string of length k, one can match it in $O(k)$ time. If we have n strings, the total work is $T_1 = O(nk)$. Since all the queries can be performed in parallel, the span is $T_\infty = O(k)$.

To calculate the memory complexity,we analyze a particular GPU implementation of suffix trees [48] , where the suffix tree is stored in a flattened form using an array E of size $\alpha \times N$, where N is the number of nodes in the tree. Each node is represented by α cells in this array; in particular, node j is represented by cells $E[(j-1)\alpha]$ to $E[j\alpha - 1]$, each cell corresponding to the α lexicographically ordered letters of the alphabet. Each of these cells essentially represents an outgoing edge of node j; however, some of the cells may be dummies, since a node can have fewer than α outgoing edges. Each non-dummy cell points to a structure called `EdgeInfo` that has three pieces of information: (1) the starting index of the substring (that this edge represents) in the reference string, (2) the length of the substring corresponding to that edge, and (3) the index in the array E at which the destination node's cells range begins.

In order to implement the batch queries, one thread is assigned to each query string, until n (the number of queries) exceeds the maximum number of allowed threads in the machine, at which point, multiple queries are assigned to each thread. Considering only computational complexity, the suffix tree is optimal. Given any query string of length k, one can match it in $O(k)$ time. For n strings, the total work is $T_1 = O(nk)$. Since all the queries can be performed in parallel, the span is $T_\infty = O(k)$. Next we will calculate the memory complexity. For each of the k characters in the query, one can locate its correct edge from the outgoing edges in $O(1)$ time, as the cells are lexicographically ordered. Thus, at most $O(k)$ possible positions in the reference sequence would be checked. The accesses to these positions may have poor locality and therefore cannot be grouped. The number of memory transfers is $M = O(nk)$. So the runtime, using Eq. (3.2), is

$$T_P = O\left(\max(\frac{nk}{P}, k, \frac{nkL}{\mathcal{T}P})\right). \tag{4.28}$$

The last term (memory complexity term) can be refined into two terms depending on the relation between the batch size n and the thread limit XP. When $n \leq XP$, each thread handles a single query; the number of threads $\mathcal{T}P$ increases with n. So, $n = O(\mathcal{T}P)$, and the two cancel in the last term. When $n > XP$, we do not have sufficient threads to run all queries on separate threads. All X possible threads are used ($\mathcal{T} = X$) and the n queries are divided among these threads. Therefore, the third term becomes $\frac{nkL}{XP}$. Considering both scenarios for all possible n, Eq. (4.28) can be expressed as:

$$T_P = O\left(\max(\frac{nk}{P}, k, kL, \frac{nkL}{XP}\right). \tag{4.29}$$

Interpretation of bounds:

- If $L \geq X$, there are two cases:
  - When $n \leq XP$, we have $kL \geq \frac{nkL}{XP} \geq \frac{nk}{P}$. Therefore, the third term (memory complexity for small problem sizes) dominates, and the running time does not depend on n.
  - When $n > XP$, we have $\frac{nkL}{XP} \geq \frac{nk}{P}$ and $\frac{nkL}{XP} > kL$. The runtime is still dominated by memory complexity, but bounded by the last term in Eq. (4.29). The runtime increases linearly with the problem size n.
- If $L < X$, there are also two cases:
  - When $n \leq LP$, we have $kL \geq \frac{nk}{P} > \frac{nkL}{XP}$. Again, the algorithm performance depends on the third term, and the runtime does not depend on n.

– When $n > LP$, we have $\frac{nk}{P} > kL$ and $\frac{nk}{P} > \frac{nkL}{XP}$. The performance becomes dominated by computation complexity, the first term. The runtime increases linearly with n.

4.2.2 Suffix Array

A suffix array is an array of integers giving the starting positions of suffixes of a reference string in lexicographical order [106]. In other words, all the suffixes are indexed by their individual starting positions in the original reference string, and then sorted lexicographically.

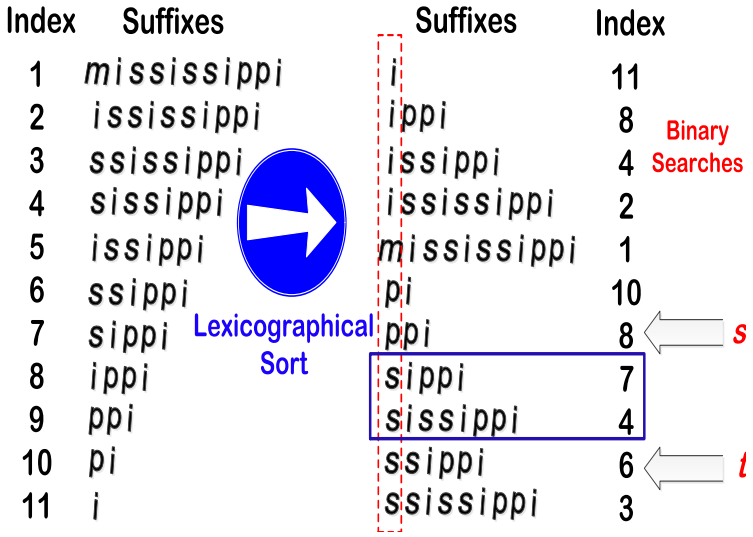


Figure 4.9: Suffix array for string '*mississippi*'. Suffixes are sorted in lexicographical order. s and t are the suffixes immediately ordered before and after the query string '*si*', and located by binary searches. '*sippi*' and '*sissippi*' are the suffixes between s and t, representing all occurrences of the query string.

To match a query string, we perform a binary search over the suffix array. At each step, we compare the query string to the string at that point in the array, and either find a match or decide to go up or down. The search does not stop until the two indices are located such

that all elements between these two in the suffix array are the instances of occurrence for the underlying query string, allowing us to find multiple matches if they exist.

The computational complexity analysis is straightforward. According to the algorithm in [48], each thread matches an individual query in $O(k \lg m)$ span with $O(k \lg m)$ work. Therefore, the total work for an n-sized batch of queries is $T_1 = O(nk \lg m)$ and the span is $T_\infty = O(k \lg m)$. Now for the memory complexity. At each step, the whole query is compared against one entire suffix in the reference string; therefore, this suffix can be accessed in chunks of size C. Therefore, each memory transfer can transfer $O(C)$ characters[6] of the string for comparison. Therefore, the memory complexity for each query is $O(\frac{k \lg m}{C})$ and the memory complexity of the batch is $M = O(\frac{nk \lg m}{C})$. Therefore, using Eq. (3.2) and by the same logic of refining the last term (based on the number of threads) as suffix trees in Eq. (4.29), the runtime can be expressed as:

$$T_P = O\left(\max(\frac{nk \lg m}{P}, k \lg m, \frac{k \lg m \cdot L}{C}, \frac{nk \lg m \cdot L}{CXP})\right). \tag{4.30}$$

Interpretation of bounds:

- If $L \geq CX$, i.e. $X \leq L/C$, there are two cases:
  - When $n \leq XP$, we have $\frac{k \lg m \cdot L}{C} \geq \frac{nk \lg m \cdot L}{CXP} \geq \frac{nk \lg m}{P}$. Therefore, the running time is dominated by memory complexity, and bounded by $\frac{k \lg m \cdot L}{C}$; it does not depend on n.

[6] Each character is represented using a constant number of bits.

- When $n > XP$, we have $\frac{nk \lg m \cdot L}{CXP} \geq \frac{nk \lg m}{P}$ and $\frac{nk \lg m \cdot L}{CXP} > \frac{k \lg m \cdot L}{C}$. Therefore, the performance is still dominated by memory complexity, but is bounded by the last term $\frac{nk \lg m \cdot L}{CXP}$. The runtime should grow linearly as n increases.

- If $L < CX$, i.e. $X > L/C$, there are another two cases:

  - When $n \leq LP/C$, we have $\frac{k \lg m \cdot L}{C} \geq \frac{nk \lg m}{P} > \frac{nk \lg m \cdot L}{CXP}$. The running time is dominated by memory complexity bounded by $\frac{k \lg m \cdot L}{C}$; it does not depend on n.

  - When $n > LP/C$, we have $\frac{nk \lg m}{P} > \frac{k \lg m \cdot L}{C}$ and $\frac{nk \lg m}{P} > \frac{nk \lg m \cdot L}{CXP}$. The running time is dominated by computational complexity (work in particular) and increases linearly with n.

4.2.3 Comparison and Empirical Validation

If we just consider computational complexity, as in the RAM or PRAM model, suffix trees are clearly better than suffix arrays for string matching by a factor of $O(\lg m)$, since their work is smaller. However, in the TMM model, the relationship is not so straightforward. Here we try to understand the interesting conclusions that can be drawn using the TMM analysis of these algorithms.

Let us focus on Eq. (4.29) and Eq. (4.30) to theoretically compare the performance of the two suffix algorithms. We notice a few interesting things about these bounds:

- First, when the number of queries n is small, for both algorithms, the runtime is independent of n, and is only dependent on the memory cost. In this region, whether suffix trees or suffix arrays will perform better depends on the relationship between

terms kL and $\frac{k \lg m \cdot L}{C}$. For most reference strings $\lg m < C$, and suffix arrays are better. Only for very large reference strings will suffix trees be faster.

- As n increases, we reach a point where the running time goes from being independent of n to being linear in n. The point of this transition depends on the characteristics of the machine; particularly the relationship between memory latency L, the limit on the number of threads per processor X, and the memory access width C. Table 4.2 gives the details of the transition points for suffix trees and suffix arrays, indicating which algorithm transitions for smaller values of n. If the machine has large memory latencies L or supports a relatively small limit on number of threads per core X, such that $X \leq \frac{L}{C} < L$, both suffix tree and suffix array running time starts depending on n at the same value of $n = XP$. However, if $X > \frac{L}{C}$, then the suffix array's runtime becomes linear in n for a smaller value of n than the suffix tree.

Table 4.2: Batch size n at which suffix tree and suffix array runtime starts depending on n.

Condition	Problem size n (Suffix Tree)	Problem size n (Suffix Array)
$L \leq X$	LP	$\frac{L}{C} \cdot P$
$\frac{L}{C} < X < L$	XP	$\frac{L}{C} \cdot P$
$X \leq \frac{L}{C}$	XP	XP

- After the transition (the point at which the runtime starts depending on n), the runtime of both the suffix tree and the suffix array increase linearly with problem size n, however at different rates. Whether suffix tree or suffix array runtime increases faster depends on the machine parameters again. When $X \geq L$, both are dominated by computation complexity (suffix tree by $O(\frac{nk}{P})$, and suffix array by $O(\frac{nk \lg m}{P})$). It is clear that the runtime of suffix array grows a factor of $O(\lg m)$ faster than suffix tree. Given sufficiently large n, one expects that suffix trees catch up in performance with suffix

arrays. When $L/C < X < L$, suffix tree is dominated by memory complexity $O(\frac{nkL}{XP})$, while suffix array is still dominated by computation $O(\frac{nk \lg m}{P})$. When $X \leq L/C < L$, they are both bounded by memory complexity (suffix tree by $O(\frac{nkL}{XP})$, and suffix array by $O(\frac{nk \lg mL}{CXP})$). In the second two cases, which one grows faster depends on the reference size m and machine parameters L, X, and C. The asymptotic performance of both after the transition point is shown in Table 4.3 for different machine parameter relationships.

Table 4.3: Bounds for suffix tree and suffix array after the transition point when runtime starts depending on n.

Condition	Bound (Suffix Tree)	Bound (Suffix Array)
$L \leq X$	Compute $\frac{nk}{P}$	Compute $\frac{nk \lg m}{P}$
$\frac{L}{C} < X < L$	Memory $\frac{nkL}{XP}$	Compute $\frac{nk \lg m}{P}$
$X \leq \frac{L}{C}$	Memory $\frac{nkL}{XP}$	Memory $\frac{nk \lg mL}{CXP}$

Comparison with empirical data: Consider the empirical performance of these algorithms as reported by Encarnaijao et al. [48]. We have reproduced the graph in Fig. 4.10 from their data. The figure supports all three predictions above, at least qualitatively.

For small values of n, both algorithms are dominated by memory complexity, and the running times of both suffix tree and array remain flat as n increases, showing that the performance is independent of n and depends on the third term of the running time bounds in Eq. (4.29) and Eq. (4.30). In addition, for these experiments, the reference sequence is also of a moderate length ($m = 10^7$), and the alphabet size is small; therefore, we expect that $C > \lg m$, suffix arrays perform better within the flat range.

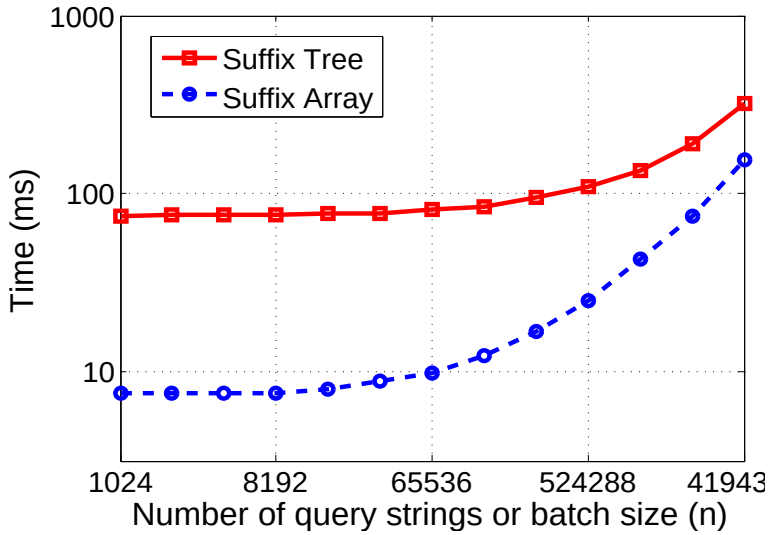


Figure 4.10: Performance of suffix trees and suffix arrays on GPU. Empirical data are from Encarnaijao et al. [48].

For the GPU being used (NVIDIA GTX 580), the hardware limit on the number of threads X is large and of the same order as the latency[7]. Therefore, the suffix array's transition (where the running time starts depending on n) happens for smaller values of n than the suffix tree. This fact is also consistent with the empirical observations. As n increases, both performance curves ultimately bend up. The bend occurs at different points for the two algorithms, and as predicted above, the suffix array curve bends sooner than the suffix tree curve.

After the transition of algorithm performance to be linear with n, the runtimes of the two algorithms increase with different rates. For the first case, the running time for suffix array increases faster than suffix tree. If we are in the second case, for these machines, we expect $L/X < \lg m$, since it is expected to be small. Therefore, for both the first and the second cases, suffix array running time grows faster than the suffix tree running time. In the figure,

[7] $X = 48$; Memory latency is 400-500 cycles, while arithmetic instructions take 4 cycles each, so L is about 100.

as predicted, the slope of the curve for suffix arrays is steeper than the slope for suffix trees after the transition. Even though the curve does not go that far, we can speculate that eventually, for large enough n, suffix trees will outperform suffix arrays.

Therefore, the choice of a particular data structure and a corresponding algorithm depends on how well they match the characteristics and features of the target hardware, and that is especially true for highly-threaded many-core GPUs. Although the asymptotic search time of suffix array $O(k \lg m)$ is greater than that of the suffix tree $O(k)$, experimental results from real implementations on GPUs show that computational complexity is not the only factor. To choose the appropriate algorithm for a particular machine, one must consider the relationship between machine parameters L, X, and C, and even the relationship between algorithmic parameters and machine parameters, $\lg m$ and C in this case.

4.3 Fast Fourier Transform (FFT)

The ***Discrete Fourier Transform (DFT)*** is a classic algorithm that is widely used in many applications such as digital imaging, signal processing, and convolution, etc. The DFT is obtained by decomposing a sequence of values into components of different frequencies. For an N-point complex sequence $x = x_0, ...x_{N-1}$, its DFT is an N-point complex sequence of $X = X_0, ..., X_{N-1}$, where

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi k \frac{n}{N}}, \qquad k = 0, ...N-1. \tag{4.31}$$

Fast Fourier Transform (FFT) is an algorithm that computes DFT in $O(N \lg N)$ operations. At each of $\lg N$ recursive steps, it divides the DFT of size N into two interleaved

DFTs of size $N/2$, followed by a combining stage consisting of $N/2$ size-2 DFTs called radix-2 'butterfly'. In each butterfly, one element of a DFT will have one addition and one subtraction operation with the element of same index in the other DFT.

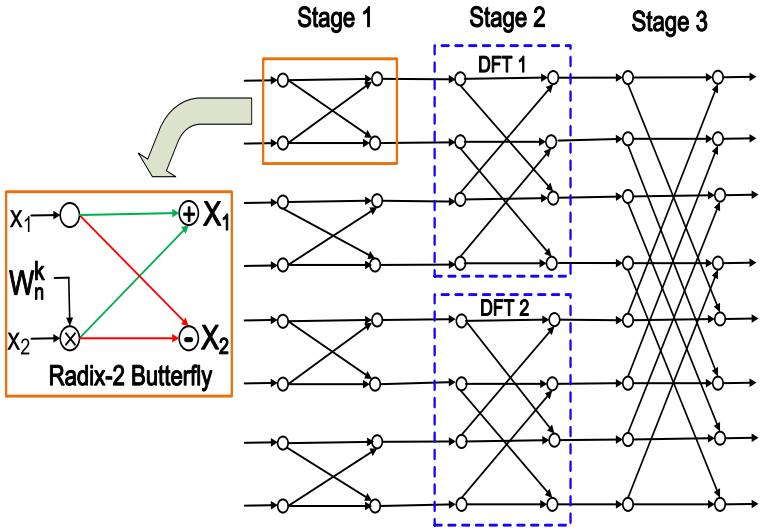


Figure 4.11: Data path and computation pattern of FFT. Radix-2 butterfly is the basic computation unit of FFT.

Next we analyze the FFT algorithm on GPUs. At each of the $\lg N$ steps, $N/2$ 2-point FFTs are computed in parallel, each by a thread. Therefore, the work is $T_1 = O(N \lg N)$ and the span is $T_\infty = O(\lg N)$. For each stage of the butterfly, we have to perform $O(N)$ memory accesses, but these are predictable at regular intervals. Therefore, they can be grouped into chunks of size C, and the total number of memory transfers is $M = O(N \lg N/C))$. Using Eq. (3.2) and by refining the last term (based on relationship between N and $\mathcal{T}$) as in suffix trees/arrays, we get the running time of

$$T_P = O\left(\max(\frac{N \lg N}{P}, \lg N, \frac{\lg N \cdot L}{C}, \frac{N \lg N \cdot L}{CXP})\right). \tag{4.32}$$

Comparison with empirical data: We compare our analytical observations with the empirical result of FFT algorithm with radix-2 Stockham algorithm on GPUs [60]. Here, we would like to highlight one interesting experiment, where they varied the memory clock rate. Note that increasing the memory clock rate is equivalent to shrinking the memory latency L. In Eq. (4.32), L affects the runtime only as long as the algorithm runtime is dominated by the memory complexity term (the last two terms). As L shrinks, eventually it is small enough that the algorithm runtime is no longer dominated by the memory complexity; instead it is dominated by the computation complexity (the first term). Therefore, after a certain point of increasing memory frequency, the algorithm runtime no longer depends on L, and further increases in frequency have no affect. Fig. 4.12 shows the running times for 2 problem sizes as the memory frequency increases. The graph validates the above observation; after a certain point, increasing the memory clock rate (decreasing L) does not affect the runtime, and the curve becomes flat.

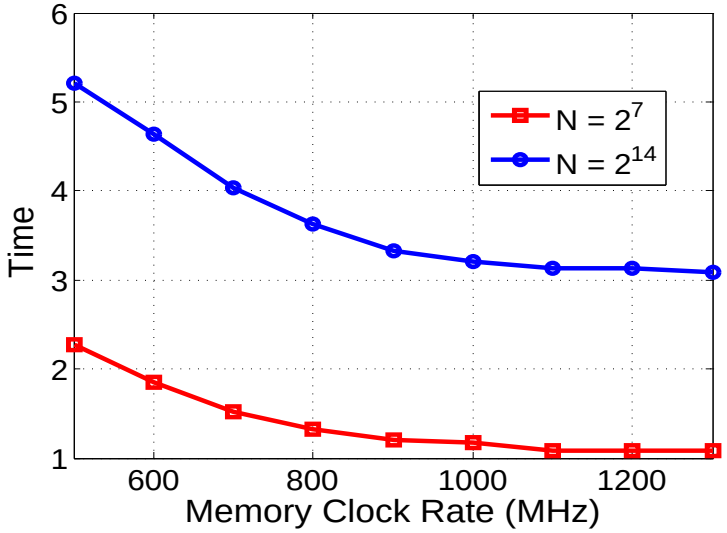


Figure 4.12: Runtime of FFT algorithm with various memory frequencies on an NVIDIA GTX280. The FFTs are performed for two problem sizes $N = 2^7$ and $N = 2^{14}$. The y-axis is the runtime plotted on an arbitrary scale, as the runtime data are converted from GFLOPs from Govindaraju et al. [60]. The x-axis shows increasing memory clock rate, denoting decreasing memory latency L.

4.4 Merge Sort

Merge sort is generally considered to be the preferred technique for external sorting, where the sequence being sorted is stored in a large external memory and the processor only has direct access to a much smaller local memory. In particular, it is asymptotically optimal in the disk access model [3]. In this section, we analyze the merge sort of Satish et al. [134].

The idea of the merge sort algorithm is to divide the input sequence into blocks of size Z, sort them in parallel locally within core groups so as to utilize the fast memory using odd-even merge sort [83], and finally recursively merge them using the ***blocked merge***. The blocked merge is an important subroutine of merge sort. It merges two sorted sequences A and B into a new sorted sequence S of size n that contains all the elements of A and B. We first analyze the algorithm for blocked merge and then use it to analyze merge sort.

4.4.1 Blocked Merge

The idea of this algorithm is to decompose the overall problem into many smaller problems, each of which is small enough to fit in fast memory. Therefore, sequences A and B are cut into smaller sequences as follows: first divide the sequences A and B into sections of size $Z/2$; say that the boundary elements are $a_1, a_2, ..., a_{2n/Z}$ and $b_1, b_2, ..., b_{2n/Z}$ respectively. We then, in parallel, search for these boundary elements in the other sequence. If we include the boundary elements and these new binary search locations, this divides both sequences into $4n/Z$ sections, each of size at most $Z/2$ and the corresponding sections match up. We can individually merge the ith section in A to the ith section in B and this leads to the correct final sequence S. The process of creating this partition involves n/Z parallel binary searches;

leading to a total of $T_\infty = O(\lg n)$ span, $T_1 = O(\frac{n}{Z} \lg n)$ work, and incurs $M = O(\frac{n}{Z} \lg n)$ memory transfers, since the unpredictable memory accesses cannot be grouped.

Once subsequences of both A and B fit in memory, $n < Z$, we can merge the many subsequences in parallel. In order to For each element $x \in A \cup B$ we compute the rank $rank(x, S)$, which is equal to $rank(x, A) + rank(x, B)$. If $x \in A$, it's rank in A is simply its index in A, since A is sorted. Its rank in B can be found by doing a binary search. Therefore, the total span is $T_\infty = O(\lg n)$ with $T_1 = O(n \lg n)$ work and $M = O(n/C)$ memory transfers (the sequence must be brought into memory once).

Since there are totally n/Z blocks that can be merged in parallel, the total span is $T_\infty = O(\lg Z)$, with $T_1 = O(n \lg Z)$ work and $M = O(n/C)$. Thus, adding the cost of partitioning and the local merges, and applying Eq. (3.2), we get

$$T_P = O\left(\max(\frac{n \lg Z + \frac{n}{Z} \lg n}{P}, \lg n, \frac{(\frac{n}{C} + \frac{n}{Z} \lg n)L}{\mathcal{T}P})\right). \tag{4.33}$$

If $\lg n \ll Z$, as is the case for many machines,

$$T_P = O\left(\max(\frac{n \lg Z}{P}, \lg n, \frac{nL}{C\mathcal{T}P})\right). \tag{4.34}$$

4.4.2 Merge Sort

The odd-even merge sort [15] is based on a bitonic sorting network and proceeds in $O(\lg n)$ phases for a sequence of size n. In each phase, there is $O(n \lg n)$ work and $O(\lg n)$ span. Therefore, the total work is $O(n \lg^2 n)$ and the total span is $O(\lg^2 n)$. For subsequences of size Z, the odd-even sorting step takes $O(\lg^2 Z)$ span, $O(Z \lg^2 Z)$ work, and $O(Z/C)$ memory

transfers. Given $O(n/Z)$ blocks to be sorted, the span is $O(\lg^2 Z)$, work is $O(n \lg^2 Z)$ with $O(n/C)$ memory transfers. Given that $Z = \Omega(Z)$, $T_1 = O(n \lg^2 Z)$, $T_\infty = O(\lg^2 Z)$.

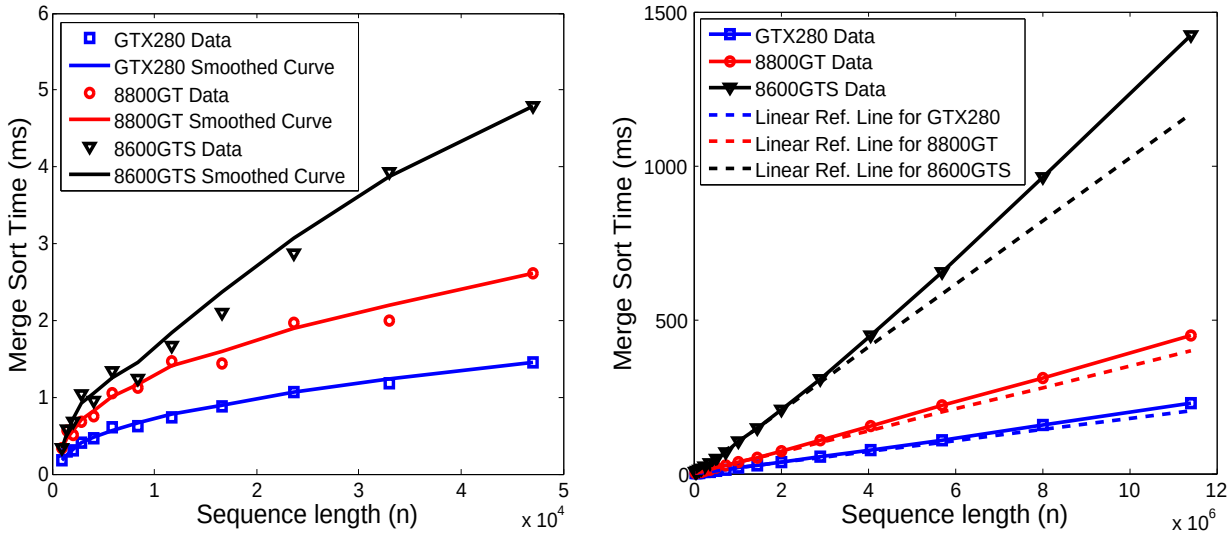
Next, the $O(n/Z)$ subsequences are merged in a parallel pair-wise merge tree of $O(\lg(n/Z))$ depth. As we move down the tree, the sequences that must be merged stop fitting in local memory, and we must use the merge algorithm described in Section 4.4.1. Therefore, each layer of the tree has work $O(n \lg Z)$, span $O(\lg n)$ and memory complexity $O(n/C)$. We multiply them by $\lg(n/Z)$ levels to get the final computational and memory complexity.

Therefore, ultimately we can combine the sorting and the merging step. Since $O(\lg(n/Z)) = O(\lg n - \lg Z) = \Omega(\lg Z)$, we can substitute into Eq. (3.2) and by refining the last term according to problem size to get the runtime:

$$T_P = O\left(\max(\frac{n \lg Z \lg \frac{n}{Z}}{P}, \lg n \lg \frac{n}{Z}, \frac{\lg \frac{n}{Z} \cdot L}{C}, \frac{n \lg \frac{n}{Z} \cdot L}{CXP})\right). \tag{4.35}$$

For small values of $n < XP$, the third term (representing memory complexity) dominates performance, and the running time increases logarithmically, that is very slowly. As n gets larger, the performance is bounded by either the first or the last term depending on the relationship between C, Z and X. In both cases, the running time increases with $n \lg \frac{n}{Z}$, that is, a little faster than linearly with n.

Comparison with empirical data: Empirical results in Fig. 4.13(a) and Fig. 4.13(b) are re-plotted from the data represented in [134]. First let us look at small values of n in Fig. 4.13(a). In this range, we see that the running time increases very slowly with n, as expected, since it depends on the third term in Eq. (4.35). If we look at large values of n, however, in Fig. 4.13(b), we see that the running time increases a little faster than linearly with n (the dashed lines show linear growth for comparison), which we can speculate is



(a) Runtime for small sequence size.

(b) Runtime for large sequence size.

Figure 4.13: Merge sort on multiple GPUs (data from [134]); Solid lines are smoothed curves from data and dotted lines are linear references. (a) For small n, the runtime increases slower than linearly with n. (b) For large n, the runtime increases faster than linearly with n.

approximately $n \lg (n/Z)$. The TMM model is able to predict the growth of running time for both small and large sequences. In addition, it indicates that increasing the hardware limit on threads (increasing X) is likely to increase the area where the growth is slow.

4.5 List Ranking

List ranking is a classic problem where we want to compute the rank of each element in a list in parallel. Here we analyze the performance of Wyllie's algorithm [163] in the TMM model. In Wylie's algorithm, each element's rank is computed in parallel by repeated pointer jumping; the successor pointer of each element in the list is repeatedly updated to its successor's successor, while also updating the rank estimate. Given a linked list of n elements, the algorithm finishes in $T_\infty = O(\lg n)$ span, thereby making the total work $T_1 = O(n \lg n)$.

Rehman [131] implement this algorithm on GPU, assigning one thread to each element in the list. Each thread accesses $O(\lg n)$ elements, and these memory accesses cannot be grouped, since they are far away from each other. So the number of memory transfers is $M = O(n \lg n)$. Refining the last term by considering all possible n, we get the runtime from Eq. (3.2):

$$T_P = O\left(\max(\frac{n \lg n}{P}, \lg n, \lg n \cdot L, \frac{n \lg n \cdot L}{XP})\right). \tag{4.36}$$

At a small list size when $n < \min(LP, XP)$, the third term in Eq. (4.36) dominates the performance. Therefore the runtime is linear with $\lg n$. As n increases, performance may be dominated by memory complexity due to the last term or computation complexity due to the first term; it depends on the relationship between L and X. Specifically, when $L > X$, and $n > XP$, the memory complexity dominates; when $L < X$ and $n > LP$, the computation complexity dominates. However, in both cases, the runtime increases with $n \lg n$.

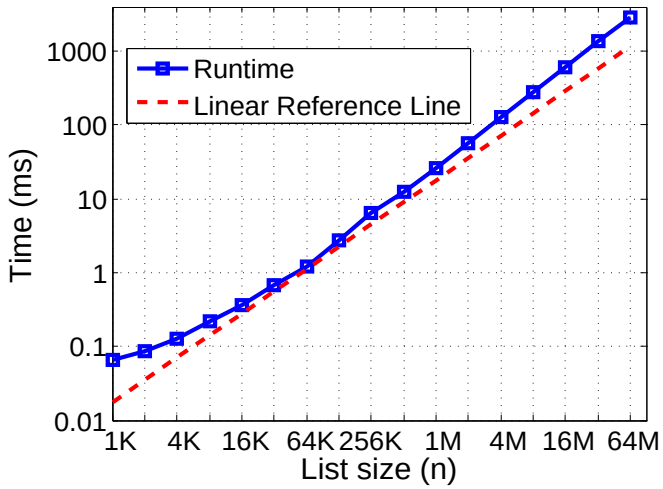


Figure 4.14: Runtime of Wyllie's algorithm on NVIDIA GTX 280 (data from [131]). The runtime grows slowly for small n and faster for larger n (dotted line is a linear reference). Note that the graph is a log-log plot in order to expose the trends over a wide range of n.

Comparison with empirical data: These observations are borne out with the empirical data which is re-plotted in Fig. 4.14 from the data presented in [131]. The runtime increases slowly for small values of n and faster for larger values of n.

4.6 Analysis of Additional Algorithms

In this section, we briefly present the bounds on other classic algorithms in the TMM model. These bounds are listed in Table 4.4. We do not describe these algorithms, but most of them are either well-known primitives (reduce, scan, merge) or classic algorithms for well-understood problems (such as connected components, minimum spanning tree, sorting). For completeness, we include the bounds of the algorithms we analyzed in this chapter. Note that the bounds on M shown are for large n; we are not presenting the detailed analysis for all ranges of n as we did above. n is generally the problem size; for graph algorithms, n is the number of nodes and m is the number of edges; for string matching, n is the number of query strings, m is the length of the reference string, and k is the maximum query length.

Table 4.4: Analysis for some more classic algorithms.

Algorithms	**Work** T_1	**Span** T_∞	**Mem. Ops** M
Reduce [136]	n	$\lg n$	n/C
Scan [136]	n	$\lg n$	n/C
Merge [134]	$n \lg Z$	$\lg Z$	n/C
Merge Sort [134]	$n \lg Z \lg \frac{n}{Z}$	$\lg Z \lg \frac{n}{Z}$	$\frac{n}{C} \lg \frac{n}{Z}$
Odd-even Sort [83]	$n \lg^2 n$	$\lg^2 n$	$\frac{n}{C} \lg \frac{n}{Z}$
ConnectedComp [143]	$(m+n) \lg n$	$\lg^2 n$	$\frac{m+n}{C} \lg n$
MST-Boruvaka [151]	$m \lg n$	$\lg n$	$m \lg n$
Suffix Tree [48]	nk	k	nk
Suffix Array [48]	$nk \lg m$	$k \lg m$	$nk \lg m/C$
FFT [60]	$n \lg n$	$\lg n$	$n \lg n/C$
List Ranking [131]	$n \lg n$	$\lg n$	$n \lg n$
APSP [101]	$n^3 \lg n$	$n \lg n$	$n^3 \lg n/(\sqrt{Z}C)$

Chapter 5

Calibrated Performance Model

In this chapter, we propose an analytic model that helps improve the understanding of the performance of memory-limited kernels on GPUs, especially as impacted by cache and various configuration parameters that can be used to tune kernel execution, such as the number of thread blocks and the number of threads per block. Distinct from the TMM model introduced in Chapter 3, this calibrated performance model is a lower-level model that is designed to quantitatively predict real runtime throughout the entire configuration space by only one real run, with scale factors. We utilize both throughput (number of input data elements processed per unit time) and execution time (time to process a fixed size data set) as the performance metrics of interest. Our focus is on applications whose performance is dominated by memory access bandwidth, either to the shared memory or to the global memory. We concentrate on the throughput of the kernel executing on the graphics engine itself, leaving the performance assessment of data transfers to and from the graphics engine for future work.

The model is first explored through the use of a synthetic micro-benchmark, which is then followed by an empirical validation using a pair of production applications used in computational biology. (In the two applications used for empirical evaluation, the performance is

dominated by kernel execution time.) A parallel Bloom filter algorithm is also described and implemented as a benchmark to validate the model and also quantify the tradeoff between false positive rates and runtime.

5.1 Performance Modeling

In terms of performance, both the CUDA and OpenCL development environments support application implementation using familiar languages, and the available debugging and performance monitoring tools provide substantial information about correctness and execution speed. An important issue that remains, however, is a comprehensive understanding of what algorithmic and architectural features have significant impact on the performance of particular applications. Getting a code running correctly is not difficult, but getting it to perform well can still be quite a challenge. Ultimately, the achieved performance for an application is a complicated interaction between a variety of parameters, some set by the application developer, others imposed by the architecture of the particular graphics engine being used.

We first present an analytic base model [105] that describes the performance of a Bloom filter [21] in Section 5.1.1. We constrain the number of requested thread blocks to evenly divide the work across the multiprocessors. Then in Section 5.1.2, we extend the base model by lifting the restriction, characterizing application performance over a wider parameter space than the base model and a larger set of applications. We also add support for modeling caches, incorporating the effect of cache misses into the performance expressions [104].

5.1.1 Base Model

Before the modeling, we first recall several important concept in describing how GPUs work and get scheduled. As previously introduced in Section 3.1.1, a GPU consists of a number of *streaming multiprocessors* with multiple *cores* executing in *Single Instruction Multiple Thread (SIMT)* fashion. Instructions are issued per *warp*, which is a consecutive batch of threads executing the same instruction while on different data. Context switches happen between warps. When an operand of a warp is not ready, this warp will be stalled and switched out by another warp of threads very fast. On each multiprocessor, there are a dynamically-determined number of thread blocks scheduled depending on the resources being used per thread block. On-chip Resources including registers and shared memory are finite. An entire thread block will be assigned to a multiprocessor and become *active block* only when the resources requested by the entire thread block are less than the resources remained on-chip. *Occupancy*, defined in Eq. (5.1), is used to describe the efficiency of resource allocation and schedule. Higher occupancy indicates higher utilization of on-chip resources.

$$Occupancy = \frac{ActiveBlocks}{MaximumActiveBlocksAllowed} \tag{5.1}$$

We characterize application performance in terms of a variety of parameters, with application parameters summarized in Table 5.1 and architecture parameters summarized in Table 5.2.

Table 5.1: Application Parameters

Parameter	Description
n	Data set size
k	Number of hashing functions
n_e	Number of elements in Bloom vectors
m	Working set size
n_{sub}	Size of decomposed sub-problem
R_T	Number of registers per thread
Z_B	Shared memory used per block (in bytes)
B_r	Requested number of blocks (total)
T_r	Requested number of threads per block

Table 5.2: Architecture Parameters

Parameter	Description
P	Number of processors (cores)
Q	Number of cores per multiprocessor
Z	Shared memory per multiprocessor (in bytes)
R	Number of registers per multiprocessor
W	Warp size (in number of threads)
C_s	Cache size
N_W	Min number of warps
$B_{\max}$	Max number of blocks (total)
T_{maxB}	Max number of threads per block
T_{maxMP}	Max number of threads per multiprocessor

When a kernel is launched, the application specifies a request for a number of blocks, B_r, and a number of threads per block, T_r. Together, these two parameters determine the occupancy of the multiprocessors in the graphics engine. First, we want to know how many active blocks can be launched on each multiprocessor, denoted B_a. This can be expressed in terms of the register usage, shared memory usage, and fixed device capability (max active blocks allowed and max threads allowed). The resultant active blocks B_a will be the smallest among the

four terms in Eq. (5.2).

$$B_a = \min\left(\left\lfloor \frac{Z}{Z_B} \right\rfloor, \left\lfloor \frac{R}{R_T \times T_r} \right\rfloor, \left\lfloor \frac{B_{\max}}{P/Q} \right\rfloor, \left\lfloor \frac{T_{\max\mathrm{MP}}}{T_r} \right\rfloor\right) \tag{5.2}$$

Second, throughput is maximized when the requested number of blocks B_r is an integer multiple of the product of B_a and P/Q (number of multiprocessors), thereby balancing the number of blocks allocated to each multiprocessor. Here, B_{opt} is the set of possible block request counts that is required to yield peak (optimal) performance.

$$B_{\text{opt}} = \{B_r = i \times B_a \times P/Q \mid i \in \mathbb{N}\} \tag{5.3}$$

Third, in a similar manner, the number of threads per block T_r should be an integer multiple of the warp size W, forming a set of possible requested threads per block necessary for peak (optimal) performance, T_{opt}.

$$T_{\text{opt}} = \{T_r = j \times W \mid j \in \mathbb{N}\} \tag{5.4}$$

Finally, we define a set of Boolean indicators that encode whether or not the requested number of blocks is in the optimal set and is feasible given the practical constraints of the engine (denoted by A_B), and whether or not the requested number of threads per block meets similar conditions (denoted by A_T).

$$\begin{aligned} A_B \quad = \quad & B_r \in B_{\text{opt}} \wedge \\ & B_r \leq B_{\max} \end{aligned} \tag{5.5}$$

$$
\begin{aligned}
A_T \;=\; & T_r \in T_{\text{opt}} \wedge \\
& T_r \geq N_W \times W \wedge \\
& T_r \leq \min\left(T_{\text{maxB}}, \frac{T_{\text{maxMP}}}{B_a}\right) \wedge \\
& T_r \geq \frac{\frac{R}{R_t}}{\frac{B_r}{P/Q}+1}
\end{aligned}
\tag{5.6}
$$

Beyond membership in B_{opt}, the only constraint on the number of blocks is that it is within the count allowed by the system. For the number of threads per block, constraints include a minimum number (to mask latencies and provide sufficient parallelism) as well as an upper bound based on resource limits. We address the performance achievable when there are insufficient threads to mask memory latencies in the next chapter.

5.1.2 Model Extension

Additional variables used in the models are listed in Table 5.3. What follows extends the model from Section 5.1.1. Each application's peak performance is first described in terms of its algorithmic complexity. The algorithmic complexity is expressed via a function $f_{\text{app}}(\overrightarrow{\text{algo}}, \overrightarrow{\text{inpt}})$, defined in terms of an algorithm parameter vector $\overrightarrow{\text{algo}}$ and an input size vector $\overrightarrow{\text{inpt}}$. $\overrightarrow{\text{algo}}$ includes parameters from the algorithm design and implementation, e.g., number of hashing functions in a Bloom filter, size of a sub-block computation, etc. $\overrightarrow{\text{inpt}}$ takes the parameters relevant to the input problem size and working set size. The form of f_{app} is, of course, application specific. Specific examples will be provided in the sections below. f_{app} can be regarded as a general adapter of the model to different problem sizes, algorithms, and even to different implementations of each algorithm.

Table 5.3: Model Variables

Variable	Description
B_a	Active number of blocks per multiprocessor
A_B	Optimal block number indicator (Boolean)
A_T	Optimal thread number indicator (Boolean)
A_C	Working set fit in cache indicator (Boolean)
B_{opt}	Set of optimal numbers of blocks (total)
T_{opt}	Set of optimal numbers of threads per block
f_{app}	Application algorithmic complexity
f_{cache}	Cache factor
f_{sched}	Block scheduling factor
r_H	Cache hit rate
r_M	Cache miss rate
G	Ratio of cache to main memory throughput
$\overrightarrow{\text{algo}}$	Vector of algorithm parameters
$\overrightarrow{\text{inpt}}$	Vector of input size parameters
$Time$	Execution time (in seconds)
$Time_{\text{min}}$	Shortest execution time (in seconds)
$Tput$	Throughput (in data elements per second)

Similar to the indicators A_B and A_T (described above) that articulate whether or not a kernel's *configuration* (the combination of B_r and T_r) is optimal, an additional indicator A_C can be used to articulate whether or not the kernel's working set m fits in on-chip memory spaces, either shared memory or L1 cache. If the working set is allocated to and fits in shared memory, A_C is true because each reference to the working set is a hit. If shared memory isn't used, A_C is true when the working set fits in L1 cache. This can be expressed as:

$$A_C = \begin{cases} m < Z & \text{if using shared memory} \\ m < C_s & \text{if using global memory.} \end{cases} \tag{5.7}$$

We now articulate the peak performance of an application given $\overrightarrow{\text{algo}}$ and $\overrightarrow{\text{inpt}}$. If and only if A_B, A_T and A_C are true does the kernel configuration provide peak performance.

$$Time_{\min} \propto f_{\text{app}}(\overrightarrow{\text{algo}}, \overrightarrow{\text{inpt}}) \quad \text{if } A_T \wedge A_B \wedge A_C \tag{5.8}$$

Moving from peak performance, we next extend Eq. (5.8) to incorporate the effects of cache and of block scheduling. Here, f_{cache} reflects the performance impact due to cache misses and f_{sched} reflects that due to the scheduling of blocks. We consider each factor in turn.

$$Time \propto f_{\text{app}}(\overrightarrow{\text{algo}}, \overrightarrow{\text{inpt}}) \times f_{\text{cache}} \times f_{\text{sched}} \quad \text{if } A_T \tag{5.9}$$

Assuming that our memory access patterns are random (due to hashing), a simple model considering the size of cache C_s and working set m for the cache hit rate is:

$$r_H = \min(1, \frac{C_s}{m}). \tag{5.10}$$

The above expression yields a hit rate of 1 when the working set size m is smaller than the cache size C_s. As m exceeds C_s, the hit rate is modeled as their ratio (reflecting the random access assumption). Given a hit rate r_H, the miss rate is straightforward to model:

$$r_M = 1 - r_H \tag{5.11}$$

We complete the cache performance model by expressing the cache factor as a linear combination of execution times that are blended by cache hit and miss rates:

$$f_{\text{cache}} = \begin{cases} 1 & \text{if } A_C \\ r_H + r_M \times G & \text{otherwise,} \end{cases} \tag{5.12}$$

where G reflects the multiplicative slowdown experienced with very low cache hit rates. In principle, one would like to express G in terms of the relative performance of the cache and the global memory. In practice, however, their relative performance difference is masked by the large number of threads supported. In this work, G is empirically determined.

In a similar manner to the cache effects, f_{sched} models the impact of block scheduling on the application performance, extending the overall model to predict execution time for numbers of requested blocks that are not only within the set B_{opt}, but also those outside of it.

$$f_{\text{sched}} = \frac{\lceil \frac{B_r}{B_a \times P/Q} \rceil \times B_a \times P/Q}{B_r} \tag{5.13}$$

The above expression reflects the block scheduling process on the multiprocessors, with the time determined by the multiprocessor with the largest number of blocks assigned to it (expressed via the ceiling function $\left\lceil \frac{B_r}{B_a \times P/Q} \right\rceil$). As we will see below, this yields a distinctive zigzag pattern in the throughput as the number of requested blocks is varied.

Given an expression for execution time in Eq. (5.9), we can describe processing throughput in terms of the data set size n and the execution time.

$$Tput = n/Time \tag{5.14}$$

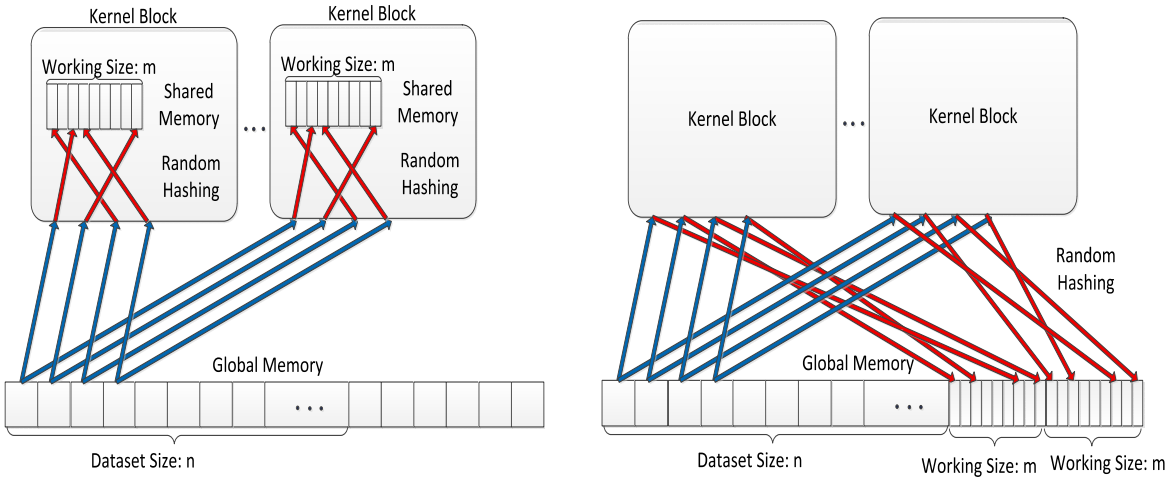
5.2 Model Application

In this section, the calibrated performance model is first validated using a synthetic micro-benchmark which allows us to quantitatively explore the impact of random memory access patterns in terms of cache effects and varied kernel configuration options. Two real applications from computational bioinformatics are also used to further examine the model's effectiveness. Given the excellent match between model predictions and empirical measurements, we conclude that the model can be effectively used not only to understand the performance of existing applications, but it can also be used to help configure the tuning parameters that must be set when executing any graphics engine kernel. The model confirms that, in general, shared memory is better suited to handling random memory access patterns. However, given sufficient parallelism and a small enough working set, even random access to global memory can be effective.

5.2.1 Synthetic Micro-benchmark for Hashing

Given the stated goal of understanding the performance of applications with poor memory access patterns, we present a synthetic micro-benchmark that allows us to quantitatively explore the impact of random access patterns on application throughput. The computation is intentionally simple enough to ensure that memory accesses dominate its performance.

The choice of memory subsystem potentially has a significant performance impact, especially for randomly distributed accesses. In the present work, we focus on the shared memory and global memory subsystems, leaving the constant memory and texture memory for future work.



(a) Shared memory random accesses. (b) Global memory random accesses.

Figure 5.1: Micro-benchmark for random hashing on GPU architectures. (a) Hash table on shared memory. (b) Hash table on global memory.

Fig. 5.1 helps us understand the operation of the micro-benchmark. Random numbers within a specified address range (i.e., working set size) are populated initially in the global memory. Then, as illustrated in Fig. 5.1(a), each thread reads an individual data element, interprets that data element as a (random) pointer to a (synthetic) hash table in shared memory, and fetches the value from the table. Each block performs the same pattern of accesses, working with an assigned range of the pointers stored in global memory. In a variation of the micro-benchmark, illustrated in Fig. 5.1(b), the random pointers point to a (synthetic) hash table in global memory. In both cases, the size of the hash table (the working set size) is denoted by m.

We are now in a position to formulate a function f_{app} for the above micro-benchmark. Here, there are no parameters appropriate for $\overrightarrow{\text{algo}}$, so it is empty. The only input parameter is the data set size n, so $\overrightarrow{\text{inpt}} = (n)$ and thereby

$$f_{\text{app}}(n) = n \tag{5.15}$$

and according to Eq. (5.9),

$$Time \propto n \times \frac{\lceil \frac{B_r}{B_a \times P/Q} \rceil \times B_a \times P/Q}{B_r} \quad \text{if } A_T \wedge A_C \ . \tag{5.16}$$

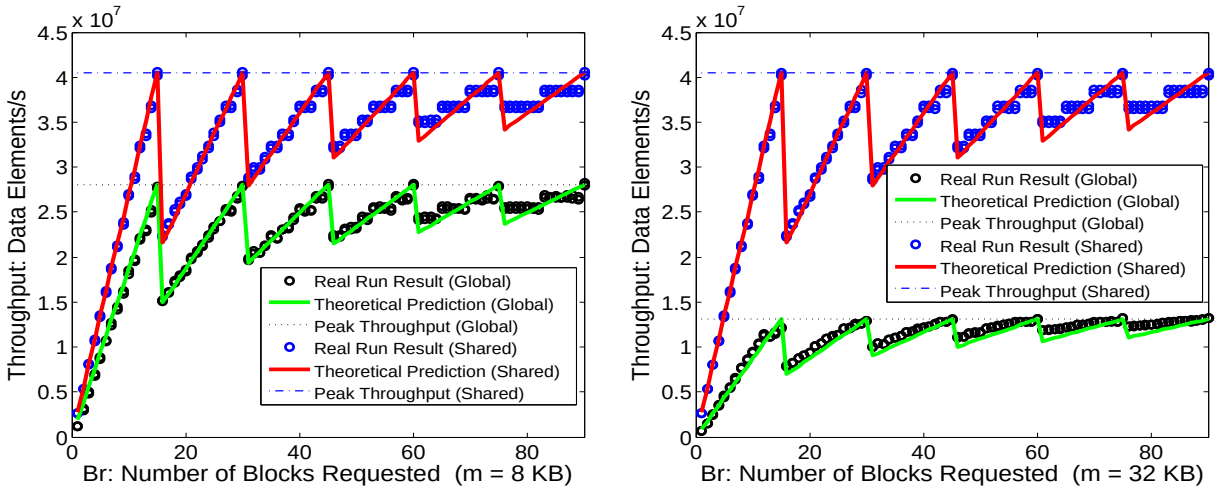
To investigate how the block scheduling factor f_{sched} influences the runtime, a small working set size is used to guarantee it fits on-chip, either in shared memory or L1 cache, so that A_C is true. The impact of the choice of memory subsystem accessed (whether shared or global) will be represented in the proportionality constant, which will be explored empirically.

The graphics engine used for this investigation is the NVIDIA GTX480, which has 15 streaming multiprocessors, $P/Q = 15$, i.e. there are totally $P = 480$ cores, and each multiprocessor has $Q = 32$ cores. The GTX480 has 1.5 GB off-chip global memory. Other architecture parameters are presented in Table 5.4.

Table 5.4: NVIDIA GTX 480 Architecture Specification

Parameter	Specification
P	480
Q	32
Z	16 KB or 48 KB (configurable)
C_s	48 KB or 16 KB (configurable)
R	32768
W	32
N_W	6 (NVIDIA recommended)
$B_{\max}$	120
T_{maxB}	1024
T_{maxMP}	1536

We first empirically investigate how a range of choices for B_r influences the throughput for $T_r = \{960, 1024\} \subseteq T_{\text{opt}}$. The data set is 2^{25} random 4-byte words. In Fig. 5.2(a), we use a



(a) Working set size $m = 8$ KB. (b) Working set size $m = 32$ KB.

Figure 5.2: Throughput vs. B_r for random accesses to both shared and global memory subsystems with same problem size ($n = 2^{25}$) but distinct working set sizes. (a) $m = 8$ KB. (b) $m = 32$ KB.

working set size of 8 KB as the hashing range. In Fig. 5.2(b), we use a working set size of 32 KB as the hashing range. In both experiments, B_r is varied continuously from 1 to 90.

Our first observation from Fig. 5.2(a) is that for both the shared memory subsystem and the global memory subsystem the empirically measured throughput is closely aligned with the model predictions. Over the range of requested blocks explored in the graph, the values $B_r = \{15, 30, 45, 60,\ 75, 90\} \subseteq B_{\text{opt}}$ give peak throughput (minimum execution time) consistent with the prediction of Eq. (5.3), and when B_r is not in B_{opt} the zigzag pattern predicted by Eq. (5.13) is observed in the empirical data.

Our second observation is that there is a fairly significant difference in throughput between the shared memory and the global memory. This is consistent with our expectation that the shared memory is better suited to the random access patterns that drive the performance of the micro-benchmark.

To explore the impact of working set size on performance, we repeat the experiments above varying m from the initial value of 8 KB to 32 KB. The results of these experiments are shown in Fig. 5.2(b) ($m = 32$ KB).

As the working set size gets larger, the throughput for the shared memory stays the same. This is consistent with the entire hash table fitting in shared memory for each working set size, so larger working sets do not provide any throughput disadvantages. In contrast to the shared memory result, the working set size has a dramatic impact on the performance of global memory accesses. While the zigzag pattern dependency on B_r is retained, the peak throughput is noticeably lower as the size of the working set increases. This is due to the smaller working set size being able to effectively exploit the on-chip caches that sit between the global memory and the multiprocessors on the GTX480.

To quantify the effect on performance of cache, the cache model f_{cache} proposed in Eq. (5.12) is examined via the micro-benchmark, fixing f_{sched} to be optimal by ensuring that $B_r \in B_{\text{opt}}$. Cache hit rate r_H and cache miss rate r_M are explored by varying the working set size m. Different L1 cache sizes are also explored by setting it to 16 KB and 48 KB. Both the measured and model-predicted rates are shown in Fig. 5.3. Dramatic increases in cache misses and decreases in cache hits are observed once a larger-than-cache working set size m is used. We see a nice correspondence between the modeled and measured results.

We next explore the cache model f_{cache}. Fig. 5.4 compares measured and predicted execution times for the micro-benchmark as the working set size is varied over the same range as in Fig. 5.3 (in this case, only for cache size 48 KB). For small working sets (that fit in cache) and for large working sets (that almost always miss cache) the execution time is flat. Eq. (5.12) does a reasonably good job of modeling the transition region between these two spaces.

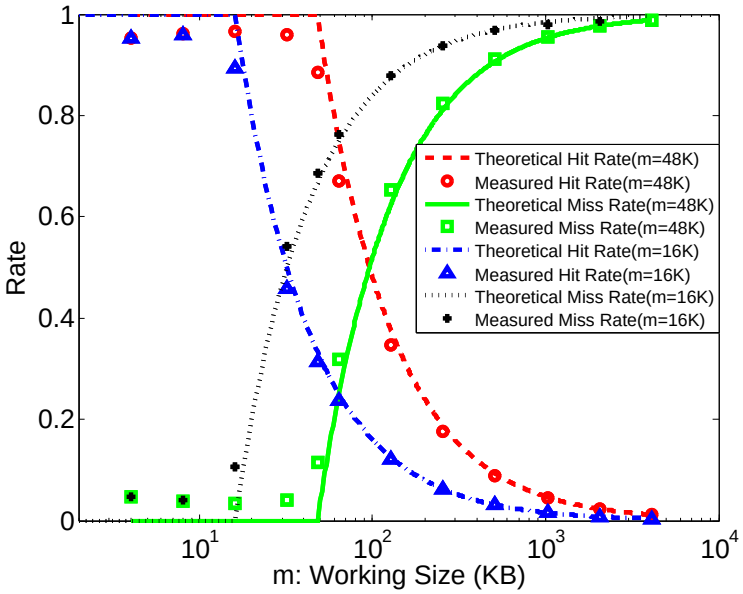


Figure 5.3: Cache hit and miss rates.

Given the close match between model predictions and empirical measurements for the micro-benchmark, we next consider a pair of real applications from the field of computational bioinformatics that are based either substantially or entirely on hashing.

5.2.2 Parallel Bloom Filters Algorithm Design and Implement

Bloom filters test set membership by performing multiple hashes on a candidate element and checking a bit-vector, called the *Bloom vector*, to see if the addresses resulting from these hashes are all set to "true." Fig. 5.5 illustrates this idea as applied to BLAST-style [26] string matching. Fixed-length candidate substrings of length w, or w-*mers*, from the database are fed into k independent hash functions, and the resulting addresses are checked against one or more Bloom vectors loaded with portions of the query sequence.

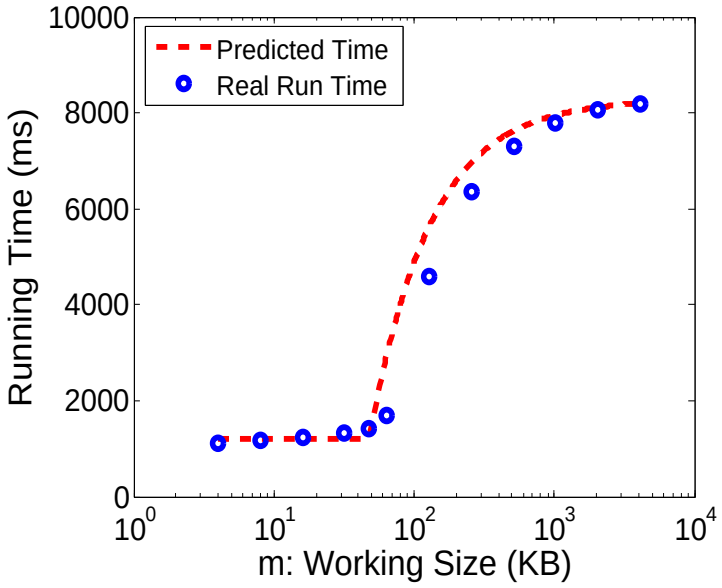


Figure 5.4: Impact of cache on execution time.

Algorithm 3 gives pseudocode describing the Bloom filter string-matching computation. With multiple candidate elements, multiple sets, and multiple hash functions, this algorithm provides many opportunities to exploit parallelism.

In our design, a long query of length Q_s is split into multiple *sub-queries* of a given length n_{sub}. Each sub-query is assigned an individual Bloom vector of size m bits, and each w-mer in the sub-query is considered to be an element of the set for that vector, such that the number of element to a single Bloom vector is $n_e = n_{sub} - w + 1 \approx n_{sub}$ [8]. Each w-mer in the database is simultaneously checked for set membership in each sub-query. This decomposition of a large set into multiple smaller sets (i.e., dividing a given query into a collection of sub-queries) is common practice with Bloom filters, as the false positive rate is a function of the number of elements in the set. A larger number of sub-queries, each with fewer elements, can lower the overall false positive rate.

[8]Each sub-query is contiguously chopped by a sliding window of size w

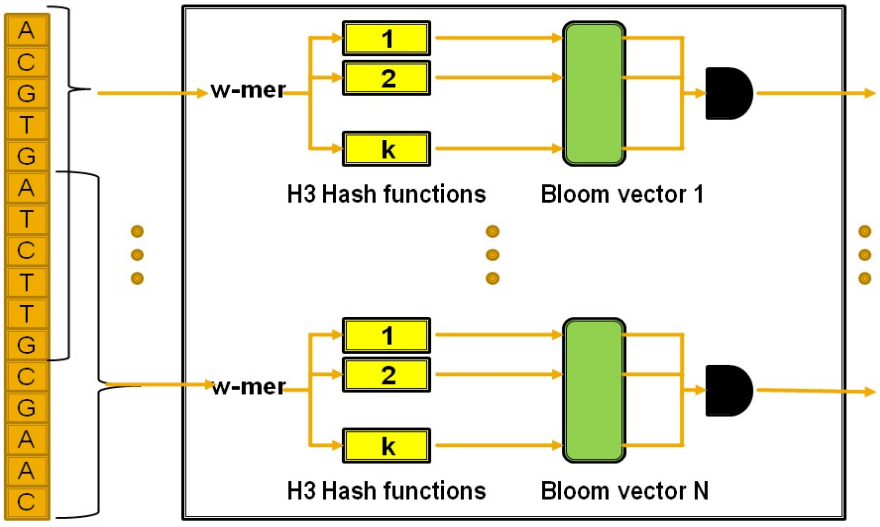


Figure 5.5: Parallel Bloom filters for detecting string matches of fixed length w between a query and a database.

Mercury BLAST [26], being FPGA-based, uses hash functions h_q from the H3 family [128], denoted as the set $\{h_q | q \in M_{Bool}\}$, where M_{Bool} represents the set of all possible $i \times j$ Boolean matrices. $q(k)$ is the bit string of the k^{th} row of a given matrix $q \in M_{Bool}$. Correspondingly, $x(k)$ is the k^{th} bit of x, the element that needs to be hashed. The hashing function $h_q(x)$ is therefore defined as

$$h_q(x) = x(1) \cdot q(1) \oplus x(2) \cdot q(2) \oplus \cdots \oplus x(i) \cdot q(i) \tag{5.17}$$

where $\cdot$ denotes the bit by bit AND operation and $\oplus$ the exclusive OR operation. These bit-level linear transformations are well-suited to hardware implementation. In this dissertation, we do not investigate the use of alternative hash functions, leaving this for future work.

Algorithm 3 Parallel Bloom Filters

```
1: Input: query sequence
2: Input: database sequence
3: Output: stream of database w-mers
4: {Initialize Bloom vectors}
5: for all sub-queries do
6:    initialize all-zero bitVector of size m bits
7:    for each w-mer in sub-query (denoted x) do
8:       for each hash function h do
9:          bitVector[hash_h(x)] = 1
10:      end for
11:   end for
12: end for
13: {Perform membership tests}
14: for all sub-queries do
15:   for each w-mer in database (denoted y) do
16:      for each hash function h do
17:         if bitVector[hash_h(y)] = 0 then
18:            discard this w-mer
19:            break
20:         end if
21:      end for
22:      if bitVector[hash_h(y)] = 1 for all h then
23:         output w-mer
24:      end if
25:   end for
26: end for
```

GPU Implementation

The GPU used for this study is the NVIDIA GTX 480, based on the Fermi architecture. It has 15 *streaming multiprocessors*, each of which has 32 *streaming processors* or processor cores (480 cores total) running at 1.4 GHz. The GTX 480 has about 1.5 GB of off-chip global memory, while each streaming multiprocessor has 48 KB of on-chip shared memory.

Kernel computations on the GPU are organized around thread blocks, which are independent from one another and are distributed across the multiprocessors for execution. Each block consists of a number of threads, which are distributed across the processor cores within a multiprocessor. Threads are scheduled in groups of 32, called *warps*. The shared memory is shared across threads but is partitioned across blocks. The registers within each core are not shared but rather are partitioned across threads.

Problem Decomposition

We implement only the membership tests from Algorithm 3 (lines 13 to 26) on the GPU. As sub-queries are independent, they are distributed across the blocks, each with their associated Bloom vector. The vectors reside in shared memory to ensure fast access. This decomposition of the problem, assuming B blocks and k hash functions, is illustrated in Fig. 5.6.

The database resides in the GPU's global memory. If, as is typically the case, the entire database is too large to fit in global memory, an additional outer loop (not shown) streams chunks of the database into global memory one at a time and serially executes the *for* loop on line 14 for each chunk.

Individual threads partition the w-mers in the database. Each thread executes the inner loop of lines 15 to 25. A thread fetches a w-mer from the database (located in global memory), computes the k hash functions for its assigned Bloom vector, checks the resulting k addresses in the vector, and finally returns any hits to global memory.

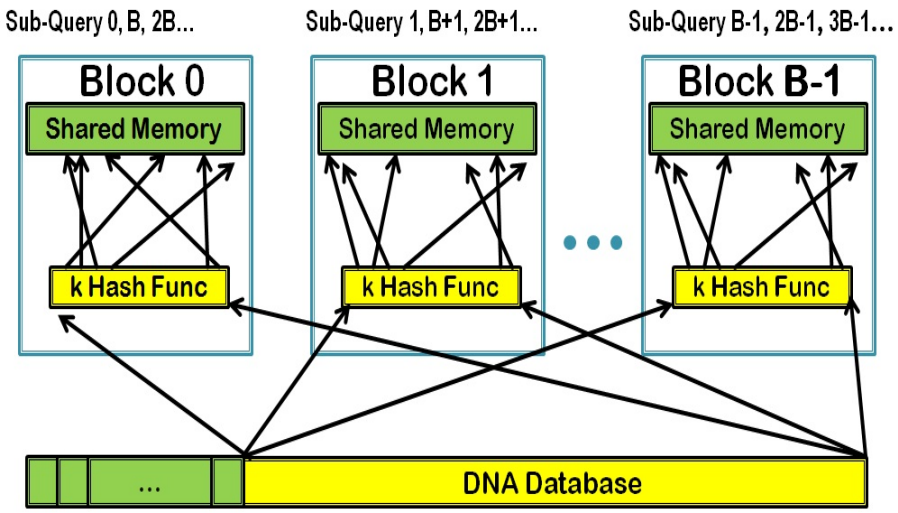


Figure 5.6: Implementation of parallel Bloom filter algorithm on GPU.

Operating Procedure

The overall sequence of operations consists of a number of steps, which are articulated below.

1. On the CPU:
   (a) The database is encoded using two bits per DNA base (character). Encoding need only be done once, after which the encoded database is stored in the file system.
   (b) The H3 hash functions for each Bloom vector are generated, and the Bloom vectors are initialized (lines 5 to 12).
   (c) The hash functions, Bloom vectors, and database are loaded into GPU memory.
2. On the GPU:
   (a) The membership tests are performed.
   (b) The database w-mers that hit in the Bloom filters are returned to CPU memory.

Optimization

We next describe various implementation decisions made in an attempt to optimize the performance of the kernel.

- *On-chip vs. off-chip memory allocation.* The off-chip global memory performs well when memory accesses within a warp can be coalesced. We accomplish this by having the 32 threads in a warp read 32 consecutive w-mers from the database. The on-chip shared memory more readily supports the random access pattern required by the Bloom vectors.

- *Thread-level parallelism (TLP) vs. Instruction-level parallelism (ILP).* There is a trade-off between the work assigned to a single thread and the total number of threads. This corresponds to balancing TLP (more threads, less to do in each thread) and ILP (fewer threads, more to do in each thread). In the Bloom filter implementation, more TLP would be exploited if we had assigned each of the k hash functions to a distinct thread. However, the additional synchronization overhead implied by a one-thread-per-hash design makes it more efficient to compute all hash functions for one w-mer in a single thread.

- *Unrolling loops.* As suggested in [133], we unrolled the loop (lines 16 to 24) that iterates through the k hash functions.

5.2.3 Bloom Filters in BLAST

BLAST is the most widely used tool for biosequence similarity search, which is a fundamental and crucial application for comparing and revealing the possibly biologically meaningful

relationships between a given query sequence and an annotated database [9]. Given the rapid rate at which new genomic sequence data is being produced, BLAST searches have become progressively more and more expensive. In Buhler et al. [26], a Bloom filter [21], a probabilistic hashing algorithm and data structure for performing set membership tests with a manageable risk of producing false positives, is introduced at the front end of the traditional BLAST pipeline to discard a large fraction of the database prior to explicit table look-up and match verification.

A parallel Bloom filter algorithm for BLAST using a graphics engine is described in Section 5.2.2. The algorithm deposits portions of the database in global memory, divides long queries into a set of sub-queries, and maps each sub-query to a specified Bloom-vector in shared memory for each kernel block. Multiple passes over the database are needed as the number of sub-queries are larger than the blocks the device can maximally support. Each thread reads a string of DNA characters (called a w-mer, since it is w characters in length) from the database in global memory, sequentially executes several hash functions in the kernel, and interrogates the values in shared memory pointed to by the hash results.

It is common for the performance of an application to be multidimensional. In the case of Bloom filters, we have two primary performance indicators of interest: throughput and sensitivity. Throughput can be quantified for our BLAST application as the number of database w-mers processed per unit time, while sensitivity is quantified as the false positive rate realized during set membership tests.

Sensitivity

The sensitivity of a Bloom filter is quantified by the false positive rate, FPR, or fraction of set membership tests that return true when the element tested is not a member of the set. Lower false positive rates reflect better Bloom filter sensitivity.

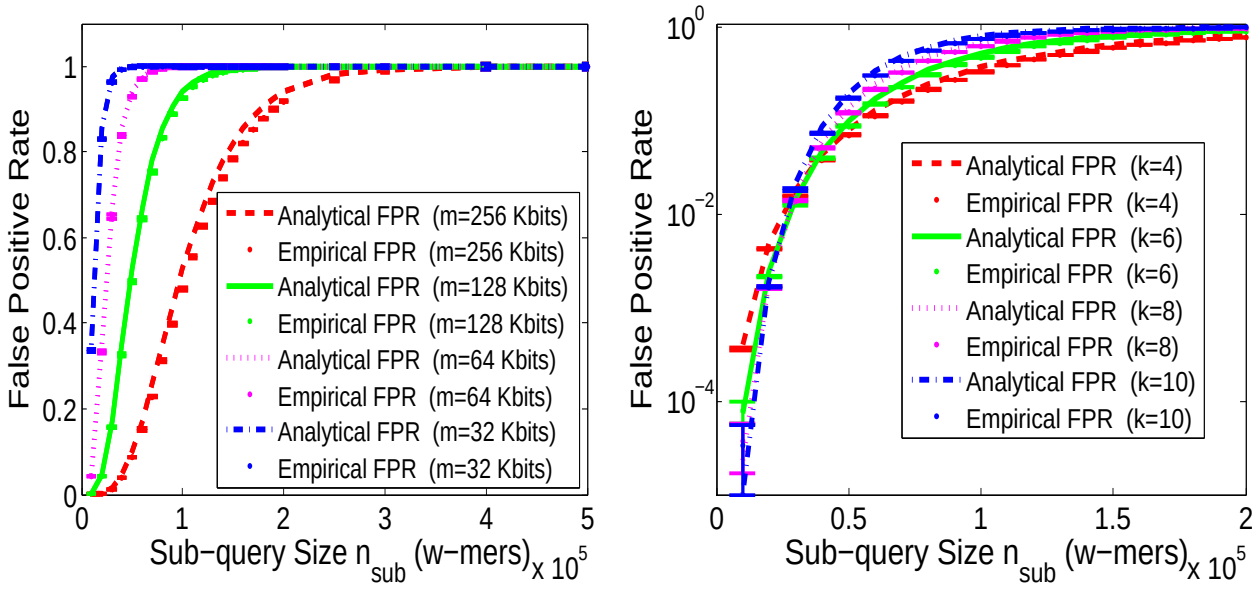
Assuming element independence and good uniformity in the hash functions, the false positive rate for a Bloom filter is well modeled [25]. FPR is a function of the Bloom vector size m, the number k of hash functions, and the number n_e of elements hashed into the vector:

$$FPR = \left(1 - \left[1 - \frac{1}{m}\right]^{kn_e}\right)^k . \tag{5.18}$$

According to the analytic model, FPR increases with n_e and decreases when m is increased. Increases in k can cause FPR to move in either direction, depending upon the value of the other two parameters.

In our usage of the Bloom filter within BLAST, both k and m are design parameters under direct control of the developer, while n_e is indirectly set by how the user decomposes the complete query into sub-queries.

To investigate whether biosequence data sets are sufficiently well-behaved so as to fit the theoretical expression for FPR above, we tested our implementation using real DNA sequences. Human chromosome 1 (250 Mbases) was used as our query sequence, while human chromosome 22 (50 Mbases) was used as the database. During execution of our GPU kernel, we counted false positives FP, false negatives FN, and true positives TP. The empirical FPR



(a) Fixed k with varied m (b) Fixed m with varied k

Figure 5.7: Theoretical and empirical results of FPR with varied sub-query size n_{sub}. (a) shows FPR for several values of m with a fixed $k = 6$. (b) shows FPR for several values of k with a fixed $m = 256$ $Kbits$.

for a database of size n is given by

$$FPR = \frac{FP}{n - TP}. \tag{5.19}$$

We confirmed empirically that $FN = 0$, as required by any correct implementation. Fig. 5.7(a) and Fig. 5.7(b) compare the theoretical and empirical FPR for a range of values of k, m, and n_{sub}. In both figures, lines indicate the theoretical FPR, while mean measured FPR values over all sub-queries are shown as points with associated 95% confidence intervals. Fig. 5.7(a) varies n_{sub} for several values of m with a fixed $k = 6$, while Fig. 5.7(b) varies n_{sub} for several values of k for a fixed $m = 256$ Kbits. As expected, FPR grows with increasing n_{sub} (effectively n_e) in all cases. For a given n_{sub}, larger m leads to a smaller FPR and larger k can influence FPR either direction (depending on the value of n_{sub}).

While the theoretical and empirical results are highly similar, the theoretical quantities frequently lie outside the confidence intervals of the empirical measurements. This is due to the fact that DNA bases are, generally, not independent of one another, but are in fact correlated. We explore the magnitude of this discrepancy by plotting a histogram of the relative error in Fig. 5.8. While there are individual measurements with relative error greater than 10%, they are few, and the bulk of the errors are near zero.

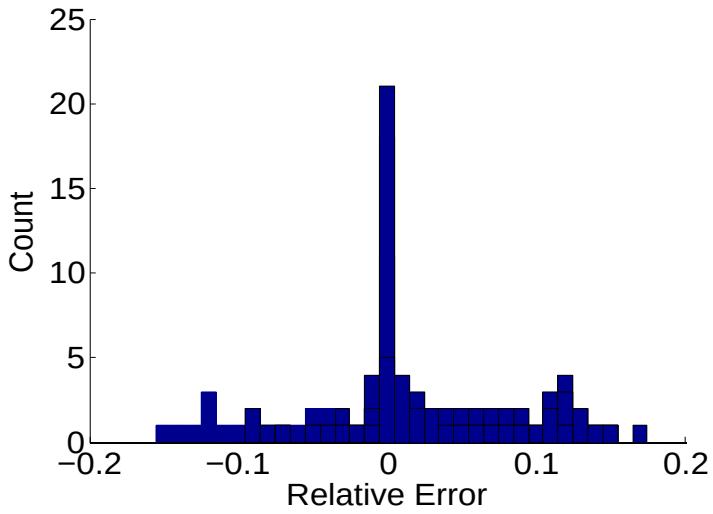


Figure 5.8: Histogram of relative error between theoretical predictions and empirical measurements for FPR.

Throughput

We focus on the throughput of the kernel execution rather than the overhead of data movement into and out of the GPU. Fig. 5.9 stacks measured data movement times below kernel execution time for a range of values of the sub-query size n_{sub}. The data movement times include loading the initial Bloom vector contents, loading the database, and returning the results. As can can be seen in the graph, the kernel dominates the overall time, and I/O is not a bottleneck for this application.

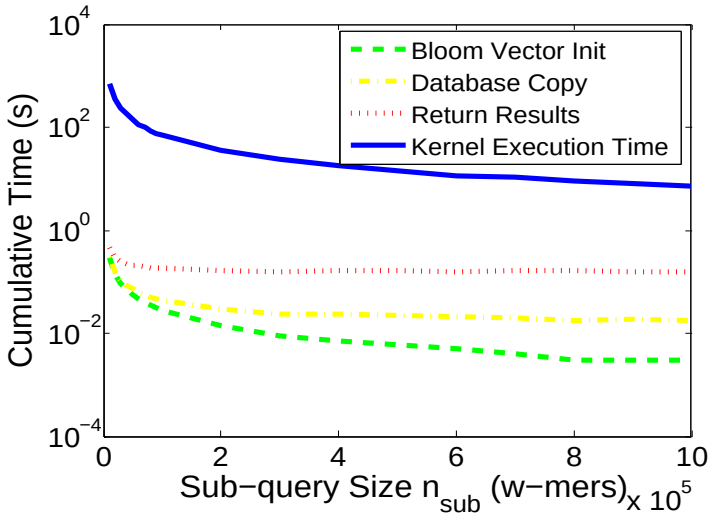


Figure 5.9: Cumulative execution time for data movement and kernel.

As there is significant interaction among the various application-specific and architecture-specific parameters that influence throughput, we will construct the model piece by piece until all relevant parameters have been included. The first two parameters to be investigated, the number of hash functions k and the sub-query size n_{sub}, influence both sensitivity and throughput and so partly control the tradeoff between them.

- Number of Hash Functions

  Kernel execution time should be linearly proportional to the number k of hash functions used. This is because each thread computes the k hash functions sequentially for each database w-mer assigned to it. Fig. 5.10 shows empirical execution times vs. k for a variety of sub-query sizes n_{sub}, along with a linear trend line fit to the data for each n_{sub}. The tight fit of the trend lines to the data confirms our expectation that $Time \propto k$. Increasing k to improve sensitivity therefore negatively impacts throughput.

- Sub-query Size

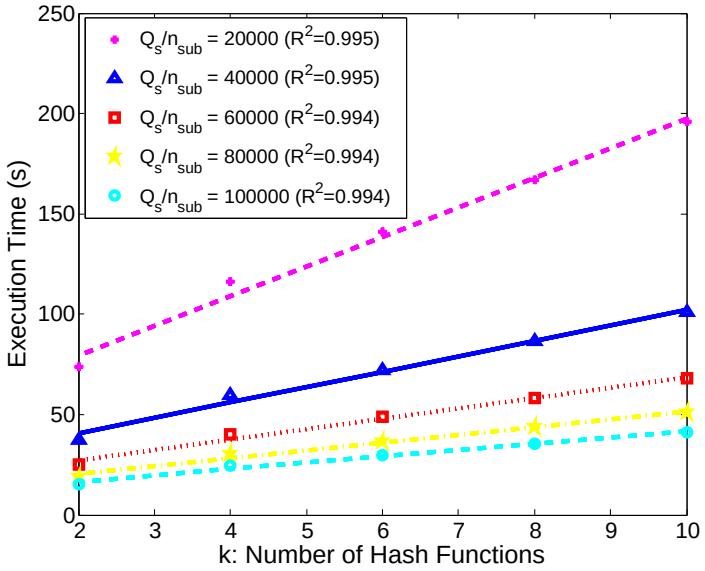


Figure 5.10: Execution time for different number of hash functions.

With a query of size Q_s that is divided into sub-queries of size n_{sub}, the number of sub-queries that must be processed is Q_s/n_{sub}. Because each sub-query is assigned to a block, and there are more sub-queries than blocks, multiple passes over the currently resident portion of the database are needed to process all sub-queries. Execution time is therefore proportional to total query size and inversely proportional to sub-query size. Fig. 5.11 tests the above relation in four sets of experiments with different numbers k of hash functions. As above, the points represent empirical execution, while lines are fitted to the empirical data. The high goodness of fit confirms that $Time \propto Q_s/n_{sub}$. Increasing n_{sub} therefore improves throughput but negatively impacts sensitivity.

Then, it is straightforward to develop an expression for f_{app} that reflects the Bloom filter implementation described above. The algorithmic parameters include the number of hashing functions k and sub-query size n_{sub}, both are included in $\overrightarrow{\text{algo}}$. In terms of input problem size,

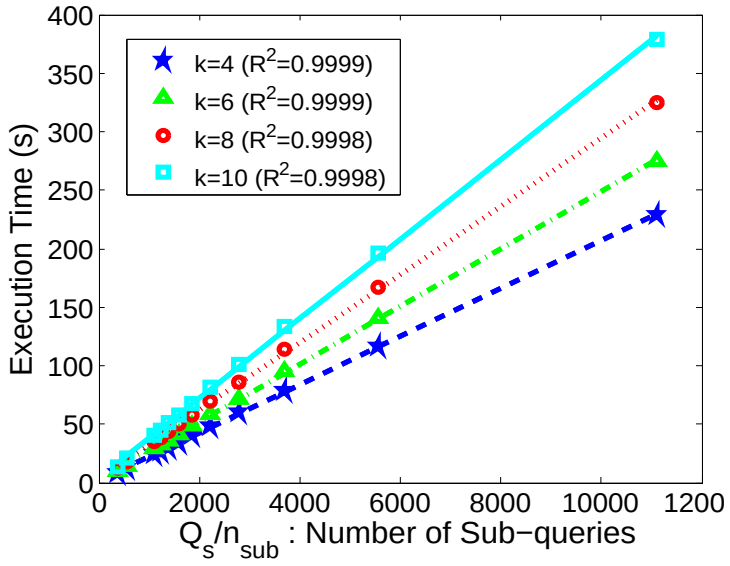


Figure 5.11: Execution time for different sub-query sizes.

we have the database sequence size n and the query sequence size Q_s. This results in

$$\overrightarrow{\text{algo}} = (k, n_{sub}),$$

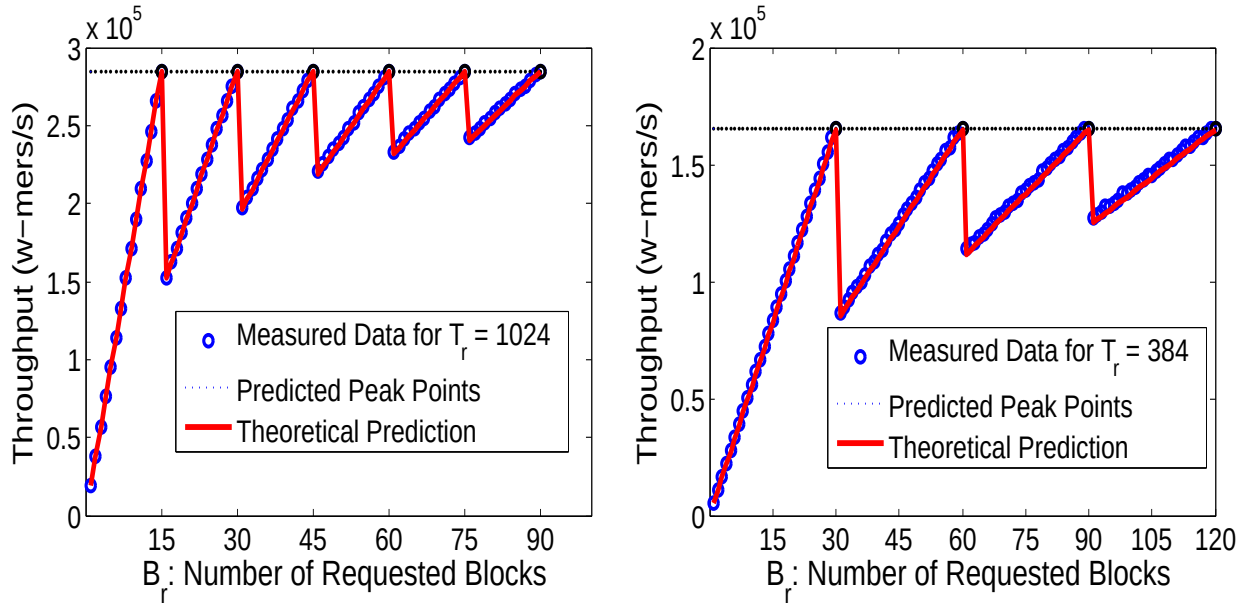
$$\overrightarrow{\text{inpt}} = (n, Q_s),$$

and

$$f_{\text{app}}(\overrightarrow{\text{algo}}, \overrightarrow{\text{inpt}}) = k \times \frac{Q_s}{n_{sub}} \times n. \tag{5.20}$$

As the Bloom-vector is entirely held in shared memory, A_C is true. Substituting Eq. (5.20) into Eq. (5.9) yields an expression for the performance of BLAST's Bloom filter executing on a graphics engine.

$$Time \propto k \times \frac{Q_s}{n_{sub}} \times n \times \frac{\lceil \frac{B_r}{B_a \times P/Q} \rceil \times B_a \times P/Q}{B_r} \quad \text{if } A_T. \tag{5.21}$$



(a) GTX 480 (15 Multiprocessors) (b) Tesla C1060 (30 Multiprocessors)

Figure 5.12: Throughput vs. B_r on two GPU machines for Bloom filter of BLASTN. (a) is for prediction and empirical measurements on GTX 480. Peak performance is hit every 15 blocks as GTX 480 has 15 multiprocessors. (b) is for prediction and empirical measurements on Tesla C1060. Peak performance is hit every 30 blocks as Tesla C1060 has 30 multiprocessors.

The Bloom filter was executed while searching human chromosome 1 (250 MBases) against human chromosome 22 (50 MBases). Choosing $T_r = 1024 \in T_{\text{opt}}$, the predictive power of Eq. (5.21) is explored on two different GPU machines in Fig. 5.12. As can be seen in the figure, there is an excellent correspondence between the empirical measurements and the model predictions. The zigzag pattern of Fig. 5.2(a) and 5.2(b) is again present, both in the empirical data and in the model's predictions.

Overall Model

The overall performance model is therefore:

$$FPR = \left(1 - \left[1 - \frac{1}{m}\right]^{kn_e}\right)^k \tag{5.22}$$

and

$$Time \propto \frac{k}{n_{sub}} \cdot Q_s \cdot n \cdot f_{sched} \quad \text{if } A_T. \tag{5.23}$$

or

$$Time = a_1 \cdot \frac{k}{n_{sub}} \cdot Q_s \cdot n \cdot \frac{\lceil \frac{B_r}{B_a \times P/Q} \rceil \times B_a \times P/Q}{B_r} + a_0 \quad \text{if } A_T. \tag{5.24}$$

for constant coefficients a_1 and a_0. Also,

$$Tput = n/Time.$$

Fig. 5.13 establishes the constant coefficients for the GTX 480 by fitting a linear curve to measured data. The range of parameters for this plot include: $k \in \{4, 6, 8, 10\}$, $10000 \leq n_{sub} \leq 300000$, $m \in \{64, 128, 256\}$ Kbits, $B_r \in B_{\text{opt}}$, $T_r \in T_{\text{opt}}$. The throughput predicted by the model, and confirmed via experiment, represents a speedup of approximately 35-fold over a 2.6 GHz, quad-core, AMD Opteron system executing the same algorithm (300,000 w-mers/s vs. 8,500 w-mers/s with $n_{sub} = 50000$ and $k = 6$). On the Opteron, the code was compiled using `gcc` version 4.1.2 at optimization level `-O2` using OpenMP to express thread-level parallelism, but no attempt was made to exploit the SIMD instruction set.

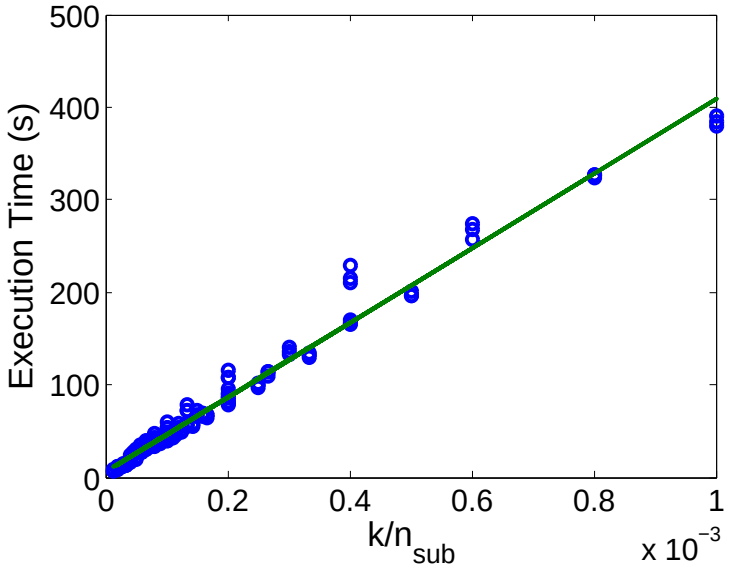


Figure 5.13: Modeled vs. measured execution time on GTX 480 ($a_1 = 4.01 \times 10^5$, $a_0 = 10$, $R^2 = 0.9909$).

5.2.4 Model Use to Evaluate Performance Tradeoffs

It is common practice when using Bloom filters to trade execution speed (throughput) for improved sensitivity (false positive rate) by partitioning the original set into subsets and performing set membership tests on each of the subsets. This is precisely what we are doing in BLAST when the original query sequence is decomposed into sub-queries. Fig. 5.14 illustrates the quantified (via the model) tradeoff between execution time and false positive rate for $k = 6$ hash functions, $m = 256$ Kbits, $B_r \in B_{\text{opt}}$ blocks, and $T_r \in T_{\text{opt}}$ threads per block.

In this way, the user is capable of quantitatively assessing the tradeoff between throughput and sensitivity as controlled by the sub-query size, n_{sub}.

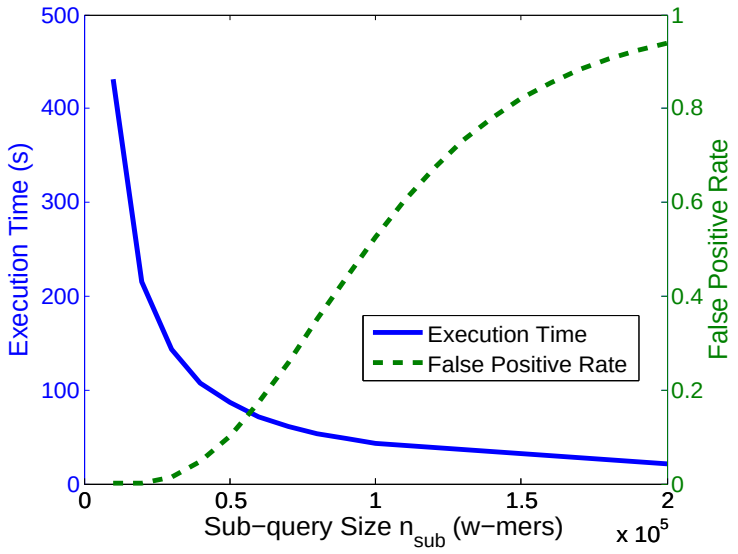


Figure 5.14: Tradeoff between false positive rate and execution time.

5.2.5 DNA Classification

Another application that exploits hashing is DNA classification with Bloom filters. As DNA sequencing technologies provide ever more data to be analyzed, frequently biologists are interested in identifying only the novel sequence in a given data set. Stranneheim et al. [144] describe an algorithm, called FACS, which uses Bloom filters to classify sequences as belonging to one of many reference sequences vs. being novel. Their perl-based implementation is evaluated using synthetic meta-genomic data sets and compared to conventional methods such as BLAT and SSAHA2. Stranneheim et al. observed a 21-fold speedup when FACS was executed on a 2.8 GHz Intel Xeon processor.

We ported FACS to the NVIDIA graphics engine to explore the potential for even greater performance gains. There are numerous opportunities for parallel execution, making it potentially well suited for the graphics engine; however, its reliance on hashing as a basic

operation poses some question as to its ultimate suitability. Let us first assess the opportunities for parallelism. Within each short query sequence (typically less than 390 characters long), hashing the w-mers (substrings of length w) are independent. In our implementation, each w-mer within a query is assigned to a thread, which is responsible for computing all of the k hashes to implement the Bloom filter.

Second, the queries themselves (totaling approximately 10^5 sequences) are also independent and can be analyzed in parallel. We assign queries to thread blocks. Further, multiple kernel invocations are used to process groups of queries, and CUDA streams are used to provide overlapping kernel execution and memory copy to/from the graphics engine. Fig. 5.15 illustrates the organization of the FACS implementation on the graphics engine. Here, Q_i is the i^{th} query sequence. In the empirical investigation that follows, Bloom filters were created

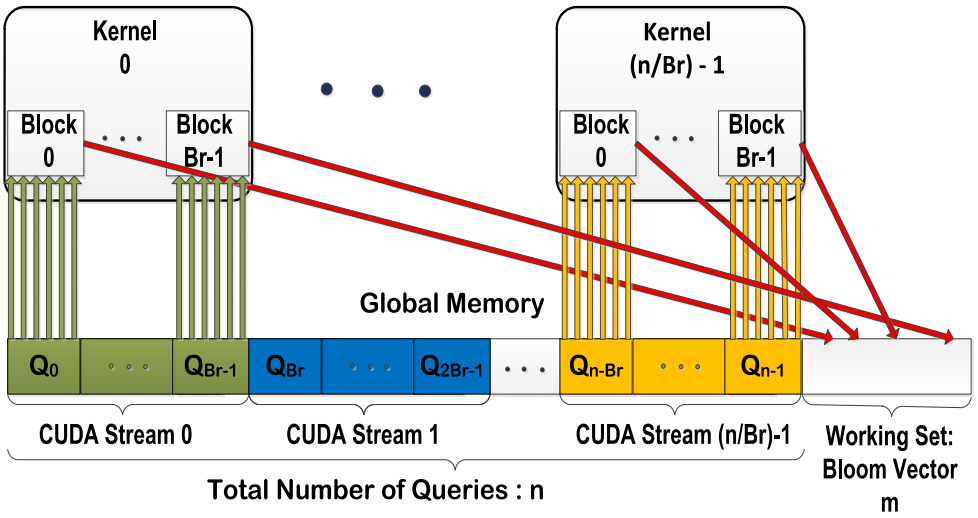


Figure 5.15: Implementation of FACS DNA classification application.

based on reference sequences from [144] with a measured false positive rate of 0.014% (lower than that in [144]), with one Bloom-vector per reference sequence. Due to the much larger size of the Bloom-vectors than on-chip shared memory, they were allocated to global memory.

The performance of this implementation can also be predicted by our model. The number of hashing functions k and the number of CUDA streams (which is inversely proportional to B_r) are the key parameters to be included in $\overrightarrow{\text{algo}}$. In terms of problem size, n is the number of sequences to be classified. This results in

$$\overrightarrow{\text{algo}} = (k, B_r) \tag{5.25}$$

and

$$\overrightarrow{\text{inpt}} = (n). \tag{5.26}$$

Since B_r might be larger than the number of active blocks that the multiprocessors can support, i.e., $B_a \times P/Q$, multiple passes are needed. So we have

$$f_{\text{app}}(\overrightarrow{\text{algo}}, \overrightarrow{\text{inpt}}) = k \times \frac{n}{B_r} \times \frac{B_r}{B_a \times P/Q} = k \times \frac{n}{B_a \times P/Q}. \tag{5.27}$$

A_C is false for this application due to the much larger Bloom-vector as working set than caches. So f_{cache} is greater than one, and substituting Eq. (5.27) and Eq. (5.12) into Eq. (6.3), we obtain the runtime expression for DNA classification as

$$\begin{aligned} Time \;\; \propto \;\; & k \times \frac{n}{B_a \times P/Q} \times \\ & \left(\min\left(1, \frac{C_s}{m}\right) + \left(1 - \min\left(1, \frac{C_s}{m}\right)\right) \times G\right) \times \\ & \frac{\lceil \frac{B_r}{B_a \times P/Q} \rceil \times B_a \times P/Q}{B_r} \quad \text{if } A_T \text{ .} \end{aligned} \tag{5.28}$$

The model is experimentally assessed on the GPU with the same synthetic meta-genome data set as [144], including 10^5 short query sequences. In our implementation, sequences are evenly distributed across a set of streaming kernels, and B_r blocks are requested on each

kernel. This division restricts the number of sequences to be processed per kernel, therefore value options for B_r are limited, thereby preventing B_r from being a multiple of $B_a \times P/Q$.

The predictions of the model are presented in Fig. 5.16. Because of the implementation restrictions described above, there are fewer empirical values for B_r relative to the previous application. Nonetheless, we see an excellent alignment between the model's predictions and the measured throughput achieved by our implementation.

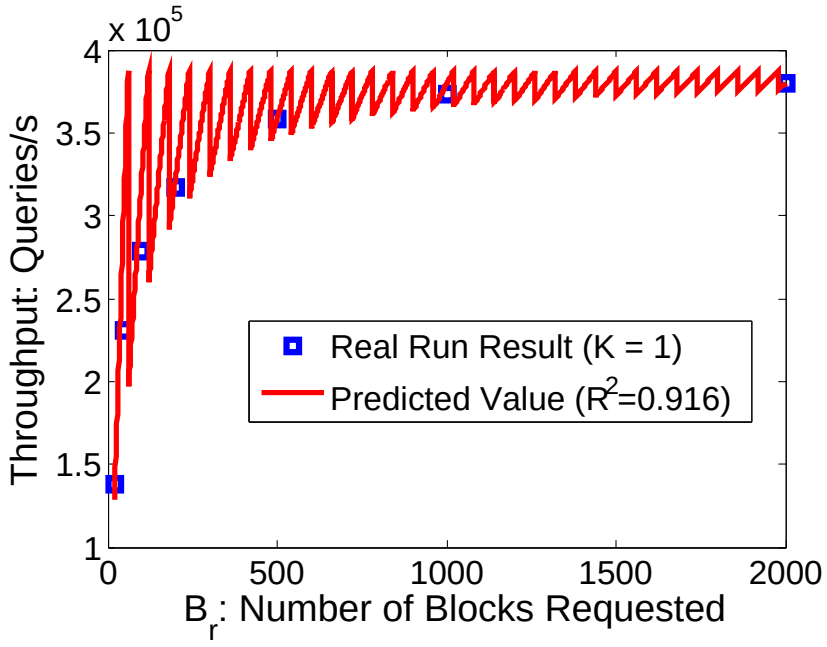


Figure 5.16: FACS throughput for different numbers of requested blocks.

For this problem, the working set size m (512 KB) is much larger than the cache, independent of the choice of cache size (16 KB or 48 KB). As a result, we do not predict a change in performance when the cache size is changed, and this was confirmed experimentally as well.

Quantifying the throughput in terms of hashes processed per second, our implementation executing on the GTX480 is 20 times faster than the perl version implementation executing on an Intel Core 2 Duo CPU running at 2 GHz.

Chapter 6

Integrated Analytical Framework

In this chapter, we extend previously proposed analytical models in Chapter 3 and 5, jointly addressing the parallelism exploited by the algorithm, the effectiveness of latency-hiding, and the utilization of multiprocessors (occupancy). In particular, the integrated model not only helps to explore and reduce the configuration space for tuning kernel execution on GPUs, but also reflects performance bottlenecks and predicts how the runtime will trend as the problem and other parameters scale. The model is validated with empirical experiments. In addition, the model points to at least one circumstance in which the occupancy decisions automatically made by the scheduler are clearly sub-optimal in terms of runtime.

6.1 Bridge the Asymptotic Model and the Calibrated Model

Performance of algorithms on GPUs largely depends on the suitability of the underlying algorithm, the effect of memory subsystem on performance, and the efficiency of scheduling on many-core architectures. An algorithm is well suited for GPUs only when it has sufficient

parallelism and is not unduly bounded by memory latencies. A program runs efficiently only when it launches a large number of threads while not incurring too much memory traffic. A scheduling scheme works perfectly only when it manages and distributes the resources among the thread hierarchy in a well balanced manner so that all the streaming multiprocessors run with full occupancy. These factors all jointly impact the performance. Ultimately, the achieved performance for an algorithm is a complicated interaction between a variety of parameters, some determined by the algorithm, some set by developers, and others imposed by the architecture of the particular machine being used. The interactions between these parameters, as well as the impact of each on the algorithm's performance, are often not well understood.

We are interested in improving the understanding of the performance of algorithms on many-core GPUs through the use of analytic performance models. Chapter 3 and 5 of this dissertation are examples of developing such models. As a general rule, these models fall in two categories: (1) asymptotic models for algorithm analysis at a high level of abstraction that attempt to capture only the essential features of GPU architectures; and (2) calibrated performance models that attempt to make specific, quantitative predictions about application runtime, including many lower level details that would be considered unimportant in an asymptotic analysis.

In this chapter, we utilize both asymptotic analysis and calibrated performance prediction on many-core GPUs, in effect drawing on the concepts of both Chapter 3 and5. We develop an integrated analytical framework combining both, analyzing algorithm efficiency and predicting the achievable execution time based on a quantification of ***parallelism***, ***latency-hiding***, and ***occupancy***. Within the context of the asymptotic analysis, in addition to the

computational complexity expressed in terms of ***work*** T_1 — the total amount of computation, or, in other words, its running time on 1 processor — and ***span*** T_∞ — the amount of computation on the critical path or, in other words, the running time on an infinite number of processors, we also consider the memory complexity determined by the number of memory transfers M from slow memory to fast memory as a critical performance measure. In general, GPUs attempt to mask memory transfers to/from slow memory by executing a substantial number of concurrent threads, whereby nominally there is always a set of threads ready for execution (i.e., not waiting on a memory reference to complete). However, one cannot launch arbitrarily large numbers of threads due to limited on-chip resources, which are managed by the scheduler. At the same time, simply seeking a large thread count per thread block may not always provide high occupancy on streaming multiprocessors and therefore cannot guarantee good performance. As a result, we model GPU scheduling mechanisms and use them as a factor bridging the gap between the asymptotic model and calibrated model, between the theoretical performance and real runtime.

Our model is useful in a number of aspects:

1. It is able to identify the performance bottleneck of a particular algorithm, and judge whether the algorithm is more likely to be performance bound by memory accesses or by computation.

2. It predicts performance trends as the problem size (or other parameters) scale up. This can be quite helpful when comparing and ordering different algorithms with various parameter settings.

3. It can explore and reduce the design and configuration space for tuning kernel execution on GPUs. A kernel execution launches a ***grid*** of thread blocks, each of which consists

of a number of threads. Problems can be decomposed and processed on this two-level thread hierarchy as shown in Fig. 2.2 by specifying the grid size (number of thread blocks per kernel) and block size (number of threads per block). Choosing how to decompose the problem into subproblems and picking the right thread block/grid size for better scheduling constitutes a gigantic search space. The model is able to project the possible runtime given different inputs and prune the space accordingly.

4. It is helpful for identifying performance improvement opportunities along two dimensions, scheduling and algorithm design. Oftentimes, general algorithms may suffer from insufficient parallelism or bad access patterns, or both to a different extent. Sub-optimal configuration of kernel launch can also impede the performance. Guided by the model, algorithms can be designed to maximize the parallelism while at the same time minimizing frequent and irregular long-latency memory accesses. The scheduling scheme should balance among the choices of sub-problem size, thread block/grid size, and also the workload and resources consumed per thread. On one hand, given a fixed resource consumption per thread, increasing threads in a block will increase on-chip resource request (registers/shared memory) for the entire thread block. Given the finite on-chip resources, the active blocks that can be launched simultaneously on-chip may reduce as previously introduced in Section 5.1.1. In this case, the occupancy defined in Eq. (5.1) may drop as the total number of active blocks and threads is reduced. However, on the other hand, enlarging the sub-problem size and, for each, assigning a bigger thread block can reduce the passes that are needed to solve the entire problem. This trade-off relation is quantified in our model.

5. It highlights the sub-optimality of existing GPU scheduling schemes in some scenarios. Chasing high occupancy, the current GPU scheduler dispatches thread blocks onto a

certain number of streaming multiprocessors in a greedy way depending on the resource usage of each thread block. However, a good occupancy does not necessarily guarantee the best runtime. We illustrate a set of use cases where artificially increasing the requested amount of shared memory results in substantial performance gains.

6.1.1 Combining the Two Models

As the first step in combining the above two models, we observe that the algorithm complexity, f_{app}, of the calibrated model directly corresponds to the asymptotic runtime, T_P. In addition, the TMM model for T_P extends f_{app} in two important ways:

1. The TMM model explicitly includes the impact of memory references on execution time. In particular, the memory complexity explicitly reflects the cost of memory behavior and, at the same time, the caching effect in terms of C — how many memory operations or data accesses can be grouped and Z — how much data can be cached and shared. This implies that T_P takes over not only the functionality of f_{app} in the model of Chapter 5 but also the functionality of f_{cache}.

2. The model of Chapter 5 is constrained to the circumstance where there are sufficient threads to mask memory latency. The TMM model has no such constraint, and therefore the condition variable A_T in Eq. (5.9) can be eliminated.

The second step in combining the two models is to substitute the appropriate[9] portions of Eq. (3.2) into Eq. (5.9).

$$\begin{aligned} Time &\propto T_P \times f_{\text{sched}} && (6.1)\\ &\propto \max\left(\frac{T_1}{P}, \frac{ML}{\mathcal{T}P}\right) \cdot \frac{\lceil \frac{B_r}{B_a P/Q} \rceil \cdot B_a P/Q}{B_r} && (6.2)\\ &\propto \max\left(T_1, \frac{ML}{\mathcal{T}}\right) \cdot \left\lceil \frac{QB_r}{B_a P} \right\rceil \cdot \frac{B_a}{QB_r} && (6.3) \end{aligned}$$

In this combined model, the asymptotic dependence on computational work is reflected by T_1 and on memory accesses is reflected by $ML/\mathcal{T}$. Both T_1 and M are algorithm-specific, and they will be expanded in the following section. Performance improvements due to increasing P are reflected in the second term, in which the ceiling function reflects the impact of processor occupancy.

6.2 Application of the Integrated Analytical Framework

In this section, we pick a classic algorithm — dynamic programming via adjacency matrix — for solving the all-pairs shortest paths problem, as a vehicle for empirically investigating the extended performance model. We develop expressions for the work, T_1, and the number of memory transactions, M, as a function of the problem size. These are then substituted into Eq. (6.3).

[9]Note that the algorithms we consider have sufficient parallelism so that the runtime is never limited by the span; that is $T_\infty \ll T_1/P$ for reasonable problem sizes. Thus, we will drop the span term in the extended model.

Given a graph $G = (V, E)$ with n vertices and m weighted edges, an ***all-pairs shortest paths*** algorithm calculates the shortest weighted path from every vertex to every other vertex. Here we consider the dynamic programming algorithm [42] that uses repeated matrix multiplication. The graph is represented as an adjacency matrix A where A_{ij} represents the weight of edge (i, j). A^l is a transitive matrix where A^l_{ij} represents the shortest path from vertex i to vertex j using at most l intermediate edges. $A^1 = A$ and A^2 can be calculated from A^1 using squaring (similar to matrix multiplication):

$$A^2_{ij} = \min_{0 \le k < n} (A^1_{ij}, A^1_{ik} + A^1_{kj}). \tag{6.4}$$

In order to calculate all pairs shortest paths, we simply calculate A^{n-1} using repeated squaring.

This algorithm was analyzed in Chapter 4. The work $T_1 = n^3 \lg n$ is the same as with traditional PRAM analysis. The memory cost M is

$$M = \frac{n^3 \lg n}{S_D C}. \tag{6.5}$$

Substituting the above expressions for T_1 and M into (6.3) yields the following:

$$Time \propto \max \left(n^3 \lg n, \frac{n^3 \lg n \cdot L}{S_D C \mathcal{T}} \right) \cdot \left\lceil \frac{QB_r}{B_a P} \right\rceil \cdot \frac{B_a}{QB_r}. \tag{6.6}$$

There is an intrinsic relation between the sub-block[10] dimension, S_D, and the number of requested thread blocks, B_r, according to how the problem is partitioned and assigned by

[10]We use the term 'sub-blocks' to refer to the partitioned data set, or working set, differentiating from 'block' which we use exclusively for thread block.

the algorithm. There are $(n/S_D)^2$ total sub-blocks, each assigned to a single thread block, $B_r = (n/S_D)^2$, i.e. $S_D = n/\sqrt{B_r}$.

When configuring a kernel, the number of threads per thread block and the number of requested thread blocks are the two direct variables that can be changed. Also, varying the sub-block dimension effectively changes the total number of requested thread blocks. As a result, we can unify two of the parameters in the expression above by substituting for S_D with $n/\sqrt{B_r}$, which yields:

$$Time \propto \max\left(n^3 \lg n, \frac{n^2 \lg n \cdot L\sqrt{B_r}}{C\mathcal{T}}\right) \cdot \left\lceil \frac{QB_r}{B_a P} \right\rceil \cdot \frac{B_a}{QB_r}. \tag{6.7}$$

The expression above is informative in a number of aspects.

1. When $\mathcal{T} > L/(S_D C)$ (i.e., $n > L\sqrt{B_r}/(C\mathcal{T})$), the latency to access memory is effectively hidden. The first term in the max dominates, indicating that the performance of the algorithm is bounded by computation:

$$Time \propto n^3 \lg n \cdot \left\lceil \frac{QB_r}{B_a P} \right\rceil \cdot \frac{B_a}{QB_r}. \tag{6.8}$$

   In this case, changing the kernel runtime configuration varying the number of threads, $\mathcal{T}$, will not have any impact on runtime.

2. When $\mathcal{T} < L/(S_D C)$ (i.e. $n < L\sqrt{B_r}/C\mathcal{T}$), the latency is not well hidden, due to either an insufficient number of threads or memory latency, L, being too large. Now, the second term in the max is larger, denoting that the runtime of the algorithm is

dominated by memory behavior:

$$Time \propto \frac{n^2 \lg n \cdot L\sqrt{B_r}}{C\mathcal{T}} \cdot \left\lceil \frac{QB_r}{B_a P} \right\rceil \cdot \frac{B_a}{QB_r}. \tag{6.9}$$

In this case, the runtime is predicted to be linear with $\sqrt{B_r}/\mathcal{T}$. This means that when performance is bounded by memory latency, enlarging the sub-block size, S_D, (i.e., effectively reducing the requested number of thread blocks, B_r) or increasing the average thread count per core, $\mathcal{T}$, can reduce the runtime.[11]

3. Considering the impact of the second two terms of Eq. (6.3), note that the number of active blocks, B_a, is determined by the scheduler according to register usage, shared memory usage, and fixed device capability (recall Eq. (5.2) in Chapter 5). Performance is maximized when the requested number of blocks, B_r, is an integer multiple of the product of B_a and P/Q, thereby balancing the number of blocks allocated to each multiprocessor and maintaining a high occupancy. According to Eq. (5.13), continuously varying B_r generates runtimes with a zigzag pattern as illustrated in Fig. 6.1. As can be readily observed, the runtime is minimized at regular intervals when B_r happens to be a multiple of 15. As B_r grows to larger values, the influence of B_r on occupancy diminishes.

6.3 Empirical Validation

In this section, we validate the predictions drawn from the analysis above through empirical measurements. We use an NVIDIA GTX480 with 15 multiprocessors, each of which has

[11] Note that increasing average threads per core does not necessarily mean increasing the threads per block, as sometimes reducing the threads per block enables the multiprocessor to schedule more thread blocks.

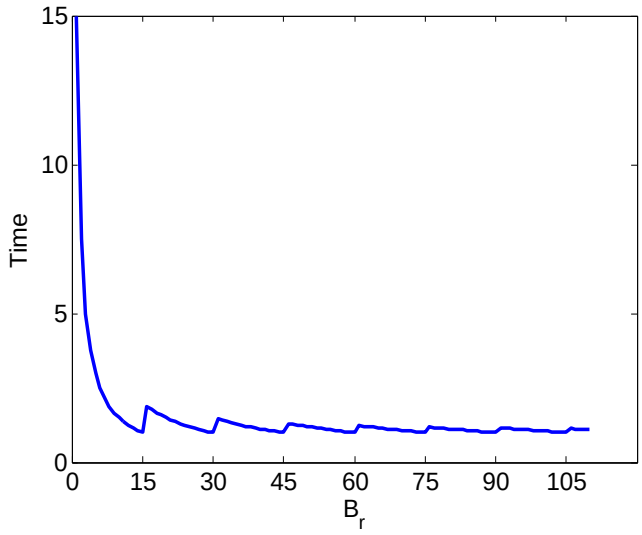


Figure 6.1: Execution time variation with requested number of blocks, B_r. For this example, $B_a = 1$, $P/Q = 15$ (e.g., as in an NVIDIA GTX480), and the max term in (6.3) is artificially set to 1.

32 cores and supports up to 1536 threads sharing the same 48 KB shared memory. We implement the dynamic programming algorithm via adjacency matrix multiplication for solving the all-pairs shortest paths problem, and set up a test-bench with which we can vary the following parameter settings: problem size, n, sub-block dimension, S_D, requested blocks B_r, and threads per core, $\mathcal{T}$. The relation between runtime and problem size n has been well studied in Chapter 4 and 5 and will not be investigated here. Instead, we will focus on verifying the effects of other parameters present in the model and of particular interest to the new extensions to the model: S_D, B_r, and $\mathcal{T}$.

Note that the values of some of the architecture parameters (e.g. Z and C) can be obtained from the specification of the architecture being used. Generally for NVIDIA GPUs, C is 32 if reads of threads are grouped; Z can be configured either to 16 KB or 48 KB for Fermi and later architectures. The value of architecture parameters, such as the memory latency L, can not be quantitatively determined reasonably from device specifications; however, for a

given architecture, these parameters will not change. In Eq. (6.9), C and L only contribute as a scale factor and can be represented by fixed coefficients as shown in Eq. (6.10).

6.3.1 Effect of $\sqrt{B_r}/\mathcal{T}$

According to Eq. (6.8) and Eq. (6.9), we infer that runtime should be linear with $\sqrt{B_r}/\mathcal{T}$ when memory bound and stay constant when compute bound. In Fig. 6.2, we fit a linear curve to the measured execution time of several runs, varying the settings for B_r and $\mathcal{T}$ as follows:

$$Time = a_1 \cdot \frac{\sqrt{B_r}}{\mathcal{T}} + a_0 \tag{6.10}$$

with $a_1 = 0.957$, $a_0 = 53.9$. The horizontal line represents the execution time when the application is compute bound (the empirical support for which is all clustered near the origin, it is plotted across the entire graph for easier visibility). For the curve fit of Eq. (6.10), $r^2 = 0.9916$, showing good linearity. The measured and predicted runtimes align with each other quite well.

Given a fixed problem size n that is reasonable on a fixed machine (we solve a graph of 8192 vertices on a GTX480), reducing $\sqrt{B_r}/\mathcal{T}$ will eventually transition the application performance from being constrained by memory latency to being constrained by computation. This observation is illustrated in Fig. 6.2. When we either choose a smaller B_r indirectly by increasing the sub-block dimension S_D, or configure the kernel to run with more threads $\mathcal{T}$, the runtime keeps dropping until a level where it turns flat. This is the point where transition happens, and after which the runtime is bounded by computation and independent of $\sqrt{B_r}/\mathcal{T}$.

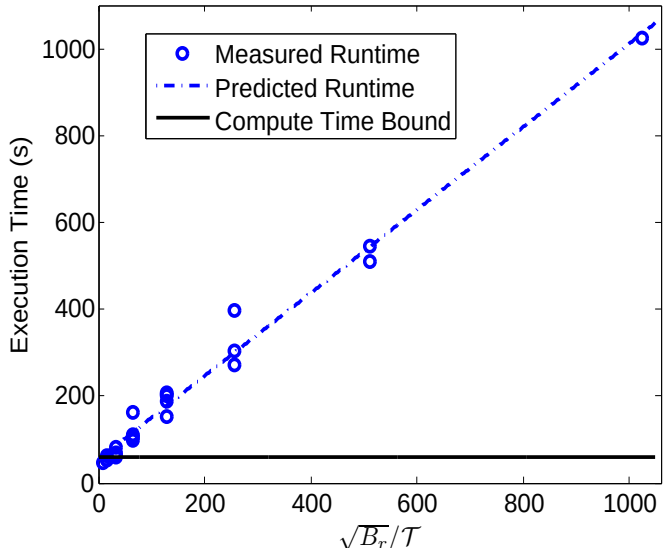


Figure 6.2: Runtime and model prediction in terms of $\sqrt{B_r}/\mathcal{T}$ for all-pairs shortest paths problem with 8192 vertices. Measurements are from various runtime configurations of $(B_r, \mathcal{T}, S_D)$, therefore with different B_a. Specifically, $B_r = (n/S_D)^2$, B_a is determined by Eq. (5.2).

6.3.2 Effect of $\mathcal{T}$

Next, we move on to investigate the effects of $\mathcal{T}$, the average threads per core, on runtime. According to Eq. (6.8) and Eq. (6.9), runtime should be inversely related to $\mathcal{T}$ when $\mathcal{T}S_D < L/C$ and stay constant if $\mathcal{T}S_D > L/C$. Fig. 6.3 illustrates this relationship fairly clearly. At small values of $\mathcal{T}$, runtime drops drastically as we increase $\mathcal{T}$. At the same time, the curves with larger S_D (smaller B_r) are more steeply sloped and flatten out for a lower value of $\mathcal{T}$. Runtime for the trials using $S_D = 64$, 32, and 16 converge to the same (flat) level at $\mathcal{T} = 8$, 16, and 32, respectively. All these observations are consistent with and can be explained by the model. In the range where $\mathcal{T}$ is low, latencies are not well hidden. When $\mathcal{T}$ gets big enough so that latencies are completely hidden, further increases in $\mathcal{T}$ do not bring any marginal benefits in terms of runtime. Larger S_D enables a smaller value of $\mathcal{T}$ as it is the product of both S_D and $\mathcal{T}$ that matters for latency hiding purposes.

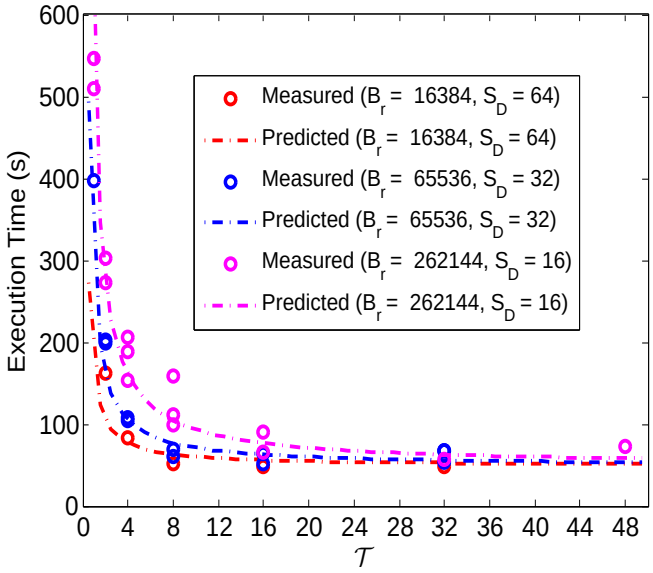


Figure 6.3: Empirically measured and model predicted runtimes in terms of $\mathcal{T}$ for all-pairs shortest paths problem with 8192 vertices. Measurements are from various runtime configurations of $(B_r, \mathcal{T}, S_D)$, therefore with different B_a. Specifically, $B_r = (n/S_D)^2$, B_a is determined by Eq. (5.2).

6.3.3 Effect of B_r

We continue by examining the effects of B_r on the application runtime. For a fixed value of $\mathcal{T}$, again, according to Eq. (6.8) and Eq. (6.9), runtime should increase with $\sqrt{B_r}$ when limited by memory latency and stay constant when compute bound. Fig. 6.4 shows how runtime changes with B_r for several distinct values of $\mathcal{T}$. Due to the sub-blocking mechanism, integral division of the matrix restricts the dimension of sub-blocks, S_D, to be processed per kernel, therefore the value options for B_r are limited. Nonetheless, we still see a reasonably good alignment between the model's predictions and the empirical measurements from the implementation for the range of $\mathcal{T}$ attempted. Curves for the larger values of $\mathcal{T}$, for example $\mathcal{T} = 16$ or 32, tend to converge to the flat level implied by sufficient latency hiding.

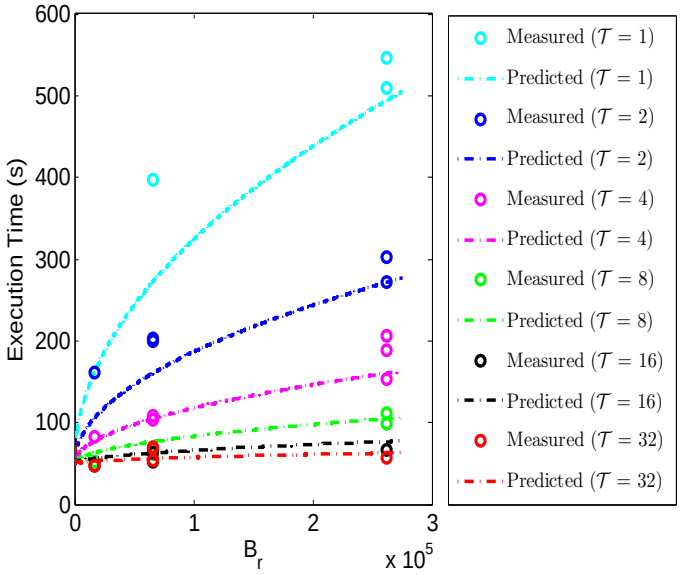


Figure 6.4: Empirically measured and model predicted runtimes in terms of B_r for all-pairs shortest paths problem with $n = 8192$ vertices. Measurements are from various runtime configurations of $(B_r, \mathcal{T}, S_D)$, therefore with different B_a. Specifically, $B_r = (n/S_D)^2$, B_a is determined by Eq. (5.2).

6.4 Discovering Unexpected Behavior

While the model predictions described above are not always perfect (see, for example, data points in Fig. 6.4 for very small values of $\mathcal{T}$), generally they do a very good job of explaining how performance trends with various performance-impacting factors. We will next illustrate the use of the model to discover an unexpected condition, in which the measured empirical performance was substantially different than the model prediction, and how that exposes additional uncertainty in realized performance of GPU applications in practical settings.

When launching a kernel on the GPU, the programmer specifies a configuration of that kernel, which includes things such as count of thread blocks, threads per thread block, etc. Other parameters, such as the number of active blocks, are automatically set as a function of the explicitly provided configuration according to Eq. (5.2). For an specific example run ($n = 8192$, $S_D = 32$, $B_r = 65,536$, $\mathcal{T} = 4$, $B_a = 4$), the measured execution time of 404 s

was 4 times longer than the predicted execution time of 105 s. As part of our investigation into this anomaly, we artificially increased the shared memory requested by the application, thereby coercing it to use a B_a of 1 instead of the automatically determined value of 4. When we ran this altered version of the application (which we verified still provided the correct result), the execution time was 108 s, much more in line with the model's prediction.

The above anomaly occurred more often than just in this individual case. Fig. 6.5 shows execution time vs. $\mathcal{T}$ for several cases of automatically determined numbers of active blocks and artificially lowered numbers of active blocks ($B_a = 1$ or 2). In many cases, requesting more memory than was truly needed resulted in substantial performance gains.

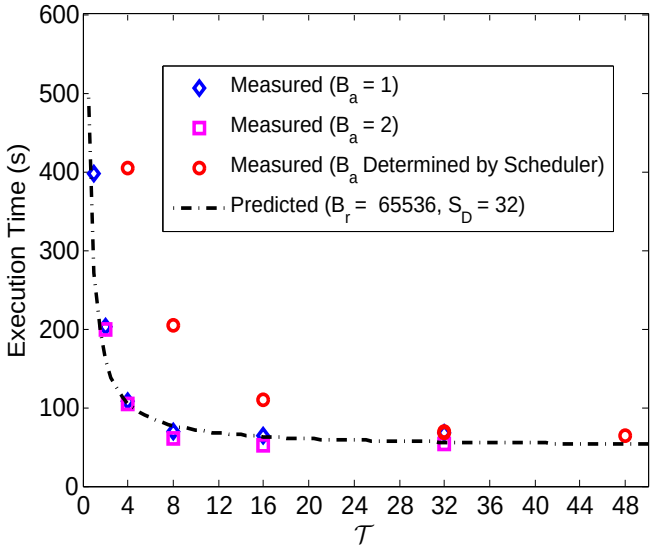


Figure 6.5: Empirical measures of all-pairs shortest paths runtime for scenarios when $B_a = 1$, $B_a = 2$ and B_a is determined automatically by the scheduler. The APSP problem with 8192 vertices is divided into sub-blocks of dimension 32.

While we do not have a satisfactory explanation for the execution time realized when B_a is at its default value, we do conclude that there is sufficient uncertainty inherent in the performance achievable on modern GPUs that the use of well understood performance models can at the very least help to identify circumstances where the application is not performing

as it should. The scheduler works in a way that the number of active blocks scheduled on each multiprocessor cannot be directly controlled. Yet, as illustrated in Fig. 6.5, there are circumstances where it is clearly beneficial to the application to be able to control the active blocks scheduled. As illustrated by our experience, however, it can be indirectly changed by altering the shared memory requested by each kernel.

Discrepancies between the model predictions and measured performance can draw the developer's attention to these cases for focused investigation on the causal relation between configuration requested and the performance achieved. This also indicates to GPU manufacturers that allowing programmers to directly manipulate the number of active blocks scheduled on multiprocessors may be warranted.

Chapter 7

Conclusion and Future Work

The thesis has presented two performance models that are well suited for highly-threaded many-core machines, particularly GPUs, and bridged both together for performance analysis of algorithms. The two models have different emphases: the TMM model for high-level theoretical analysis of asymptotic performance on GPUs, particularly analyzing the compute complexity and memory complexity such that whether the underlying algorithm is compute bound or memory bound can be judged in terms of how well the memory latencies are hidden by the massive threads; the second model for lower-level calibrated real execution time analysis and prediction considering runtime configuration, caching, and scheduling. We also develop an integrated analytical framework extending the two existing models, combine them by modeling GPU scheduling mechanisms and incorporating both the computation complexity and memory complexity as critical performance-impacting measures. By doing so, our analytical framework is able to capture the parallelism, latency-hiding, and occupancy together in one model, reflecting performance bottlenecks, reducing the configuration space for kernel execution, and predicting achievable execution times as well as how execution time will trend as the various parameters scale.

A large number of problems and classic algorithms, including four all-pairs shortest paths algorithms, FFT, list ranking, string matching using suffix tree/array, merge sort, Bloom filter, and DNA sequencing, are analyzed under this analytical framework with ample details. A parallel Bloom filters algorithm for BLASTN on GPUs is designed with 35-fold speedup achieved. A synthetic micro-benchmark for hashing is also implemented allowing flexible manipulation of hashing ranges and choice of on-chip/off-chip memory spaces for investigation of the impact of various memory access patterns on performance.

With extensive experimental validation as well as data from other research literature on a wide range of GPU machines, we compare the analytically predicted results with empirical results. This comparison indicates that our model is effective at explaining empirical performance for highly-threaded many-core GPUs. In particular, the TMM model seems to be effective at predicting the effect of scaling the problem size, overall thread count, the machine characteristics like memory latency and local memory size on the trend of algorithm performance; the calibrated model is accurate to reflect the effect of changing the sub-block dimension, thread count per block, requested thread blocks, and local memory usage on the real execution time of algorithms. In addition, the model points to at least one circumstance in which the occupancy decisions automatically made by the scheduler are clearly sub-optimal.

There are several possible directions of future work:

First, more algorithms and benchmarks can be analyzed and developed. Most of the algorithms analyzed in this thesis are relatively straightforward with ample parallelism. More complex algorithms, which may incur frequent data dependency, branch divergence, and possibly irregular remote memory accesses, can also be analyzed via this model. The model

is expected to indicate the unsuitability of such algorithms on GPUs, or guidance from the model to re-design them so as to be adapted on GPU in a performance-oriented way.

Second, the TMM model only captures the performance of a single machine with a 2-level memory hierarchy. While we assume that it is global memory vs. memory local to multiprocessors, in principle, it can be any two levels of fast and slow memory. The 2-level memory hierarchy can be extended to multi-level hierarchies which are becoming increasingly common, for example considering algorithms running on systems containing more than one GPU in a distributed environment.

Finally, more other highly-threaded many-core machines can be investigated, such as the Yarc data machine/Cray XMT and AMD GPUs, as an extension to further validate the applicability of the model in this thesis to a broader set of real machines.

References

[1] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. of 19th ACM Symposium on Theory of Computing*, pages 305–314, 1987.

[2] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *Proc. of 28th Symposium on Foundations of Computer Science*, pages 204–216, 1987.

[3] Alok Aggarwal and J Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[4] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[5] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the GPU. *ACM Trans. on Graphics*, 28(5), December 2009.

[6] Bowen Alpern and Larry Carter. Modeling parallel computers as memory hierarchies. *Massively Parallel Computers*, 10598, 1993.

[7] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2/3):72–109, 1994.

[8] Bowen Alpern, Larry Carter, and Ted Selker. Visualizing computer memory architectures. In *Proc. of 1st Conf. on Visualization*, pages 107–113, 1990.

[9] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 1990.

[10] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, pages 1–6, New York, NY, USA, 1990. ACM.

[11] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proc. of 20th Symp. on Parallelism in Algorithms and Architectures*, pages 197–206, 2008.

[12] David A. Bader and Guojing Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *J. Parallel Distrib. Comput.*, 66(11), November 2006.

[13] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-Mei Hwu. An adaptive performance modeling tool for GPU architectures. In *Proc. of Symp. on Principles and Practice of Parallel Programming*, pages 105–114, 2010.

[14] J. Barnat, P. Bauch, L. Brim, and M. Ceska. Computing strongly connected components in parallel on CUDA. In *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 544–555, 2011.

[15] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[16] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[17] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious b-trees. In *Proc. of 17th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 228–237, 2005.

[18] P. Bieganski, J. Riedl, J.V. Cartis, and E.F. Retzel. Generalized suffix trees for biological sequence data: applications and implementation. In *Proc. of Twenty-Seventh Hawaii International Conference on System Sciences*, volume 5, pages 35–44, 1994.

[19] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proc. 19th ACM-SIAM Symp. Discrete Algorithms*, pages 501–510, 2008.

[20] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proc. of 23rd ACM Symp. on Parallelism in Algorithms and Architectures*, pages 355–366, 2011.

[21] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[22] Vincenzo Bonnici, Alfredo Ferro, Rosalba Giugno, Alfredo Pulvirenti, and Dennis Shasha. Enhancing graph database indexing by suffix tree structure. In *Proc. of 5th IAPR International Conference on Pattern Recognition in Bioinformatics*, pages 195–203, 2010.

[23] Otakar Boruvka. O Jistém Problému Minimálním (About a Certain Minimal Problem) (in Czech, German summary). *Práce Mor. Prírodoved. Spol. v Brne III*, 3, 1926.

[24] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. of 10th Int'l Conf. on Very Large Databases*, pages 323–333, 1984.

[25] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[26] J.D. Buhler, J.M. Lancaster, A.C. Jacob, and R.D. Chamberlain. Mercury BLASTN: Faster DNA sequence comparison using a streaming hardware architecture. In *Proc. of Reconfigurable Systems Summer Institute*, June 2007.

[27] Jeremy D. Buhler, Kunal Agrawal, Peng Li, and Roger D. Chamberlain. Efficient deadlock avoidance for streaming computation with filtering. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 235–246, 2012.

[28] Aydın Buluç, John R. Gilbert, and Ceren Budak. Solving path problems on the GPU. *Parallel Comput.*, 36(5-6):241–253, June 2010.

[29] Daniel Cederman and Philippas Tsigas. GPU-Quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, January 2010.

[30] Roger D. Chamberlain, Ron K. Cytron, Mark A. Franklin, and Ronald S. Indeck. The Mercury system: exploiting truly fast hardware for data search. In *Proc. of Int'l Workshop on Storage Network Architecture and Parallel I/Os*, pages 65–72, 2003.

[31] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.*, 68(10), October 2008.

[32] Wenlin Chen, Yixin Chen, and Kilian Q. Weinberger. Fast flux discriminant for large-scale sparse nonlinear classification. In *Proc. of 20th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining (KDD)*, 2014.

[33] Wenlin Chen, Kilian Q. Weinberger, and Yixin Chen. Maximum variance correction with application to a* search. In *Proc. of 30th Intl. Conf. on Machine Learning (ICML)*, 2013.

[34] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proc. of 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2010.

[35] Ka Wong Chong, Yijie Han, and Tak-Wah Lam. Concurrent threads and optimal parallel minimum spanning trees algorithm. *J. ACM*, 48:297–323, 2001.

[36] Rezaul A. Chowdhury, Francesco Silvestri, Brandon Blakeley, and Vijaya Ramachandran. Oblivious algorithms for multicores and network of processors. In *Proc. of 24th IEEE Int'l Parallel and Distributed Processing Symp.*, pages 1–12, April 2010.

[37] Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious dynamic programming. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 591–600, New York, NY, USA, 2006. ACM.

[38] Rezaul Alam Chowdhury and Vijaya Ramachandran. The cache-oblivious Gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *Proc. of 19th ACM Symp. on Parallel Algorithms and Architectures*, pages 71–80, 2007.

[39] Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proc. of 20th Symp. on Parallelism in Algorithms and Architectures*, pages 207–216, 2008.

[40] Richard Cole, Philip N. Klein, and Robert E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, pages 243–250, New York, NY, USA, 1996. ACM.

[41] Richard Cole and Vijaya Ramachandran. Efficient resource oblivious algorithms for multicores. *CoRR*, abs/1103.4071:v1, 2011.

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

[43] L.B. Costa, S. Al-Kiswany, and M. Ripeanu. GPU support for batch oriented workloads. In *Proc. of 28th Int'l Performance Computing and Communications Conf.*, pages 231–238, Dec. 2009.

[44] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Proc. of 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 1993.

[45] S. Datta, P. Beeraka, and R. Sass. RC-BLASTn: Implementation and evaluation of the BLASTn scan function. In *Proc. of Symp. on Field Programmable Custom Computing Machines*, pages 88–95, 2009.

[46] Frank Dehne and Kumanan Yogaratnam. Exploring the limits of GPUs with parallel graph algorithms. *CoRR*, abs/1002.4482, 2010.

[47] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[48] G. Encarnaijao, N. Sebastiao, and N. Roma. Advantages and GPU implementation of high-performance indexed DNA search based on suffix arrays. In *Proc. of Int'l Conf. on High Performance Computing and Simulation*, pages 49–55, 2011.

[49] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, June 1962.

[50] National Center for Biological Information. Growth of GenBank. `http://www.ncbi.nlm.nih.gov/genbank/genbankstats.html`, February 2009.

[51] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proc. of 10th ACM Symp. on Theory of computing*, 1978.

[52] M Frigo, C E Leiserson, H Prokop, and S Ramachandran. Cache-oblivious algorithms. In *Proc. of 40th Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[53] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 271–280, 2006.

[54] Yong Fu, Mo Sha, Chengjie Wu, Andrew Kutta, Anna Leavey, Chenyang Lu, Humberto Gonzalez, Weining Wang, Bill Drake, Yixin Chen, and Pratim Biswas. Thermal modeling for a HVAC controlled real-life auditorium. In *International Conference on Distributed Computing Systems (ICDCS'14)*, July 2014.

[55] N. Fujimoto. Faster matrix-vector multiplication on GeForce 8800GTX. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS).*, pages 1–8, 2008.

[56] Leslie M. Goldschlager. A universal interconnection pattern for parallel computers. *J. ACM*, October 1982.

[57] X. Gong, W. Qian, Y. Yan, and A. Zhou. Bloom filter-based XML packets filtering for millions of path queries. In *21st Int'l Conf. on Data Engineering*, pages 890–901, 2005.

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

[59] Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A memory model for scientific algorithms on graphics processors. In *Proc. of ACM/IEEE Conf. on Supercomputing.*, 2006.

[60] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete Fourier transforms on graphics processors. In *Proc. of ACM/IEEE Supercomputing Conf.*, 2008.

[61] Naga K. Govindaraju and Dinesh Manocha. Cache-efficient numerical algorithms using graphics hardware. *Parallel Comput.*, 33(10-11):663–684, November 2007.

[62] L.L. Gremillion. Designing a Bloom filter for differential file access. *Communications of the ACM*, 25(9):600–604, September 1982.

[63] Jesse D. Hall, Nathan A. Carr, and John C. Hart. Cache and bandwidth aware matrix multiplication on the GPU. Technical report, University of Illinois at Urbana-Champaign, 2003.

[64] Sardar Anisul Haque, Marc Moreno Maza, and Ning Xie. A many-core machine model for designing algorithms with minimum parallelism overheads. In *Proc. of High Performance Computing Symp.*, 2013.

[65] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[66] Pawan Harish, Vibhav Vineet, and P. J. Narayanan. Large Graph Algorithms for Massively Multithreaded Architectures. Technical Report IIIT/TR/2009/74, International Institute of Information Technology Hyderabad.

[67] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *Proc. of ACM/IEEE Supercomputing Conf.*, 2007.

[68] Dan He. Using suffix tree to discover complex repetitive patterns in DNA sequences. In *Proc. of 28th Int'l Conf. of the IEEE Engineering in Medicine and Biology Society*, pages 3474–3477, 2006.

[69] Zhengyu He and Bo Hong. Dynamically tuned push-relabel algorithm for the maximum flow problem on CPU-GPU-Hybrid platforms. In *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 1–10, 2010.

[70] M.C. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt. Single pass streaming BLAST on FPGAs. *Parallel Computing*, 33:741–756, 2007.

[71] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proc. of 16th ACM Symp. on Principles and Practice of Parallel Programming*, 2011.

[72] Sungpack Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 78–88, 2011.

[73] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. of 36th Int'l Symp. on Computer Architecture*, pages 152–163, 2009.

[74] Mohamed Hussein, Amitabh Varshney, and Larry Davis. On Implementing Graph Cuts on CUDA. In *Proc. of Workshop on General Purpose Processing on Graphics Processing Units. (GPGPU 2008)*, 2008.

[75] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Siow Mong Goh. Scalable framework for mapping streaming applications onto multi-GPU systems. *ACM SIGPLAN Notices*, 47(8):1–10, 2012.

[76] Changhao Jiang and M. Snir. Automatic tuning matrix multiplication performance on graphics hardware. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 185–194, 2005.

[77] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, January 1977.

[78] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2), March 1995.

[79] Richard M. Karp. A survey of parallel algorithms for shared-memory machines. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1988.

[80] Richard M. Karp and Avi Wigderson. A fast parallel algorithm for the maximal independent set problem. *J. ACM*, 32(4), October 1985.

[81] Gary J. Katz and Joseph T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proc of 23rd ACM SIGGRAPH/EUROGRAPHICS Symp. on Graphics Hardware*, 2008.

[82] Yooseong Kim and Aviral Shrivastava. Cumapz: a tool to analyze memory access patterns in CUDA. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 128–133, 2011.

[83] Peter Kipfer and Rüdiger Westermann. Improved GPU sorting. In Matt Pharr, editor, *GPU Gems 2*, chapter 46. Addison Wesley, March 2005.

[84] J. Steven Kirtzic and Ovidiu Daescu. A parallel algorithm development model for the GPU architecture. In *Proc. of Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, 2012.

[85] Kishore Kothapalli, Rishabh Mukherjee, M. Suhail Rehman, Suryakant Patidar, P. J. Narayanan, and Kannan Srinathan. A performance prediction model for the CUDA GPGPU platform. In *Proceedings of International Conference on High Performance Computing (HiPC)*, pages 463–472, December 2009.

[86] Christian Kreibich and Jon Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proc. of the 2nd Workshop on Hot Topics in Networks*, 2003.

[87] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, A. Jacob, and J. Lancaster. Biosequence similarity search on the Mercury system. *J. VLSI Signal Processing*, 49:101–121, October 2007.

[88] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proc. of 37th Int'l Symp. on Computer Architecture*, pages 451–460, 2010.

[89] Jr. Lestor R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA, 1962.

[90] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D Chamberlain. Deadlock avoidance for streaming computations with filtering. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 243–252. ACM, 2010.

[91] Cheng Ling, Khaled Benkrid, and Tsuyoshi Hamada. A parameterisable and scalable Smith-Waterman algorithm implementation on CUDA-compatible GPUs. In *Proceedings of 7th IEEE Symposium on Application Specific Processors*, SASP '09, pages 94–100, 2009.

[92] Weiguo Liu, Wolfgang Muller-Wittig, and Bertil Schmidt. Performance predictions for general-purpose computation on GPUs. In *Proc. of Int'l Conf. on Parallel Processing*, 2007.

[93] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Muller-Wittig. Streaming algorithms for biological sequence alignment on GPUs. *IEEE Trans. Parallel Distrib. Syst.*, pages 1270–1281, 2007.

[94] Yongchao Liu, Bertil Schmidt, and Douglas Maskell. DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI. *BMC Bioinformatics*, 12(1):85, 2011.

[95] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. CUDASW++2.0: Enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3, 2010.

[96] M Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, STOC '85, pages 1–10, New York, NY, USA, 1985. ACM.

[97] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

[98] Agnieszka Lupinska. Parallel implematation of flow and matching algorithms. *CoRR*, abs/1110.6231, 2011.

[99] Lin Ma, K. Agrawal, and R.D. Chamberlain. A memory access model for highly-threaded many-core architectures. In *Proc. of IEEE 18th Int'l Conf. on Parallel and Distributed Systems (ICPADS)*, pages 339–347, 2012.

[100] Lin Ma, Kunal Agrawal, and Roger Chamberlain. Theoretical analysis of classic algorithms on highly-threaded many-core GPUs. In *Proc. of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 391–392, 2014.

[101] Lin Ma, Kunal Agrawal, and Roger D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202–215, January 2014.

[102] Lin Ma, Kunal Agrawal, and Roger D. Chamberlain. Analysis of classic algorithms on GPUs. In *Proc. of the 12th ACM/IEEE Int'l Conf. on High Performance Computing and Simulation (HPCS)*, pages 65–73, 2014.

[103] Lin Ma, Roger Chamberlain, and Kunal Agrawal. Performance modeling for highly-threaded many-core GPUs. In *Proc. of the 25th Int'l Conf. on Application-specific Systems, Architectures and Processors*, pages 84–91, 2014.

[104] Lin Ma and Roger D. Chamberlain. A performance model for memory bandwidth constrained applications on graphics engines. In *Proc. of Int'l Conf. on Application-specific Systems, Architectures and Processors*, pages 24–32, 2012.

[105] Lin Ma, Roger D. Chamberlain, Jeremy D. Buhler, and Mark A. Franklin. Bloom filter performance on graphics engines. In *Proc. of Int'l Conf. on Parallel Processing*, pages 522–531, 2011.

[106] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *Proc. of the ACM-SIAM Symp. on Discrete Algorithms*, 1990.

[107] Pedro J. Martín, Roberto Torres, and Antonio Gavilanes. CUDA solutions for the SSSP problem. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 904–913, Berlin, Heidelberg, 2009. Springer-Verlag.

[108] Kazuya Matsumoto, Naohito Nakasato, and Stanislav G. Sedukhin. Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system. In *Proc. of IEEE Int'l Conf. on High Performance Computing and Communications*, pages 145–152, 2011.

[109] M.D. McIlroy. Development of a spelling list. *IEEE Trans. on Communications*, 30(1):91–99, 1982.

[110] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 107–116, New York, NY, USA, 2012. ACM.

[111] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *Proc. of 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 117–128, 2012.

[112] Paulius Micikevicius. General parallel computation on commodity graphics hardware: Case study with the all-pairs shortest paths problem. In *PDPTA*, pages 1359–1365, 2004.

[113] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '03, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[114] S. Mu, X. Zhang, N. Zhang, J. Lu, Y.S. Deng, and S. Zhang. IP routing processing with graphic processors. In *Proc. of Conf. on Design, Automation and Test in Europe*, pages 93–98, 2010.

[115] J.K. Mullin and D.J. Margoliash. A tale of three spelling checkers. *Software – Practice and Experience*, 20(6):625–630, 1990.

[116] Koji Nakano. The hierarchical memory machine model for GPUs. In *Proc. of Int'l Parallel and Distributed Processing Symp. Workshops & PhD Forum*, 2013.

[117] Sadegh Nobari, Thanh-Tung Cao, Panagiotis Karras, and Stéphane Bressan. Scalable parallel minimum spanning forest computation. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 205–214, New York, NY, USA, 2012. ACM.

[118] NVIDIA. CUDA Programming Guide 5.0. October 2012.

[119] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[120] A.K. Parakh, M. Balakrishnan, and K. Paul. Performance estimation of GPUs with cache. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pages 2384–2393, 2012.

[121] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proc. of 16th Int'l Conf. on Supercomputing*, pages 189–198, 2002.

[122] Seth Pettie and Vijaya Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.*, 31(6), June 2002.

[123] Benjarath Phoophakdee and Mohammed J. Zaki. Genome-scale disk-based suffix tree indexing. In *Proc. of ACM SIGMOD Int'l Conference on Management of Data*, pages 833–844, 2007.

[124] Chung Keung Poon and Vijaya Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In *Proceedings of 8th International Symposium on Algorithms and Computation (ISAAC)*, pages 212–222, 1997.

[125] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technology Journal*, 36:1389–1401, 1957.

[126] Harald Prokop. Cache-oblivious algorithms, 1999. Master's thesis, MIT.

[127] M.O. Rabin. Probabilistic algorithms. *Algorithms and Complexity*, pages 21–39, 1976.

[128] M.V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Trans. on Computers*, 46(12):1378–1381, December 1997.

[129] F. Rasheed, M. Alshalalfa, and R. Alhajj. Efficient periodicity mining in time series databases using suffix trees. *IEEE Trans. on Knowledge and Data Engineering*, 23(1):79–94, 2011.

[130] M. Suhail Rehman, Kishore Kothapalli, and P. J. Narayanan. Fast and scalable list ranking on the GPU. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 235–243, New York, NY, USA, 2009. ACM.

[131] Mohammed S. Rehman. Exploring irregular memory access applications on the GPU. Master's thesis, International Institute of Information Technology, Hyderabad, India, 2010.

[132] Scott Rostrup, Shweta Srivastava, and Kishore Singhal. Fast and memory-efficient minimum spanning tree on the GPU. *Int. J. Comput. Sci. Eng.*, 8(1):21–33, February 2013.

[133] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-Mei Hwu. Program optimization space pruning for a multithreaded GPU. In *Proc. of 6th IEEE/ACM Int'l Symp. on Code Generation and Optimization*, pages 195–204, 2008.

[134] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proc. of IEEE Int'l Parallel and Distributed Processing Symp.*, pages 1–10, 2009.

[135] Olaf Schenk, Matthias Christen, and Helmar Burkhart. Algorithmic performance studies on graphics processing units. *J. Parallel Distrib. Comput.*, 68(10), October 2008.

[136] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proc. of 22nd ACM SIGGRAPH/EUROGRAPHICS Symp. on Graphics Hardware*, pages 97–106, 2007.

[137] H. Shi, B. Schmidt, W. Liu, and W. Muller-Wittig. Accelerating error correction in high-throughput short-read DNA sequencing data with CUDA. In *Proc. of Int'l Parallel and Distributed Processing Symp.*, 2009.

[138] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proceedings of 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 11–22, 2012.

[139] Erik Sintorn and Ulf Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381 – 1388, 2008.

[140] S. Solomon, P. Thulasiraman, and R.K. Thulasiram. Exploiting Parallelism in Iterative Irregular Maxflow Computations on GPU Accelerators. In *Proceedings of 12th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 297–304, 2010.

[141] Robert Solovay and Volker Strassen. A fast Monte-Carlo test for primality. *SIAM J. Comput.*, 6(1):84–85, 1977.

[142] J. Soman, K. Kishore, and P. J. Narayanan. A fast GPU algorithm for graph connectivity. In *Proceedings of IEEE International Parallel Distributed Processing Symposium, Workshops and Phd Forum (IPDPSW)*, pages 1–8, 2010.

[143] Jyothish Soman, Kishore Kothapalli, and P. J. Narayanan. Some GPU algorithms for graph connected components and spanning tree. *Parallel Processing Letters*, 20(4):325–339, 2010.

[144] Henrik Stranneheim, Max Käller, Tobias Allander, Björn Andersson, Lars Arvestad, and Joakim Lundeberg. Classification of DNA sequences using Bloom filters. *Bioinformatics*, 26:1595–1600, July 2010.

[145] I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 513–522, 2010.

[146] P. Valdurez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. on Database Systems*, 9(1):133–161, 1984.

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

[148] Leslie G. Valiant. A bridging model for multi-core computing. *Journal of Computer and System Sciences*, 77(1), January 2011.

[149] V. Vineet and P. J. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1–8, 2008.

[150] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 167–171, New York, NY, USA, 2009. ACM.

[151] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In *Proc. of Conf. on High Performance Graphics*, pages 167–171, 2009.

[152] Uzi Vishkin, George C. Caragea, and Bryant Lee. Models for advancing PRAM and other algorithms into parallel programs for a PRAM-On-Chip platform. In *Handbook of Parallel Computing: Models, Algorithms and Applications*. CRC Press, 2007.

[153] Jeffrey Scott Vitter and Mark H. Nodine. Large-scale sorting in uniform memory hierarchies. *J. Parallel Distrib. Comput.*, 17(1-2):107–114, January 1993.

[154] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12:148–169, 1993.

[155] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12:110–147, 1994.

[156] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proc. of ACM/IEEE Conf. on Supercomputing*, 2008.

[157] Min-Feng Wang, Yen-Ching Wu, and Meng-Feng Tsai. Exploiting frequent episodes in weighted suffix tree to improve intrusion detection system. In *Proc. of 22nd Int'l Conf. on Advanced Information Networking and Applications - Workshops*, pages 1246–1252, 2008.

[158] W. Wang, S. Guo, F. Yang, and J. Chen. GPU-based fast minimum spanning tree using data parallel primitives. In *Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on*, page 14. IEEE.

[159] W. Wang, Y. Huang, and S. Guo. Design and implementation of GPU-based Prim's algorithm. *International Journal of Modern Education and Computer Science (IJMECS)*, 3(4):55, 2011.

[160] Stephen Warshall. A theorem on Boolean matrices. *J. ACM*, 9(1):11–12, January 1962.

[161] Chengjie Wu, Mo Sha, Dolvara Gunatilaka, Abusayeed Saifullah, Chenyang Lu, and Yixin Chen. Analysis of EDF scheduling for wireless sensor-actuator networks. In *IEEE/ACM Symposium on Quality of Service (IWQoS'14)*, May 2014.

[162] Chengjie Wu, You Xu, Yixin Chen, and Chenyang Lu. Submodular game for distributed application allocation in shared sensor networks. In *The 31st IEEE International Conference on Computer Communications (INFOCOM'12)*, March 2012.

[163] James C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Cornell University, Ithaca, NY, USA, 1979.

[164] Zhixiang Xu, Olivier Chapelle, and Kilian Q. Weinberger. The greedy miser: Learning under test-time budgets. In *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, pages 1175–1182, 2012.

[165] Zhixiang Xu, Matt Kusner, Kilian Q. Weinberger, and Minmin Chen. Cost-sensitive tree of classifiers. In *Proceedings of The 30th International Conference on Machine Learning*, pages 133–141, 2013.

[166] Zhixiang Eddie Xu, Matt J Kusner, Kilian Q. Weinberger, Minmin Chen, and Olivier Chapelle. Classifier cascades and trees for minimizing feature evaluation cost. *Journal of Machine Learning Research*, 15:2113–2144, 2014.

[167] Xiaochun Ye, Dongrui Fan, Wei Lin, Nan Yuan, and P. Ienne. High performance comparison-based sorting algorithm on many-core GPUs. In *Proceedings of IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 1–10, 2010.

[168] Dell Zhang and Wee Sun Lee. Extracting key-substring-group features for text classification. In *Proc. of 12th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pages 474–483, 2006.

[169] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In *Proc. of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[170] Yao Zhang and J.D. Owens. A quantitative performance analysis model for GPU architectures. In *Proc. of IEEE Int'l Symp. on High Performance Computer Architecture*, pages 382–393, February 2011.

[171] Yixin Zhuang, Ming Zou, Nathan Carr, and Tao Ju. A general and efficient method for finding cycles in 3D curve networks. *ACM Trans. Graph.*, 32(6), November 2013.

[172] Ming Zou, Tao Ju, and Nathan Carr. An algorithm for triangulating multiple 3D polygons. *Computer Graphics Forum*, 32(5):157–166, 2013.

Vita

Lin Ma

Degrees

Ph.D. Washington Univ. in St.Louis, Computer Science, Dec. 2014
M.S. Washington Univ. in St.Louis, Computer Science, May 2014
M.E. Beijing Univ. of Posts & Telecom., Computer Eng., Mar. 2008
B.S. North China Electric Power Univ., Electrical Eng., Jun. 2005
B.A. North China Electric Power Univ., English L & L, Jun. 2005

Publications

Lin Ma, Roger D. Chamberlain, Jeremy D. Buhler, Mark A. Franklin (2011). Bloom Filter Performance on Graphics Engines, *Proc. of the 40th Int'l Conf. on Parallel Processing (ICPP)* : 522–531.

Lin Ma, Kunal Agrawal, and Roger D. Chamberlain (2012). A Memory Access Model for Highly-threaded Many-core Architectures, *Proc. of IEEE 18th Int'l Conf. on Parallel and Distributed Systems (ICPADS)* : 339–347.

Lin Ma, and Roger D. Chamberlain (2012). A Performance Model for Memory Bandwidth Constrained Applications on Graphics Engines, *Proc. of the 23th Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP)* : 24–31.

Lin Ma, Kunal Agrawal, and Roger D. Chamberlain (2014). A Memory Access Model for Highly-threaded Many-core Architectures, *Future Generation Computer Systems (FGCS)*. **30**: 202–215. [**Impact Factor: 2.639**].

Lin Ma, Kunal Agrawal, and Roger D. Chamberlain (2014). Theoretical Analysis of Classic Algorithms on Highly-threaded Many-core GPUs, *Proc. of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* : 391–392.

Lin Ma, Roger D. Chamberlain, and Kunal Agrawal (2014). Performance Modeling for Highly-threaded Many-core GPUs, *Proc. of the*

25th Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP) : 84–91.

Lin Ma, Kunal Agrawal, and Roger D. Chamberlain (2014). Analysis of Classic Algorithms on GPUs, *Proc. of the 12th ACM/IEEE Int'l Conf. on High Performance Computing and Simulation (HPCS)* : 65–73. [**Outstanding Paper Award Runner-up**].

December 2014