Future Generation Computer Systems 30 (2014) 202–215

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A memory access model for highly-threaded many-core architectures☆

Lin Ma*, Kunal Agrawal, Roger D. Chamberlain

Department of Computer Science and Engineering, Washington University in St. Louis, United States

HIGHLIGHTS

- We design a memory model to analyze algorithms for highly-threaded many-core systems.
- The model captures significant factors of performance: work, span, and memory accesses.
- We show the model is better than PRAM by applying both to 4 shortest paths algorithms.
- Empirical performance is effectively predicted by our model in many circumstances.
- It is the first formalized asymptotic model helpful for algorithm design on many-cores.

ARTICLE INFO

Article history:
Received 25 January 2013
Received in revised form
14 May 2013
Accepted 17 June 2013
Available online 15 July 2013

Keywords:
PRAM
TMM
All Pairs Shortest Paths (APSP)
Highly-threaded many-core
Memory access model

ABSTRACT

A number of highly-threaded, many-core architectures hide memory-access latency by low-overhead context switching among a large number of threads. The speedup of a program on these machines depends on how well the latency is hidden. If the number of threads were infinite, theoretically, these machines could provide the performance predicted by the PRAM analysis of these programs. However, the number of threads per processor is not infinite, and is constrained by both hardware and algorithmic limits. In this paper, we introduce the Threaded Many-core Memory (TMM) model which is meant to capture the important characteristics of these highly-threaded, many-core machines. Since we model some important machine parameters of these machines, we expect analysis under this model to provide a more fine-grained and accurate performance prediction than the PRAM analysis. We analyze 4 algorithms for the classic all pairs shortest paths problem under this model. We find that even when two algorithms have the same PRAM performance, our model predicts different performance for some settings of machine parameters. For example, for dense graphs, the dynamic programming algorithm and Johnson's algorithm have the same performance in the PRAM model. However, our model predicts different performance for large enough memory-access latency and validates the intuition that the dynamic programming algorithm performs better on these machines. We validate several predictions made by our model using empirical measurements on an instantiation of a highly-threaded, many-core machine, namely the NVIDIA GTX 480.

© 2013 The Authors. Published by Elsevier B.V. All rights reserved.

1. Introduction

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 low-overhead 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. Researchers have designed algorithms to solve many interesting problems for these devices, such as GPU sorting or hashing [1–4], linear algebra [5–7], dynamic programming [8,9], graph algorithms [10–13], and many other classic algorithms [14,15]. These projects generally report impressive gains in performance. These devices appear to be here to stay. While there is a lot of folk wisdom on how to design good algorithms for these highly-threaded machines, in addition to a significant body of work on performance analysis [16–20], there are no systematic theoretical models to analyze the performance of programs on these machines. We are interested in analyzing and characterizing performance of algo-

☆ This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

* Corresponding author. Tel.: +1 3144986029.
E-mail address: lin.ma@cse.wustl.edu (L. Ma).

0167-739X/$ – see front matter © 2013 The Authors. Published by Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.future.2013.06.020

rithms on these highly-threaded, many-core machines in a more abstract, algorithmic, and systematic manner.

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. The most fundamental model that is used to analyze sequential algorithms is the Random Access Machine (RAM) model [21], 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 [22] 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. There are also a number of other models that consider the memory access costs of sequential algorithms in different ways [23–29].

For parallel computing, the analogue for the RAM model is the Parallel Random Access Machine (PRAM) model [30], and there is a large body of work describing and analyzing algorithms in the PRAM model [31,32]. 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 by the algorithm takes unit time. For modern machines, however, this assumption seldom holds. Therefore, researchers have designed various models that capture memory hierarchies for various types of machines such as distributed memory machines [33–35], shared memory machines and multi-cores [36–40], or the combination of the two [41,42].

All of these models capture particular capabilities and properties of the respective target machines, namely shared memory machines or distributed memory machines. While superficially highly-threaded, many-core machines such as GPUs are shared memory machines, their characteristics are very different from traditional multi-core or multiprocessor shared memory machines. The most important distinction between the 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 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. Highly-threaded many-cores 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 such machines, since we can ignore the effect of memory transfers which is exactly what 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. Since no highly-threaded, many-core machine allows an infinite number of threads, it is important to understand both (1) how many threads does a particular algorithm need to achieve PRAM performance, and (2) how does an algorithm perform when it has fewer threads than required to get PRAM performance? In this paper, we attempt to characterize these properties of algorithms. To motivate this enterprise and to understand the importance of high thread counts on highly-threaded, many-core machines, let us consider a simple application that performs Bloom filter set membership tests on an input stream of biosequence data on GPUs [3]. The problem is embarrassingly parallel, each set membership test is independent of every other membership test. Fig. 1 shows the performance of this application, varying the number of threads per processor core, for two distinct GPUs. For both machines, 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.

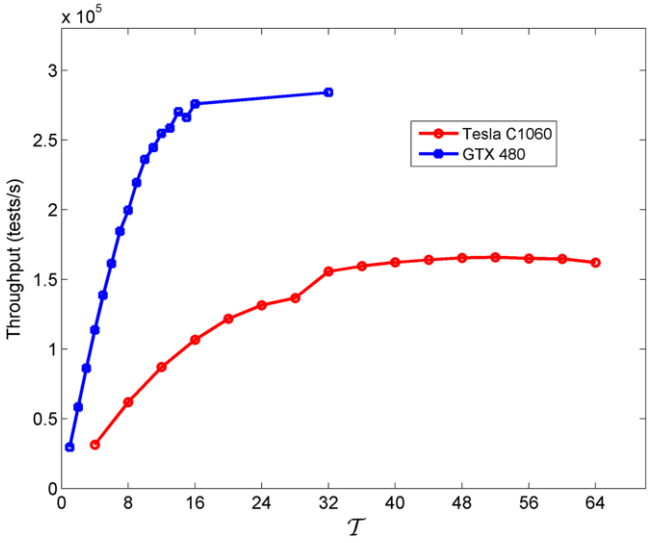


Fig. 1. Throughput of Bloom filter algorithm for set membership testing on biosequence data. Performance (in membership tests per second) is plotted vs. number of threads per processor both for a Tesla C1060 and a GTX 480 GPU.

In this work, we propose the *Threaded Many-core Memory (TMM)* model that captures the performance characteristics of these highly-threaded, many-core machines. This 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, it can be used in any system which has fast context switching and enough threads to hide memory latency. Typical examples of such machines include both NVIDIA and AMD/ATI GPUs and the YarcData uRiKA system. 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.

If the latency of transfers from slow memory to fast memory is small, or if the number of threads per processor is infinite, then this model generally provides the same analysis results as the PRAM analysis. It, however, provides more intuition. (1) Ideally, we want to get the PRAM performance for algorithms using the fewest

number of threads possible, since threads do have overhead. This model can help us pick such algorithms. (2) It also captures the reality of when memory latency is large and the number of threads is large but finite. In particular, it can distinguish between algorithms that have the same PRAM analysis, but one may be better at hiding latency than another with a bounded number of threads.

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 either the machine or the algorithm. We also assume that the hardware provides 0-cost and 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 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.

We analyze 4 classic algorithms for the problem of computing All Pairs Shortest Paths (APSP) on a weighted graph in the TMM model [43]. We compare the analysis from this model with the PRAM analysis of these 4 algorithms to gain intuition about the usefulness of both our model and the PRAM model for analyzing performance of algorithms on highly-threaded, many-core machines. Our results validate the intuition that this model can provide more information than the PRAM model for the large latency, finite thread case. In particular, we compare these algorithms and find specific relationships between hardware parameters (latency, fast memory size, limits on number of threads) under which some algorithms are better than others even if they have the same PRAM cost.

Following the formal analysis, we assess the utility of the model by comparing empirically measured performance on an individual machine to that predicted by the model. For two of the APSP algorithms, we illustrate the impact of various individual parameters on performance, showing the effectiveness of the model at predicting measured performance.

This paper is organized as follows. Section 2 presents related work. Section 3 describes the TMM model. Section 4 provides the 4 shortest paths algorithms and their analysis in both the PRAM and TMM models. Section 5 provides the lessons learned from this model; in particular, we see that algorithms that have the same PRAM performance have different performance in the TMM model since they are better at hiding memory latency with fewer threads. Section 6 continues the discussion of lessons learned, concentrating on the effects of problem size. Section 7 shows performance measurements for a pair of the APSP algorithms executing on a commercial GPU, illustrating correspondence between model predictions and empirical measurements. Finally, Section 8 concludes.

2. Related work

In this section, we briefly review the related work. We first review the work on abstract models of computations for both sequential and parallel machines. We then review recent work on algorithms and performance analysis of GPUs which are the most common current instantiations of highly-threaded, many-core machines.

Many machine and memory models have been designed for various types of parallel and sequential machines. In an early work, Aggarwal et al. [25] 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) [27] 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 [26] 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 [28,29]. Later, an 'ideal-cache' model was introduced in [23,24] 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.

In the parallel case, although widely used, the PRAM [30] model is unrealistic because it assumes all processors work synchronously and that interprocessor communication is free. Quite different to PRAM, the Bulk-Synchronous Parallel (BSP) model [34] 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. [33] 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. [35] present a two-level memory model and give a realistic treatment of parallel block transfers in parallel machines. But this model assumes that 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. [36] 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. [37] present a multi-core-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. [44] present a concurrent cache-oblivious model. Blelloch et al. [38] also propose a parallel cache-oblivious (PCO) model to account for costs of a wide range of cache hierarchies. Chowdhury et al. [39] 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 [42] 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.

While there has not 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. Liu et al. [19] 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. Govindaraju et al. [45] 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. Ryoo et al. [46] 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 [47]. 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. [17] propose another analytical model to capture 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). Meantime, Baghsorkhi et al. [16] 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 [15] 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, Sim et al. [48] develop a performance analysis framework that consists of an analytical model and profiling tools. The framework does a good job in performance diagnostics on case studies of real codes. Kim et al. [49] also design a tool to estimate GPU memory performance by collecting performance-critical parameters. Parakh et al. [50] present a model to estimate both computation time by precisely counting instructions and memory access time by a method to generate address traces. All of these efforts are mainly focused on the practical calibrated performance models. No attempts have been made to develop an asymptotic theoretical model applicable to a wide range of highly-threaded machines.

3. TMM model

The TMM model is meant to model the asymptotic performance of algorithms on highly-threaded, many-core machines. The model should abstract 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 section, we will describe the important characteristics of these highly-threaded, many-core architectures and our model for analyzing algorithms for these architectures.

3.1. Highly-threaded, many-core 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),[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 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 even most multi-core machines), the TMM model captures the chunk size as one of its parameters.

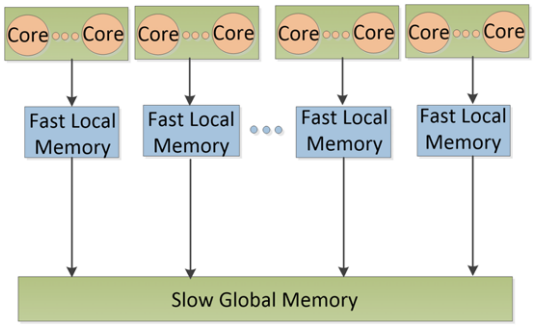


Fig. 2. Abstracted highly-threaded, many-core architecture. The short arrows from the cores to the local memory symbolize low latency, while the long arrows to the global memory symbolize high latency.

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. 2. Note that this architecture abstraction ignores a number of details about the physical machine, including thread grouping, scheduling, etc.

3.2. TMM model parameters

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

In addition to the architecture parameters, we must also consider the parameters which are determined by the algorithm. We assume that the programmer has written a correct synchronous

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

Table 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

Table 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

program and taken care to balance the workload across the core groups. These program parameters are shown in Table 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).

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 the 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 $\mathcal{T} \leq Z/(QS)$.

3.3. TMM model applicability

The TMM model is a high-level abstract model, meant to be applicable to many instantiations of hardware platforms that feature a large number of threads with fast context switching 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.

For NVIDIA GPUs, a number of streaming multiprocessors (core groups in our terminology) share the same global memory. On each of these core groups, 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 core group. 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 [51]. 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. Based on the description from Alverson et al. [52] about the nature of the computations this processor was designed 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-multi-threaded graph processors (named Threadstorm) supporting 128 threads/processor, and highly scalable I/O. Therefore, 128 defines parameter X, the hard limit of number of threads per processor. 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 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 or chunk size C 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. 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.

3.4. 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. The 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

$$T_E = O\left(\max\left(T_1, \frac{M \cdot L}{\mathcal{T}}\right)\right). \quad (1)$$

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

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:

$$T_P = O\left(\max\left(\frac{T_E}{P}, T_\infty\right)\right) = O\left(\max\left(\frac{T_1}{P}, T_\infty, \frac{M \cdot L}{\mathcal{T} \cdot P}\right)\right). \quad (2)$$

Therefore, speedup on P cores, S_P, is

$$S_P = \frac{T_1}{T_P} = \Omega\left(\min\left(P, \frac{T_1}{T_\infty}, \frac{P \cdot T_1 \cdot \mathcal{T}}{M \cdot L}\right)\right). \quad (3)$$

For linear speedup, S_P should be 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) 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.

4. Analysis of all pairs shortest paths algorithms using the TMM model

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. We are 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. In this section, we are interested in asymptotic insights, therefore, we assume that the graphs are large graphs. In particular $n > Z$.

4.1. Dynamic programming via matrix multiplication

Our first algorithm is a dynamic programming algorithm [53] 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:

$$A^2_{ij} = \min_{0 \le k < n} (A^1_{ij}, A^1_{ik} + A^1_{kj}). \quad (4)$$

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 are

$$T_P = O\left(\max\left(\frac{n^3 \lg n}{P}, n \lg n\right)\right) \quad (5)$$

$$S_P = \Omega\left(\min(P, n^2)\right). \quad (6)$$

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

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, sub-blocks of the result matrix (whose size is denoted by B) are assigned to core groups for computation. The threads in a core group read in the required input sub-blocks, perform the computation of Eq. (4) for their assigned sub-block, and write the sub-block out to global memory. This happens $\lg n$ times by repeated squaring.

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 $O(n/B)$ other blocks. However, due to the regularity in memory accesses, each block can be read fully coalesced. Therefore, the number of memory operations for one matrix multiply is $O((n^2/C) \cdot (n/B)) = O(n^3/(BC))$. Also note that since we must fit a $B \times B$ block in a local memory of size Z on one core group, we get $B = \Theta(\sqrt{Z})$. Therefore, for $\lg n$ matrix multiplication operations, $M = O(n^3 \lg n/(\sqrt{Z} \cdot C))$.

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

$$T_P = O\left(\max\left(\frac{T_1}{P}, T_\infty, \frac{M \cdot L}{\mathcal{T} \cdot P}\right)\right) \quad (7)$$

$$= O\left(\max\left(\frac{n^3 \lg n}{P}, n \lg n, \frac{n^3 \lg n \cdot L}{\sqrt{Z} \cdot C \cdot \mathcal{T} \cdot P}\right)\right). \quad (8)$$

Therefore, the speedup on P processors is

$$S_P = T_1/T_P \quad (9)$$

$$= \Omega\left(\min\left(P, n^2, \frac{\sqrt{Z} \cdot C \cdot \mathcal{T}}{L} \cdot P\right)\right). \quad (10)$$

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

4.2. Johnson's algorithm: Dijkstra's algorithm using binary heaps

Johnson's algorithm [54] 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

[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.

algorithm is a greedy algorithm for calculating single source shortest paths. The pseudo-code for Dijkstra's algorithm is given in Algorithm 1 [55]. 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 in the graph.

Algorithm 1 Dijkstra's Algorithm

```
Input: Graph G = (V, E), |V| = n, |E| = m
Input: W is weight of edges, |W| = m
Input: S is source vertex
Output: dist[n]
    {Initialize distance array}
for all u ∈ V do
  dist[u] = ∞
end for
dist[S] = 0
for all u ∈ V do
  Q ← dist[u]
end for
    {Propagate the distance update to all vertices}
while Q not empty do
  u = deletemin(Q)
  for each edge (u, v) ∈ E do
    if dist[v] > dist[u] + W[u, v] then
      dist[v] = dist[u] + W[u, v]
      decreasekey(Q, v)
    end if
  end for
end while
```

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 are

$$T_P = O\left(\max\left(\frac{mn \lg n}{P}, m \lg n\right)\right) \tag{11}$$

$$S_P = \Omega\left(\min(P, n)\right). \tag{12}$$

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

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.

$$T_P = O\left(\max\left(\frac{T_1}{P}, T_\infty, \frac{M \cdot L}{\mathcal{T} \cdot P}\right)\right) \tag{13}$$

$$= O\left(\max\left(\frac{mn \lg n}{P}, m \lg n, \frac{mn \lg n \cdot L}{\mathcal{T} \cdot P}\right)\right). \tag{14}$$

Therefore, the speedup on P processors is

$$S_P = \Omega\left(\min\left(P, n, \frac{\mathcal{T}}{L} \cdot P\right)\right). \tag{15}$$

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.

4.3. Johnson's algorithm: Dijkstra's algorithm using 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.

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

$$T_P = O\left(\max\left(\frac{n^3}{P}, n \lg n + m\right)\right) \tag{16}$$

$$= O\left(\max\left(\frac{n^3}{P}, n \lg n, m\right)\right) \tag{17}$$

$$S_P = \Omega\left(\min\left(P, \frac{n^2}{\lg n}, \frac{n^3}{m}\right)\right). \tag{18}$$

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 ($Q\mathcal{T}$

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

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, $M = O(n^3/C + mn)$.

$$T_P = O\left(\max\left(\frac{T_1}{P}, T_\infty, \frac{M \cdot L}{\mathcal{T} \cdot P}\right)\right) \tag{19}$$

$$= O\left(\max\left(\frac{n^3}{P}, \frac{n^2 \lg Z}{Z}, \frac{\left(\frac{n^3}{C} + mn\right) \cdot L}{\mathcal{T} \cdot P}\right)\right) \tag{20}$$

$$= O\left(\max\left(\frac{n^3}{P}, \frac{n^2 \lg Z}{Z}, \frac{n^3 \cdot L}{C \cdot \mathcal{T} \cdot P}, \frac{mn \cdot L}{\mathcal{T} \cdot P}\right)\right). \tag{21}$$

Speedup is

$$S_P = \Omega\left(\min\left(P, \frac{nZ}{\lg Z}, \frac{C \cdot \mathcal{T}}{L} \cdot P, \frac{n^2 \cdot \mathcal{T}}{m \cdot L} \cdot P\right)\right). \tag{22}$$

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

4.4. 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 [56,57].

Algorithm 2 Bellman–Ford

```
Input: Graph G = (V, E), |V| = n, |E| = m
Input: W is weight of edges, |W| = m
Input: S is source vertex
Output: dist[n]
    {Initialize distance array}
for all u in V do
  dist[u] = ∞
end for
dist[S] = 0
    {Update the distance for all vertices n − 1 times}
for i = 1 : (n − 1) do
  for each edge e(u, v) ∈ E do
    if dist[v] > dist[u] + W[u, v] then
      dist[v] = dist[u] + W[u, v]
    end if
  end for
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 $O(n)$.

$$T_P = O\left(\max\left(\frac{mn^2}{P}, n\right)\right). \tag{23}$$

$$S_P = \Omega\left(\min(P, mn)\right). \tag{24}$$

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.

$$T_P = O\left(\max\left(\frac{T_1}{P}, T_\infty, \frac{M \cdot L}{\mathcal{T} \cdot P}\right)\right) \tag{25}$$

$$= O\left(\max\left(\frac{mn^2}{P}, n, \frac{mn^2 \cdot L}{C \cdot \mathcal{T} \cdot P}\right)\right). \tag{26}$$

Therefore, the speedup on P processors is

$$S_P = \Omega\left(\min\left(P, mn, \frac{C \cdot \mathcal{T}}{L} \cdot P\right)\right). \tag{27}$$

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 $L \leq \min(CX, CZ/Q)$.

5. Comparison of the 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 3 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,

Table 3
Algorithm running times and constraints.

Algorithm	Time (PRAM)	Time (TMM)	Constraints	
Dynamic programming	$\frac{n^3 \lg n}{P}$	$\frac{n^3 \lg n \cdot L}{\sqrt{Z}C\mathcal{T}P}$	$L \leq \sqrt{Z}CX$	$L \leq Z^{3/2}C/Q$
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)	$\frac{n^3}{P}$	$\frac{n^3 L}{C\mathcal{T}P}, \frac{n^2}{m} \geq C$	$L \leq CX$	$L \leq Z/Q \cdot C$
		$\frac{mnL}{\mathcal{T}P}, \frac{n^2}{m} < C$	$L \leq n^2X/m$	$L \leq n^2Z/(mQ)$
n iteration Bellman–Ford	$\frac{n^2 m}{P}$	$\frac{mn^2 L}{C\mathcal{T}P}$	$L \leq CX$	$L \leq CZ/Q$

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 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.

5.1. 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:

5.1.1. 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 $O(n^3 \lg n/P)$.

5.1.2. 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 $O(n^3/P)$.

5.2. 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:

$$\frac{n^3 \lg n \cdot L}{\sqrt{Z}C\mathcal{T}P} = \frac{n^2 L}{\mathcal{T}P} \cdot \frac{n \lg n}{\sqrt{Z}C} < \frac{n^2 L}{\mathcal{T}P} \cdot \frac{n}{C}. \tag{28}$$

While Johnson's algorithm has runtime:

$$\min\left(\frac{n^3 L}{C\mathcal{T}P}, \frac{mnL}{\mathcal{T}P}\right) = \frac{n^2 L}{\mathcal{T}P} \cdot \min\left(\frac{n}{C}, \frac{m}{n}\right). \tag{29}$$

If $n^2/m < C$, i.e. dense graphs, $n/C < m/n$. Combine (28) and (29), we have

$$\frac{n^3 \lg n \cdot L}{\sqrt{Z}C\mathcal{T}P} < \frac{n^3 L}{C\mathcal{T}P}, \quad \text{if } \frac{n^2}{m} < C. \tag{30}$$

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 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.

6. Effect of problem size

In Section 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.

6.1. Vertices fit in local memory

When $n < Z$, all the vertices fit in local memory. Note that this does not 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. Now 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.

6.2. 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 $M = O(n^3/C)$. And the run time under the TMM model reduces to $O(n^3L/(C\mathcal{T}P))$. 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 the better one.

7. Empirical investigation

In this section, we conduct experiments to understand the extent of the applicability of our model in explaining the performance of algorithms on a real machine. This evaluation is a proof-of-concept that the model successfully predicts performance on one example of a 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 an NVIDIA GPU.

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.
- *Comparison of the dynamic programming algorithm and Johnson's algorithm with arrays*: 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.

7.1. Experimental Setup

The experiments are carried out on an NVIDIA GTX 480, which has 15 multiprocessors, each with 32 cores. As a typical highly-threaded, many-core machine, it also features a 1.5 GB 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.

7.2. 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 $\{1k, 2k, 4k, 8k, 16k\}$ vertices. To better utilize fast local memory, the problem is decomposed into sub-blocks, and we must also pick a block size. Since we only care about the effect of

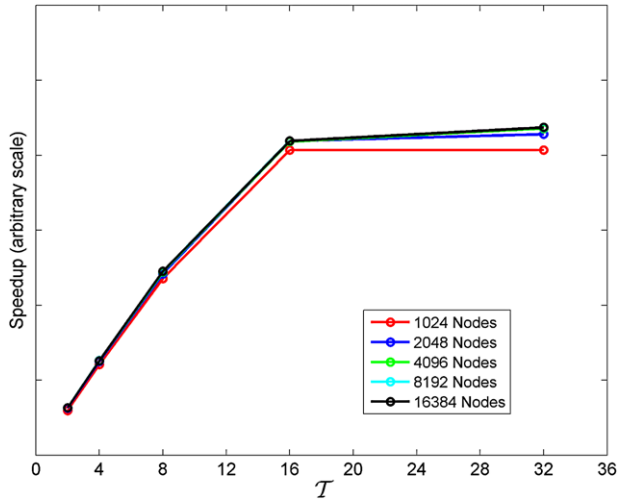


Fig. 3. Speedup (theoretical T_1 via PRAM model over empirically measured T_P) of the dynamic programming algorithm, varying the number of threads per core from 2 to 32 (data block size = 64).

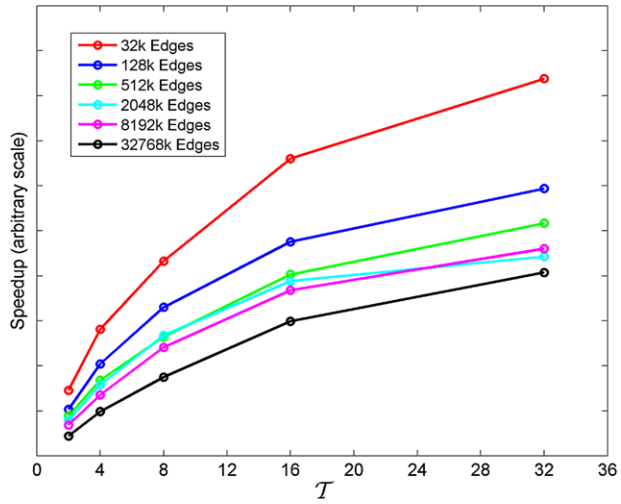


Fig. 4. Speedup of Johnson's algorithm using arrays vs. threads/core for different graph densities. All curves are with 8k nodes. Again, speedup is theoretical T_1 divided by empirically measured T_P.

threads and not the effect of shared memory (to be considered in the next subsection), here we show the results with a block size of 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. Fig. 3 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. (10)) and then flattens out. This indicates that for this experiment, 16 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. (10)).

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 32 M). The speedup graph is shown in Fig. 4. As we increase the number of threads, the speedup increases. We see two other interesting things, however. First, we never see the flattening

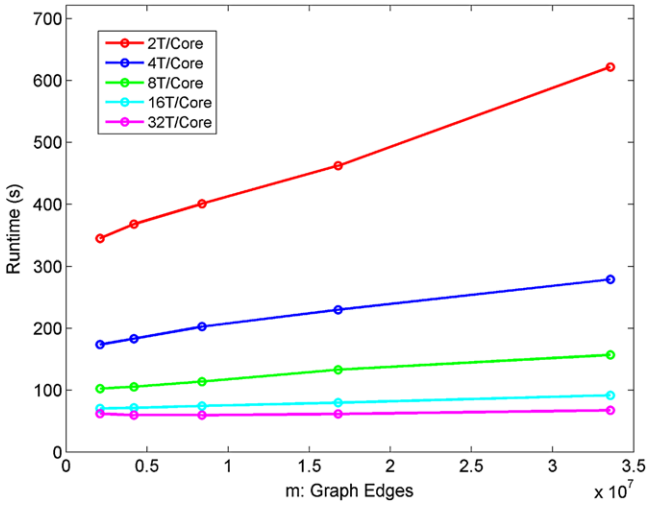


Fig. 5. Runtime of Johnson's algorithm on graphs with constant 8k nodes and varying density by increasing edges. Threads/core varies from 2 to 32.

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. 5. Here, the runtime is plotted vs. number of graph edges for varying threads/core. The linear relationship predicted by the last term of Eq. (21) (for dense graphs) is illustrated clearly in the figure.

7.3. 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. 6 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 size B can be determined. Then, varying thread counts has the same effect as previously illustrated in Fig. 3, increasing threads/core increases performance until the PRAM range is reached. But as we can see from the figure, different block sizes have different performance for the same number of threads/core. This effect is predicted by Eq. (10). As we increase the size of local memory, the performance improves, since we can use bigger blocks.

In order to isolate the effect of block size from the effects of other parameters, we also plot this data in a pair of different formats in Figs. 7 and 8, in both cases limiting the number of threads/core to below the PRAM range (i.e., the range where speedup is linear in threads/core). The first curve shows the

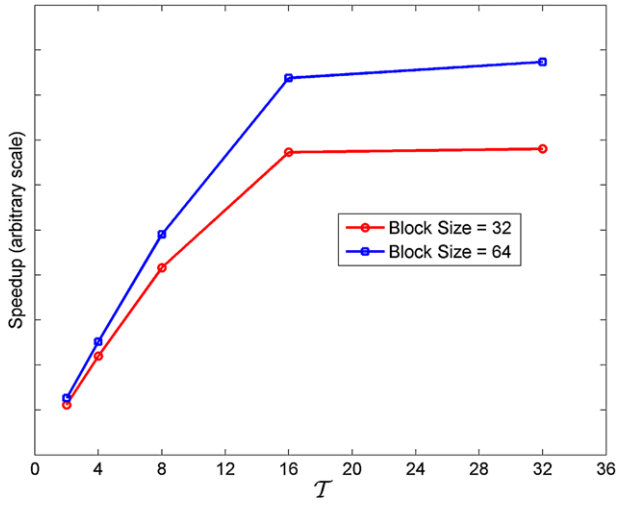


Fig. 6. Speedup of the dynamic programming algorithm for different block sizes, varying the threads/core on graphs with 16k nodes.

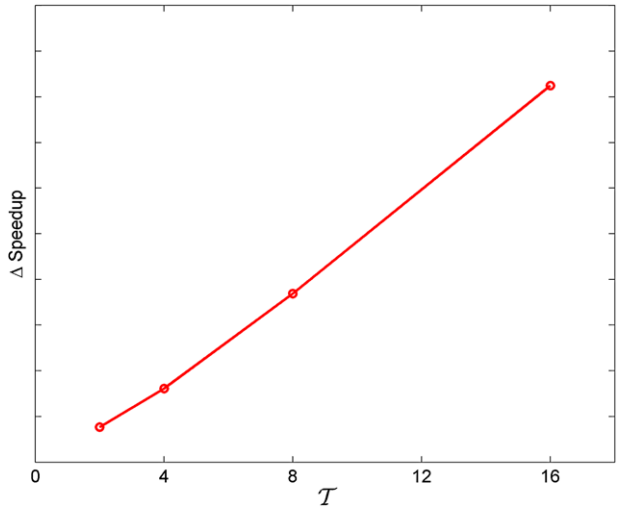


Fig. 7. Spread of performance between block size 64 and block size 32. The speedup scale is the same as that of Fig. 6.

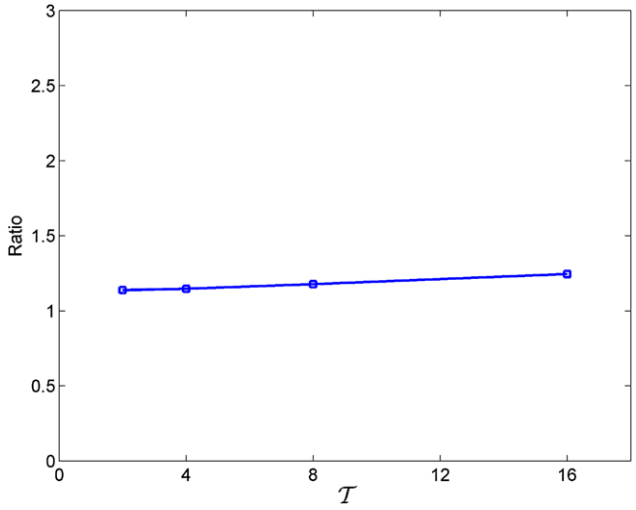


Fig. 8. Ratio of performance between block size 64 and block size 32.

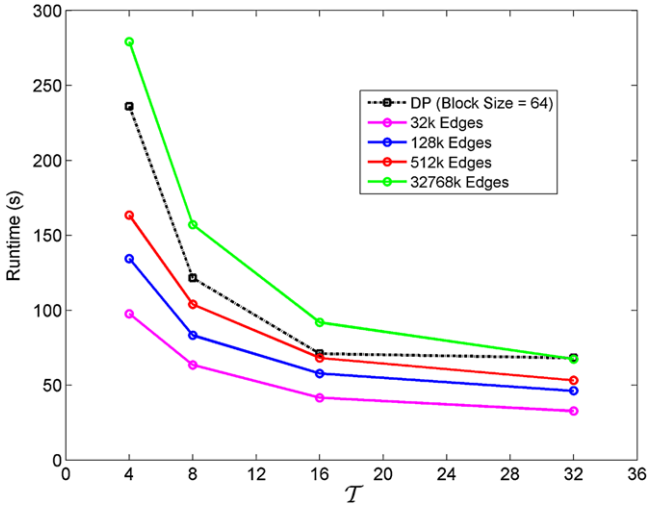


Fig. 9. Runtime of the dynamic programming (DP) algorithm relative to Johnson's algorithm on a graph with 8k nodes, varying threads/core from 4 to 32 and edges from 32k to 32 M.

difference between the speedups for different block sizes. As the curve indicates, the delta speedup increases linearly with the number of threads/core, consistent with the model prediction of $(B_1 - B_2)\mathcal{T}$. The second curve shows the ratio of the performance of block size 64 to block size 32, indicating a flat line, since the thread term cancels out.

7.4. Comparison between the dynamic programming and Johnson's algorithms

It is interesting to compare 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 Section 5.2, 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. 9 demonstrates this effect concretely.

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.

8. Conclusions

In this paper, we present a memory access model, called the Threaded Many-core Memory (TMM) model, that is well suited for modern highly-threaded, many-core systems that employ many threads and fast context switching to hide memory latency. The model analyzes the significant factors that affect performance on many-core machines. In particular, it requires the work and depth (like PRAM algorithms), but also requires the analysis of the number of memory accesses. Using these three values, we can properly order algorithms from slow to fast for many different settings of machine parameters on highly-threaded, many-core machines. We analyzed 4 shortest paths algorithms in the TMM model and compared the analysis with the PRAM analysis. We find that algorithms with the same PRAM performance can have

different TMM performance under certain machine parameter settings. In addition, for certain problem sizes which fit in local memory, algorithms which are faster on PRAM may be slower under the TMM model. Further, we implemented a pair of the algorithms and showed empirical performance is effectively predicted by the TMM model under a variety of circumstances. Therefore, TMM is a model well-suited to compare algorithms and decide which one to implement under particular environments. To our knowledge, this is the first attempt to formalize the analysis of algorithms for highly-threaded, many-core computers using a formal model and asymptotic analysis.

There are many directions of future work. One obvious direction is to design more algorithms under the TMM model. Ideally, this model can help us come up with new algorithms for highly-threaded, many-core machines. Empirical validation of the TMM model across a wider number of physical machines and manufacturers is also worth doing. In addition, our current model only incorporates 2 levels of memory hierarchy. While in this paper we assume that it is global memory vs. memory local to core groups, in principle, it can be any two levels of fast and slow memory. We would like to extend it to multi-level hierarchies which are becoming increasingly common. One way to do this is to design a "parameter-oblivious" model where algorithms do not know the machine parameters. Other than the dynamic programming algorithm, all of the algorithms presented in this paper are, in fact, parameter-oblivious. And matrix multiplication in the dynamic programming can easily be made parameter-oblivious. In this case, the algorithms should perform well under all settings of parameters, allowing us to apply the model at any two levels and get the same results.

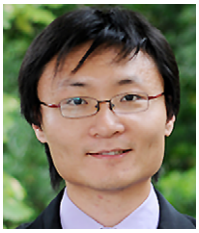
Acknowledgments

This work was supported by NSF grants CNS-0905368 and CNS-0931693 and Exegy, Inc.

References

[1] D.A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J.D. Owens, N. Amenta, Real-time parallel hashing on the GPU, ACM Trans. Graph. 28 (5) (2009) 154:1–154:9.
[2] N.K. Govindaraju, S. Larsen, J. Gray, D. Manocha, A memory model for scientific algorithms on graphics processors, in: Proc. of ACM/IEEE Conf. on Supercomputing, 2006.
[3] L. Ma, R.D. Chamberlain, J.D. Buhler, M.A. Franklin, Bloom filter performance on graphics engines, in: Proc. of Int'l Conf. on Parallel Processing, 2011, pp. 522–531.
[4] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore GPUs, in: Proc. of IEEE Int'l Symp. on Parallel and Distributed Processing, 2009.
[5] J.W. Choi, A. Singh, R.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.
[6] V. Volkov, J.W. Demmel, Benchmarking GPUs to tune dense linear algebra, in: Proc. of ACM/IEEE Conf. on Supercomputing, 2008.
[7] Y. Zhang, J. Cohen, J.D. Owens, Fast tridiagonal solvers on the GPU, in: Proc. of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2010.
[8] W. Liu, B. Schmidt, G. Voss, W. Muller-Wittig, Streaming algorithms for biological sequence alignment on GPUs, IEEE Trans. Parallel Distrib. Syst. (2007) 1270–1281.
[9] Y. Liu, B. Schmidt, D.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 (2010) 93.
[10] S. Hong, S.K. Kim, T. Oguntebi, K. Olukotun, Accelerating CUDA graph algorithms at maximum warp, in: Proc. of 16th ACM Symp. on Principles and Practice of Parallel Programming, 2011.
[11] G.J. Katz, J.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.
[12] K. Matsumoto, N. Nakasato, S.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, 2011, pp. 145–152.
[13] D. Merrill, M. Garland, A. Grimshaw, Scalable GPU graph traversal, in: Proc. of 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, 2012, pp. 117–128.
[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA, J. Parallel Distrib. Comput. 68 (10) (2008) 1370–1380.
[15] Y. Zhang, J. Owens, A quantitative performance analysis model for GPU architectures, in: Proc. of IEEE Int'l Symp. on High Performance Computer Architecture, 2011, pp. 382–393.
[16] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, W.-M. Hwu, An adaptive performance modeling tool for GPU architectures, in: Proc. of 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, 2010, pp. 105–114.
[17] S. Hong, H. 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, 2009, pp. 152–163.
[18] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. 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, 2010, pp. 451–460.
[19] W. Liu, W. Muller-Wittig, B. Schmidt, Performance predictions for general-purpose computation on GPUs, in: Proc. of Int'l Conf. on Parallel Processing, 2007.
[20] L. Ma, R.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, 2012.
[21] A.V. Aho, J.E. Hopcroft, The Design and Analysis of Computer Algorithms, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.
[22] A. Aggarwal, J. Vitter, The input/output complexity of sorting and related problems, Commun. ACM 31 (9) (1988) 1116–1127.
[23] M. Frigo, C.E. Leiserson, H. Prokop, S. Ramachandran, Cache-oblivious algorithms, in: Proc. of 40th Symposium on Foundations of Computer Science, 1999, pp. 285–297.
[24] H. Prokop, Cache-oblivious algorithms, Master's Thesis, MIT, 1999.
[25] A. Aggarwal, B. Alpern, A. Chandra, M. Snir, A model for hierarchical memory, in: Proc. of 19th ACM Symposium on Theory of Computing, 1987, pp. 305–314.
[26] B. Alpern, L. Carter, T. Selker, Visualizing computer memory architectures, in: Proc. of 1st Conf. on Visualization, 1990, pp. 107–113.
[27] A. Aggarwal, A.K. Chandra, M. Snir, Hierarchical memory with block transfer, in: Proc. of 28th Symposium on Foundations of Computer Science, 1987, pp. 204–216.
[28] B. Alpern, L. Carter, E. Feig, T. Selker, The uniform memory hierarchy model of computation, Algorithmica 12 (2/3) (1994) 72–109.
[29] J.S. Vitter, M.H. Nodine, Large-scale sorting in uniform memory hierarchies, J. Parallel Distrib. Comput. 17 (1–2) (1993) 107–114.
[30] S. Fortune, J. Wyllie, Parallelism in random access machines, in: Proc. of 10th ACM Symp. on Theory of computing, 1978.
[31] R.M. Karp, A survey of parallel algorithms for shared-memory machines, Tech. Rep., University of California at Berkeley, Berkeley, CA, USA, 1988.
[32] U. Vishkin, G.C. Caragea, B. 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.
[33] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. 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.
[34] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[35] J.S. Vitter, E.A.M. Shriver, Algorithms for parallel memory I: two-level memories, Algorithmica 12 (1994) 110–147.
[36] L. Arge, M.T. Goodrich, M. Nelson, N. Sitchinava, Fundamental parallel algorithms for private-cache chip multiprocessors, in: Proc. of 20th Symp. on Parallelism in Algorithms and Architectures, 2008, pp. 197–206.
[37] G.E. Blelloch, R.A. Chowdhury, P.B. Gibbons, V. Ramachandran, S. Chen, M. Kozuch, Provably good multicore cache performance for divide-and-conquer algorithms, in: Proc. of 19th ACM–SIAM Symp. Discrete Algorithms, 2008, pp. 501–510.
[38] G.E. Blelloch, J.T. Fineman, P.B. Gibbons, H.V. Simhadri, Scheduling irregular parallel computations on hierarchical caches, in: Proc. of 23rd ACM Symp. on Parallelism in Algorithms and Architectures, 2011, pp. 355–366.
[39] R.A. Chowdhury, F. Silvestri, B. Blakeley, V. Ramachandran, Oblivious algorithms for multicores and network of processors, in: Proc. of 24th IEEE Int'l Parallel and Distributed Processing Symp., 2010, pp. 1–12.
[40] R. Cole, V. Ramachandran, Efficient resource oblivious algorithms for multicores, CoRR abs/1103.4071, 2011, v1.
[41] R.A. Chowdhury, V. Ramachandran, The cache-oblivious Gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation, in: Proc. of 19th ACM Symp. on Parallel Algorithms and Architectures, 2007, pp. 71–80.
[42] R.A. Chowdhury, V. Ramachandran, Cache-efficient dynamic programming algorithms for multicores, in: Proc. of 20th Symp. on Parallelism in Algorithms and Architectures, 2008, pp. 207–216.

[43] L. Ma, K. Agrawal, R. 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, 2012, pp. 339–347.
[44] M.A. Bender, J.T. Fineman, S. Gilbert, B.C. Kuszmaul, Concurrent cache-oblivious *b*-trees, in: Proc. of 17th ACM Symposium on Parallelism in Algorithms and Architectures, 2005, pp. 228–237.
[45] N.K. Govindaraju, S. Larsen, J. Gray, D. Manocha, A memory model for scientific algorithms on graphics processors, in: Proc. of ACM/IEEE Supercomputing Conf., 2006.
[46] S. Ryoo, C.I. Rodrigues, S.S. Stone, S.S. Baghsorkhi, S.-Z. Ueng, J.A. Stratton, W.-M. Hwu, Program optimization space pruning for a multithreaded GPU, in: Proc. of 6th IEEE/ACM Int'l Symp. on Code Generation and Optimization, 2008, pp. 195–204.
[47] K. Kothapalli, R. Mukherjee, M.S. Rehman, S. Patidar, P.J. Narayanan, K. Srinathan, A performance prediction model for the CUDA GPGPU platform, 2009, pp. 463–472.
[48] J. Sim, A. Dasgupta, H. Kim, R. Vuduc, A performance analysis framework for identifying potential benefits in GPGPU applications, in: Proc. of 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2012, pp. 11–22.
[49] Y. Kim, A. Shrivastava, CuMAPz: a tool to analyze memory access patterns in CUDA, in: Proc. of 48th Design Automation Conference, 2011, pp. 128–133.
[50] A. Parakh, M. Balakrishnan, K. Paul, Performance estimation of GPUs with cache, in: Parallel and Distributed Processing Symposium Workshops Ph.D. Forum, IPDPSW, 2012, pp. 2384–2393.
[51] NVIDIA, Cuda Programming Guide 5.0.
[52] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith, The Tera computer system, in: Proc. of 4th International Conference on Supercomputing, ICS'90, ACM, New York, NY, USA, 1990, pp. 1–6.
[53] T.H. Cormen, C. Stein, R.L. Rivest, C.E. Leiserson, Introduction to Algorithms, second ed., McGraw-Hill Higher Education, 2001.
[54] D.B. Johnson, Efficient algorithms for shortest paths in sparse networks, J. ACM 24 (1) (1977) 1–13.
[55] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1) (1959) 269–271.
[56] R. Bellman, On a routing problem, Quart. Appl. Math. 16 (1958) 87–90.
[57] J. Lestor, R. Ford, D.R. Fulkerson, Flows in Networks, Princeton University Press, Princeton, NJ, USA, 1962.



Lin Ma is a Ph.D. Candidate of Computer Science and Engineering at Washington Univ. in St. Louis. His research interest includes parallel architecture and algorithms, performance evaluation and tuning model for multi-threading system, accelerating application-specific architectures, and high-performance streaming computing over architecturally diverse system of CPUs and GPUs.

Kunal Agrawal is an Assistant Professor of Computer Science and Engineering at Washington Univ. in St. Louis, where she came after completing her Ph.D. under Charles Leiserson at MIT. Dr. Agrawal is interested in many aspects of parallel computing and works primarily on the design of provably good runtime systems for parallel programming environments. She has worked on various topics such as scheduling, resource allocation, transactional memory, cache-aware, and cache-oblivious streaming.

Roger D. Chamberlain is a Professor of Computer Science and Engineering at Washington Univ. in St. Louis, where he serves as Associate Department Chair. Dr. Chamberlain's research interests include specialized computer architectures for a variety of applications (e.g., astrophysics and biology), high-performance parallel and distributed application development, energy-efficient computation, and high-capacity I/O systems. He teaches in the areas of digital systems, parallel processing, computer architecture, embedded systems, and reconfigurable logic.