A Memory Access Model for Highly-threaded Many-core Architectures

Lin Ma, Kunal Agrawal, and Roger D. Chamberlain
Department of Computer Science and Engineering, Washington University in St. Louis
{lin.ma, kunal, roger}@seas.wustl.edu

***Abstract*—Many-core architectures are excellent in hiding memory-access latency by low-overhead context switching among a large number of threads. The speedup of algorithms carried out on these machines depends on how well the latency is hidden. If the number of threads were infinite, then theoretically these machines should provide the performance predicted by the *PRAM* analysis of the programs. However, the number of allowable threads per processor is not infinite. 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 give more fine-grained 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 Floyd-Warshall algorithm and Johnson's algorithms 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 Floyd-Warshall algorithm performs better on these machines.**

***Keywords*-PRAM, *TMM*, All-pairs Shortest Paths (APSP), Many-core**

I. 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 distinctive features of these devices include (1) a large number of hardware threads with low-overhead context switching between them, (2) explicitly managed memory hierarchies, and (3) high-latency-high-bandwidth data transfer between fast and slow memories. Researchers have designed algorithms to solve many interesting problems for these devices, such as GPU sorting or hashing [5], [24], [36], [40], linear algebra [14], [45], [47], dynamic programming [33], [34], graph algorithms [25], [29], [37], [38], and many other classic algorithms [13], [48]. These projects generally report impressive gains in performance. These devices appear to be here to stay. We are interested in analyzing and characterizing performance of algorithms on these highly threaded many-core machines in a more abstract, algorithmic, and systematic manner. While there is a lot of folk wisdom on how to design good algorithms for GPUs in addition to a significant body of work on performance analysis [9], [26], [30], [32], [35], there are no systematic theoretical models to analyze the performance of programs on these machines.

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*) [4] model, 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 spacial and temporal locality on these machines can lead to better performance. Other models that consider the memory access costs of sequential algorithms include the cache-oblivious model [23], [39], Hierarchical Memory Model (*HMM*) [1], Memory Hierarchy (*MH*) model [7], Block Transfer model (*BT*) [2], and Uniform Memory Hierarchy (*UMH*) model [6], [43].

For parallel computing, the analogue for *RAM* model is the Parallel Random Access Machine (*PRAM*) model [22] and there is a large body of work describing and analyzing algorithms in the *PRAM* model [28], [42]. 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, however, and assumes that each memory access by the algorithm takes unit time. For modern machines, however, this assumption seldom holds. Therefore, researcher have designed various models for distributed memory machines [19], [41], [44], shared memory machines and multicores [8], [11], [12], [15], [18] or the combination of the two [16], [17].

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 the traditional multicore or multiprocessor shared memory machines. The high-level characteristics that we will focus on are: (1) These many-core machines have a

large number of threads and a super fast context switching mechanism. Therefore, if a thread stalls on a memory operation, some other thread may be scheduled in its place. (2) Explicitly managed memory hierarchy (instead of hardware managed caches) which allows programs to place data at a particular level. (3) Automatic coalescing of memory accesses, where if multiple threads access data from a slower memory in a predictable pattern, this data can be fetched with just one memory access instead of many. These aspects of the GPU-style many-core machines make their algorithm design parameters very different from those used in multicore algorithm design. In the multicore models in the literature, researchers count the number of memory transfers from slow memory to fast memory, and algorithms are designed to minimize these, since memory transfers take a significant amount of time. Since, nominally, only one thread is running on one processor, this thread blocks on the memory transfer. Since many-cores are explicitly designed to hide memory latency via thread switching, 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.

In this work, we investigate this intuition. In particular, we propose the *Threaded Many-core Memory (TMM)* model that captures the performance characteristics of these 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 GPU-like many-core machines with SIMD computations, in principle, it can be used in any system which has fast context switching and enough threads to hide memory latency. So it would also apply to multicore machines which implement fast context-switches. If the latency of transfer 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 algorithm 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 large number of machines which allow a large number of threads with fast context switching. Therefore, it abstracts away implementation details of either the machine or the algorithm, since it is meant to be general and applicable to many machines which are in the similar paradigm. 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 in one multiprocessor. 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 this model. 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 GPU-style, many-core machines. Our results validate the intuition that this model can provide more information than the *PRAM* model for 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.

This paper is organized as follows. Section II describes the *TMM* model. Section III provides the 4 shortest paths algorithms and their analysis in both the *PRAM* and the *TMM* model. Section IV 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. Finally, Section VI provides the conclusions.

II. Modeling

The *TMM* model is meant to model the important characteristics of GPU-style, many-core architectures while abstracting away the details. In this section, we will describe the important characteristics of these many-core architectures and our model for analyzing algorithms for these architectures.

A. Many-core Architectures

Many-core architectures typically consist of a number of multiprocessors, each containing a number of processors (or cores),[1] a fixed number of registers, and a fixed size of local on-chip shared fast memory. A large global memory is shared by all the multiprocessors. Registers are the fastest to access, the shared local on-chip memory is slower than registers but faster than the global memory. Accessing the global memory may potentially take 100s of cycles.

These architectures support a large number of hardware threads, much larger than the number of cores. Cores on a single multiprocessor execute in SIMD style where groups of threads execute in lock-step. When a thread group executing on a multiprocessor stalls on a slow memory access, in theory, a context switch occurs and another thread group is scheduled on that multiprocessor. When servicing this memory stall, the memory subsystem will coalesce memory accesses that are within certain address range bounds (i.e., they are sufficiently close together). The architecture is abstracted in Figure 1. Note that this architecture abstraction ignores a number of details about the physical machine, including warps, half-warps, blocks, and block scheduling.

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

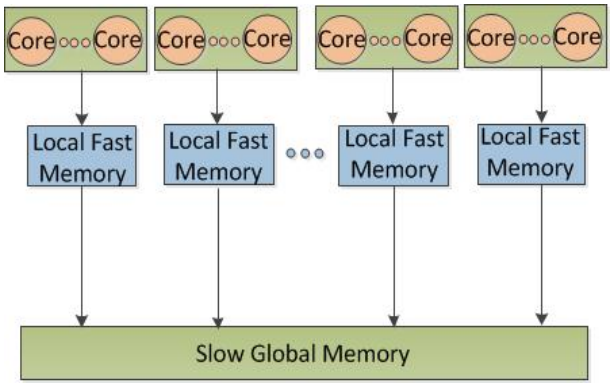

Fig. 1. Abstracted many-core architecture.

B. TMM Model Parameters

The *TMM* model captures the important characteristics of a many-core architecture by using six parameters shown in Table I. L is the latency for accessing the slow memory (in our case, the global memory which is shared by all the multiprocessors). P is the total number of processors (or cores) in the machine. C is the maximum number of memory accesses that can be coalesced while accessing global memory. The parameter Z represents the size of local fast memory per multiprocessor and Q represents the total number of cores per multiprocessor. Note that we do not have a parameter for the number of cores per multiprocessor, 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 I
ARCHITECTURE PARAMETERS.

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

In addition to the architecture parameters, we must also consider parameters which are decided by the algorithm. We assume that the programmer has written a proper SIMD style program and taken care to balance the workload across the multiprocessors. The parameters decided by the program are shown in Table II. T_1 represents the work of the algorithm, that is, the total number of operations that the program must perform. 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 the 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. If multiple accesses can be coalesced, then they will only count as 1 operation when accounting for M. $\mathcal{T}$ is the total 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 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 total number of threads $\mathcal{T} \leq T_1/(T_\infty \cdot P)$. 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 total number of threads per core is at most $\mathcal{T} \leq Z/(SQ)$.

TABLE II
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

C. TMM Analysis structure

In order to analyze a program performance in *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

$$T_E = \max(T_1, \frac{M \cdot L}{\mathcal{T}}) \tag{1}$$

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:

$$T_P = T_E/P = \max(T_1/P, T_\infty, \frac{M \cdot L}{\mathcal{T} \cdot P}) \tag{2}$$

Therefore, speedup on P cores, S_P, is

$$S_P = T_1/T_P = \min(P, T_1/T_\infty, \frac{P \cdot T_1 \cdot \mathcal{T}}{M \cdot L}) \tag{3}$$

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 equation (3) dominate, then a

many-core algorithm is the same as the corresponding *PRAM* algorithm. On the other hand, if the last term dominates, then the many-core 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 many-core machines. Therefore, this model can help us pick algorithms that provide performance as close as possible to *PRAM* algorithms.

III. Analysis of All Pairs Shortest Paths Algorithms using *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 $n > Z$.

A. Floyd-Warshall Algorithm: Dynamic Programming via Matrix Multiplication

Our first algorithm is the Floyd-Warshall algorithm [46], [21], which is a dynamic programming algorithm 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:

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

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 $T_1 = O(n^3 \lg n)$.[2] The span of a single matrix multiplication algorithm is $O(n)$. Therefore, the total span of the algorithm is $T_\infty = O(n \lg n)$.

[2]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 time and speedup using P processors is

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

$$S_P = O\left(\min(P, n^2)\right) \tag{6}$$

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

TMM Algorithm and Analysis: TMM algorithms are tailored to 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 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 multiprocessors for computation. The threads in a multiprocessor read in the required input sub-blocks, perform the computation of equation (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 accesses. 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 n/B other blocks. However, due to the regularity in memory accesses, each block can be read fully coalesced. Therefore, the number of memory accesses 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 multiprocessor, 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.

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

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

Therefore, the speedup on P processors is

$$S_P = T_1/T_P \tag{9}$$

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

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 $\mathcal{T} \le \min(X, Z/(SQ))$, and $S = O(1)$, since each thread only needs constant memory. Therefore, we can conclude that the algorithm achieves linear speedup as long as $L \le \min(\sqrt{Z}CX, Z^{3/2}C/Q)$.

B. Johnson's Algorithm: Dijkstra's Algorithm using Binary Heaps

Johnson's algorithm [27] is an all-pairs shortest paths algorithm that uses Dijkstra's single source algorithm as the

subroutine and calls it n times 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 [20]. The single source algorithm consists of n insert operations, m decrease-key operations and n delete-min operations. 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

```
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
```

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

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

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

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 to 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 accesses is $M = O(mn \lg n)$.[3]

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

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

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

Therefore, the speedup on P processors is

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

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 $\mathcal{T} \leq \min(X, Z/(SQ))$, 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 previous algorithm.

C. Johnson's Algorithm: Dijkstra's Algorithm using an Array

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 reduce 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.

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

The time and speedup using P processors is

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

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

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

TMM Algorithm and Analysis: The *TMM* algorithm is similar to the *PRAM* algorithm, except that each multiprocessor is responsible for a single-source shortest path calculation. Therefore, all the threads on a single multiprocessor ($Q\mathcal{T}$ 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 size-Z chunks. 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, $M = O(n^3/C + mn)$.

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

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

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

Speedup is

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

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

D. 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 [31], [10].

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 $O(n)$.

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 ∈ (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
```

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

$$S_P = O\left(\min(P, mn)\right). \quad (24)$$

TMM Algorithm and Analysis: The *TMM* algorithm for this problem is more complicated and requires more data structure support. Each multiprocessor 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 is responsible for 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.

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

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

Therefore, the speedup on P processors is

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

In this case, we get linear speedup if $C\mathcal{T}/L \geq 1$. Subject to the limits on threads of $\mathcal{T} \leq \min(X, Z/(SQ))$ and $S = O(1)$

for constant local memory usage per thread, this requires $L \leq \min(CX, CZ/Q)$.

IV. Comparison of the various algorithms

As our analysis of shortest paths algorithms indicates, the *TMM* model allows us to take the unique properties of many-core architectures into consideration while analyzing the algorithms. Therefore, the model provides more nuance to GPU algorithms 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 III 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 many-core machines provide *PRAM* performance. However, the cut-off value for L is different for different algorithms. Therefore, the *TMM* model can be informative for comparison purposes between algorithms.

A. 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 many-core machines may vary significantly. Let us consider a few examples:

1) Floyd-Warshall vs. Johnson's Algorithm with 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 $O(n^3 \lg n/P)$.

2) Johnson's Algorithm with Binary Heap vs. Johnson's Algorithm using an Array 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 $O(n^3/P)$.

B. 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 size of the graph. In particular for certain sizes of graphs, some algorithms are better than others even if they are asymptotically worse.

Consider the example of Floyd-Warshall vs. Johnson's Algorithm using Arrays. In the *PRAM* model, the Floyd-Warshall algorithm is unquestionably worse than Johnson's algorithm. However, if L is large, say $O(Z^{3/2}C/Q)$, then Johnson's algorithm has a running time of $O(n^3\sqrt{Z}/P)$, while the dynamic programming algorithm has a running time of $O(n^3 \lg n/P)$. As long as $\lg n < \sqrt{Z}$, the dynamic programming algorithm is better. We get a similar result when comparing dynamic programming 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.

V. Effect of Problem Size

In Section IV, 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.

A. 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 edges. 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 with binary heap, 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 which uses an array for storing vertices, the cost is lower. Now each multiprocessor can store the vertex array and does not need to access it from slow memory. Therefore the bounds on the number of memory accesses 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

TABLE III
ALGORITHM RUNNING TIMES AND CONSTRAINTS.

Algorithm	Time (*PRAM*)	Time (*TMM*)	Constraints	
Floyd-Warshall	$\frac{n^3 \lg n}{P}$	$\frac{n^3 \lg n \cdot L}{\sqrt{Z} C \mathcal{T} P}$	$L \leq \sqrt{Z} C X$	$L \leq Z^{3/2} C / Q$
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	$\frac{n^2 m}{P}$	$\frac{mn^2 L}{C \mathcal{T} P}$	$L \leq CX$	$L \leq CZ/Q$

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.

B. Edges Fit in the Combined Local Memories

When $m = O(PZ/Q)$, the edges fit in all the memories of the multiprocessors 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 multiprocessors can be responsible for relaxing a single edge. Now a portion of the arrays A, B and W fits in each multiprocessor'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 *TMM* model is reduced to $O(n^3 L/(C\mathcal{T}P))$. Again, compare the Bellman-Ford algorithm with Johnson's algorithm using heap. When $m = O(n^2/\lg n)$, Johnson's algorithm that uses heap is better than Bellman-Ford algorithm in *PRAM* model. However, in *TMM* model, Johnson's has run time of $O(Lmn \lg n/(\mathcal{T}P)) = O(Ln^3/(\mathcal{T}P))$, while the Bellman-Ford's with run time of $O(Ln^3/(C\mathcal{T}P))$ flips to be the better one.

VI. CONCLUSION

In this paper, we present a memory access model, called the *TMM* model, that is well suited for modern highly-threaded, many-core systems that employ wide SIMD processing and fast context switches 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 GPU-like 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. 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 GPU-like, 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. 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 multiprocessors, 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 Floyd-Warshall, all of the algorithms presented in this paper are, in fact, parameter-oblivious. And matrix multiplication in Floyd-Warshall 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.

ACKNOWLEDGEMENT

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

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*, 1987, pp. 305–314.

[2] A. Aggarwal, A. K. Chandra, and M. Snir, "Hierarchical memory with block transfer," in *Proc. of 28th Symposium on Foundations of Computer Science*, 1987, pp. 204–216.

[3] A. Aggarwal and J. Vitter, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

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

[5] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the GPU," Dec. 2009.

[6] B. Alpern, L. Carter, and E. Feig, "The uniform memory hierarchy model of computation," *Algorithmica*, vol. 12, no. 2-3, 1994.

[7] B. Alpern, L. Carter, and T. Selker, "Visualizing computer memory architectures," in *Proc. of the 1st Conference on Visualization*, 1990, pp. 107–113.

[8] L. Arge, M. T. Goodrich, M. Nelson, and 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.

[9] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and 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.

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

[11] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably good multicore cache performance for divide-and-conquer algorithms," in *Proc. 19th ACM-SIAM Symp. Discrete Algorithms*, 2008, pp. 501–510.

[12] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and 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.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, Oct. 2008.

[14] J. W. Choi, A. Singh, and 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.

[15] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran, "Oblivious algorithms for multicores and network of processors," in *Proc. of 24th IEEE Int'l Parallel and Distributed Processing Symp.*, Apr. 2010, pp. 1–12.

[16] R. A. Chowdhury and 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.

[17] ——, "Cache-efficient dynamic programming algorithms for multicores," in *Proc. of 20th Symp. on Parallelism in Algorithms and Architectures*, 2008, pp. 207–216.

[18] R. Cole and V. Ramachandran, "Efficient resource oblivious algorithms for multicores," *CoRR*, vol. abs/1103.4071, 2011.

[19] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and 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.

[20] E. W. Dijkstra, "A note on two problems in connexion with graphs," *NUMERISCHE MATHEMATIK*, vol. 1, no. 1, pp. 269–271, 1959.

[21] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, 1962.

[22] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proc. of 10th ACM Symp. on Theory of computing*, 1978.

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

[24] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," in *Proc. of ACM/IEEE Conf. on Supercomputing*, 2006.

[25] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proc. of 16th ACM Symp. on Principles and Practice of Parallel Programming*, 2011.

[26] S. Hong and 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.

[27] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *J. ACM*.

[28] R. M. Karp, "A survey of parallel algorithms for shared-memory machines," University of California at Berkeley, Berkeley, CA, USA, Tech. Rep., 1988.

[29] G. J. Katz and 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.

[30] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and 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.

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

[32] W. Liu, W. Muller-Wittig, and B. Schmidt, "Performance predictions for general-purpose computation on GPUs," in *Proc. of Int'l Conf. on Parallel Processing*, 2007.

[33] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig, "Streaming algorithms for biological sequence alignment on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, pp. 1270–1281, 2007.

[34] Y. Liu, B. Schmidt, and 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*, vol. 3, 2010.

[35] L. Ma and 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.

[36] L. Ma, R. D. Chamberlain, J. D. Buhler, and M. A. Franklin, "Bloom filter performance on graphics engines," in *Proc. of Int'l Conf. on Parallel Processing*, 2011, pp. 522–531.

[37] K. Matsumoto, N. Nakasato, and 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.

[38] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proc. of 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2012, pp. 117–128.

[39] H. Prokop, "Cache-oblivious algorithms," MIT, 1999, Master's thesis.

[40] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proc. of IEEE Int'l Symp. on Parallel and Distributed Processing*, 2009.

[41] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, Aug. 1990.

[42] U. Vishkin, G. C. Caragea, and 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.

[43] J. S. Vitter and M. H. Nodine, "Large-scale sorting in uniform memory hierarchies," *J. Parallel Distrib. Comput.*, vol. 17, no. 1-2, pp. 107–114, Jan. 1993.

[44] J. S. Vitter, E. A. M. Shriver, and E. A. M. S. Z, "Algorithms for parallel memory I: Two-level memories," *Algorithmica*, vol. 12, pp. 110–147, 1994.

[45] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. of ACM/IEEE Conf. on Supercomputing*, 2008.

[46] S. Warshall, "A theorem on boolean matrices," *J. ACM*, vol. 9, no. 1, Jan. 1962.

[47] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the GPU," in *Proc. of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[48] Y. Zhang and J. Owens, "A quantitative performance analysis model for GPU architectures," in *Proc. of IEEE Int'l Symp. on High Performance Computer Architecture*, Feb. 2011, pp. 382–393.