Analysis of Classic Algorithms on GPUs

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

***Abstract*—The recently developed *Threaded Many-core Memory (TMM)* model provides a framework for analyzing algorithms for highly-threaded many-core machines such as GPUs. In particular, it tries to capture the fact that these machines hide memory latencies via the use of a large number of threads and large memory bandwidth. The TMM model analysis contains two components: computational complexity and memory complexity.**

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

Keywords*—*Threaded Many-core Memory (TMM) Model

I. Introduction

Graphics Processing Units (GPUs) are increasingly being exploited as powerful compute engines. A number of high-performance GPU algorithms have been developed, such as sorting [1], hashing [2], dynamic programming [3], graph algorithms [4], and other classic algorithms [5]. Many performance studies have also been conducted [6], [7], [8] to understand the performance of GPU applications.

We are interested in the theoretical asymptotic analysis of algorithms, since it allows us to compare the high-level performance characteristics of algorithms without worrying about low level implementation details of both the algorithm and the particular machine. PRAM is the most common model for theoretical analysis of parallel algorithms. However, the PRAM model ignores some important characteristics of GPU-like highly-threaded machines. For example, PRAM assumes that all memory accesses take constant time; this assumption is violated by the complex memory subsystems of most modern machines and is especially not a reasonable assumption for GPUs. Recently, a few models for asymptotic analysis of GPU algorithms have been proposed [9], [10] that do try to take important characteristics of these machines into consideration.

In this paper, we focus on one of these models, namely, the ***Threaded Multi-core Memory (TMM)*** model [9]. In the TMM model, a program is analyzed in terms of both its computational complexity and its memory complexity. Computational complexity analysis is identical to the standard analysis used by parallel algorithms, in particular, by the PRAM model. However, the TMM model is meant to capture important characteristics of the memory subsystem of highly-threaded many-core machines, such as GPUs. Therefore, the analysis framework for memory complexity in the TMM model takes three important characteristics of these machines into consideration. (1) There is a large latency for accessing the largest and slowest memory on the machine, namely the global memory. (2) These machines have a large number of hardware managed threads and use fast context switching between these threads to hide this latency. (3) In addition to the fast context switching, these machines typically have a large memory bandwidth between the core and the slow memory; therefore, if the memory accesses are regular and predictable, then many memory operations can be *grouped* into one large bandwidth memory transfer. Therefore, in this model, the memory complexity of a program depends not only on how many memory accesses it contains, but also on both how many threads it can use and how effectively its memory accesses can be grouped.

In the TMM model, the total running time of an algorithm on P cores depends on both its computational complexity and memory complexity. The computational complexity is derived in terms of ***work*** T_1 — the total amount of computation, or, in other words, its running time on 1 processor — and ***span*** T_∞[1] — the amount of computation on the critical path or, in other words, the running time on an infinite number of processors. The memory complexity is derived using a parameter M, which is the total number of memory transfers (either grouped or not) from slow memory. The total running time on P cores is then derived using both computational and memory complexity, as well as machine parameters such as the latency to slow memory, memory bandwidth to slow memory, and number of available hardware threads.

Asymptotic models are meant to capture important characteristics of the machine while ignoring low-level details; however, a model is only useful if it can predict and explain empirical data. Like all asymptotic models, the TMM model makes a few different categories of predictions: (1) For a given algorithm, it predicts the impact of increasing problem sizes on performance. (2) When we have multiple algorithms for the same problem, asymptotic models can help us compare them and understand relative performance at a high level. However, unlike the standard RAM or PRAM models, these predictions may depend on some machine parameters, such as the relationship between memory latency, the fast memory size, and the number of allowable threads on the machine.[2]

[1] Also called depth, time, or critical-path length in the literature. We reserve the term "time" for the running time on P processors, not an infinite number of processors.

[2] This is not very different from other models that also consider memory subsystem, such as DAM (see Section IX).

In addition, unlike the PRAM model, the TMM model also allows us consider the effects of changing the parameters of the machine itself (such as reducing memory latency or changing the fast memory size) on the performance of an algorithm.

In this paper, we evaluate the effectiveness of the TMM model in terms of its ability to predict empirical performance of algorithms. We extend our analysis [11] of a number of classic algorithms with a wider range and more in-depth investigation under the TMM model. Specifically, we analyze suffix trees and suffix arrays for the problem of string matching, Fast Fourier Transform (FFT), merge sort, and Wylie's list ranking algorithm. For all of these algorithms, we compare the analytical conclusions derived using the TMM model to the empirical observations of researchers who have previously implemented these algorithms, in order to investigate whether the TMM analysis correctly predicts the trends observed in the empirical data, and whether it can explain previously unexplained characteristics of the data. In addition, we also take a second look at an all-pairs shortest paths algorithm, which was previously analyzed in the original TMM paper [9], and perform additional experiments, both to investigate if the TMM model applies to the newest GPU machines and to investigate the effect of changing machine parameters.

Our findings indicate the TMM model is effective at explaining many kinds of empirical observations, and its analysis framework appears to be well suited to understanding and predicting the high-level characteristics of the performance of algorithms on these machines. In particular, one can compare the effect of computational and memory complexity on the runtime for various parameter settings, and explain the behavior of algorithms for varying problem sizes. In addition, the TMM model also predicts the effect of changing the machine parameters, such as memory latency, on the performance of algorithms. Note that the TMM model ignores constant factors; therefore, it can only explain high-level trends, not particular numerical values at which these trends may occur. For all the classic algorithms considered in this paper, the TMM analysis indicates that considering both computational and memory complexity is necessary to understand the performance of algorithms on many-core machines such as GPUs.

II. Threaded Many-core Memory Model

The TMM model [9] models highly-threaded many-core machines as consisting of a number of core groups (called multiprocessors on NVIDIA GPUs), each containing a number of processors (or cores) and a fast local on-chip memory of size Z shared within a core group. Computation and access to fast memory takes unit time in the TMM model. A large slow global memory is shared by all the core groups and accessing the slow memory takes L time steps (L is the ***memory latency***). Data is transferred from slow to fast memory in ***chunks*** of maximum size C, also called the ***chunk size*** or ***memory access width***; this represents the large bandwidth between slow and fast memory.[3] These machines support a large number of hardware threads, much larger than the total number of cores P, and these threads are used to hide the memory latency. The hardware limit on the ***number of threads per core*** is represented by X; the total number of threads supported on the machine is therefore bounded by XP.

An algorithm in the TMM model is analyzed using its computational complexity represented by its ***work*** T_1 — the total number of operations in the computation — and ***span*** T_∞ — the number of operations along the critical path. The memory complexity is analyzed in terms of the ***total number of global memory transactions*** M. Note that up to C accesses to global memory are grouped; if they are grouped, it counts as a single memory transaction when we calculate M. In addition, $\mathcal{T}$ is an additional parameter for ***number of threads per core*** used by the algorithm; it depends on both the problem size and the hardware limit X. Table I summarizes the model parameters.

TABLE I: TMM model parameters.

	Description
Z	Size of fast local memory per core group
L	Time for a slow global memory access
C	Memory access width/chunk size
P	Total number of processors (cores)
X	Hardware limit on number of threads per core
T_1	Total number of operations in the program (work)
T_∞	Number of operations on the critical path (span)
M	Number of global memory transactions
$\mathcal{T}$	Number of threads per core

The TMM model assumes that the program is scheduled perfectly. Under this assumption, and using the above parameters, the performance of an algorithm in this model is described using the following equation which calculates T_P, the running time of the algorithm on P cores of the machine:

$$T_P = O\left(\max(\frac{T_1}{P}, T_\infty, \frac{ML}{\mathcal{T}P})\right). \quad (1)$$

All three terms are lower bounds on the runtime of the algorithm. The first two terms, which represent computational complexity, should be obvious. The third term requires some explanation. There are a total of M memory operations, and each takes L time steps. Therefore, assuming perfect scheduling, the number of steps spent on memory transactions per core is ML/P. Since each core has $\mathcal{T}$ threads running, $\mathcal{T}$ of these operations can be overlapped, leading to the average amount of time spent on memory access $ML/(\mathcal{T}P)$.

In the following sections, we will calculate the work T_1, span T_∞ and the number of memory transactions M for all the algorithms we consider. Using these, we will calculate T_P and compare each of its component terms. If the last term (the term including M) dominates, we say that the running time is dominated by memory complexity, otherwise, we say that the performance is dominated by computational complexity. However, for all algorithms we consider, there is sufficient parallelism so that the runtime is never limited by the span; that is $T_\infty \ll T_1/P$ for reasonable problem sizes. Thus, although we analyze the span for each algorithm for completeness, during our interpretation and comparison with empirical data, we mostly consider the first term when talking about computational complexity. In addition, when we

[3]The chunk can either be a cache line of hardware managed caches or an explicitly-managed combined read from multiple threads.

compare two terms, we are doing ***asymptotic comparison*** rather than value comparison. This allows us to compare quantities with different units. This comparison is valid since the difference due to dimension affects only the constant factors.

III. SUFFIX TREE VS. SUFFIX ARRAY

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

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

A. Suffix Tree

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

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

$$T_P = O\left(\max(\frac{nk}{P}, k, \frac{nkL}{\mathcal{T}P})\right). \tag{2}$$

The last term (memory complexity term) can be refined into two terms depending on the relation between the batch size n and the thread limit XP. When $n \leq XP$, each thread handles a single query; the number of threads $\mathcal{T}P$ increases with n. So, $n = O(\mathcal{T}P)$, and the two cancel in the last term. When $n > XP$, we do not have sufficient threads to run all queries on separate threads. All X possible threads are used ($\mathcal{T} = X$) and the n queries are divided among these threads. Therefore, the third term becomes $\frac{nkL}{XP}$. Considering both scenarios for all possible n, Eq. (2) can be expressed as:

$$T_P = O\left(\max(\frac{nk}{P}, k, kL, \frac{nkL}{XP}\right). \tag{3}$$

B. Suffix Array

A suffix array is an array of integers giving the starting positions of suffixes of a reference string in lexicographical order [12]. In other words, all the suffixes are indexed by their individual starting positions in the original reference string, and then sorted lexicographically. To match a query string, we perform a binary search over the suffix array. At each step, we compare the query string to the string at that point in the array, and either find a match or decide to go up or down. The search does not stop until the two indices are located such that all elements between these two in the suffix array are the instances of occurrence for the underlying query string, allowing us to find multiple matches if they exist.

The computational complexity analysis is straightforward. According to the algorithm in [13], each thread matches an individual query in $O(k \lg m)$ span with $O(k \lg m)$ work. Therefore, the total work for an n-sized batch of queries is $T_1 = O(nk \lg m)$ and the span is $T_\infty = O(k \lg m)$. Now for the memory complexity. At each step, the whole query is compared against one entire suffix in the reference string; therefore, this suffix can be accessed in chunks of size C. Therefore, each memory transfer can transfer $O(C)$ characters[4] of the string for comparison. Therefore, the memory complexity for each query is $O(\frac{k \lg m}{C})$ and the memory complexity of the batch is $M = O(\frac{nk \lg m}{C})$. Therefore, using Eq. (1) and by the same logic of refining the last term (based on the number of threads) as suffix trees in Eq. (3), the runtime can be expressed as:

$$T_P = O\left(\max(\frac{nk \lg m}{P}, k \lg m, \frac{k \lg m \cdot L}{C}, \frac{nk \lg m \cdot L}{CXP})\right). \tag{4}$$

C. Comparison and Empirical Validation

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

Let us focus on Eq. (3) and Eq. (4) to theoretically compare the performance of the two suffix algorithms. We notice a few interesting things about these bounds:

1) First, when the number of queries n is small, for both algorithms, the runtime is independent of n, and is only dependent on the memory cost. In this region, whether suffix trees or suffix arrays will perform better depends on the relationship between terms kL and $\frac{k \lg m \cdot L}{C}$. For most reference strings $\lg m < C$, and suffix arrays are better. Only for very large reference strings will suffix trees be faster.

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

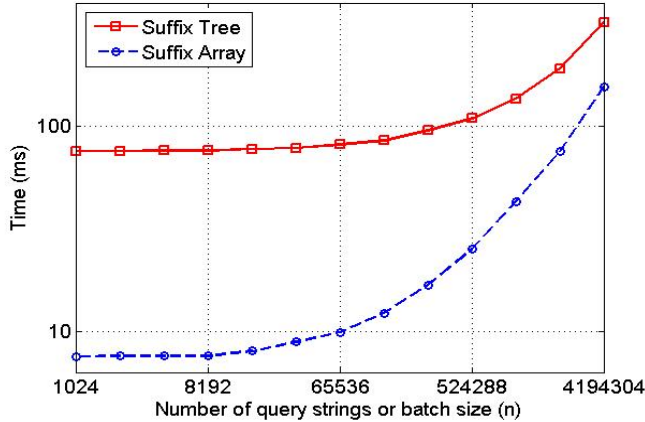


Fig. 1: Performance of suffix trees and suffix arrays on GPU. Empirical data are from Encarnaijao et al. [13].

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

3) After the transition (the point at which the runtime starts depending on n), the runtimes of both the suffix tree and the suffix array increase linearly with problem size n, however at different rates. Whether suffix tree or suffix array runtime increases faster depends on the machine parameters again. When $X \geq L$, both are dominated by computation complexity (suffix tree by $O(\frac{nk}{P})$, and suffix array by $O(\frac{nk \lg m}{P})$). It is clear that the runtime of suffix array grows a factor of $O(\lg m)$ faster than suffix tree. Given sufficiently large n, one expects that suffix trees catch up in performance with suffix arrays. When $L/C < X < L$, suffix tree is dominated by memory complexity $O(\frac{nkL}{XP})$, while suffix array is still dominated by computation $O(\frac{nk \lg m}{P})$. When $X \leq L/C < L$, they are both bounded by memory complexity (suffix tree by $O(\frac{nkL}{XP})$, and suffix array by $O(\frac{nk \lg mL}{CXP})$). In the second two cases, which one grows faster depends on the reference size m and machine parameters L, X, and C.

Comparison with empirical data: Consider the empirical performance of these algorithms as reported by Encarnaijao et al. [13]. We have reproduced the graph in Figure 1 from their data. The figure supports all three predictions above, at least qualitatively. For small values of n, both algorithms are dominated by memory complexity, and the running times of both suffix tree and array remain flat as n increases, showing that the performance is independent of n and depends on the third term of the running time bounds in Eq. (3) and Eq. (4). In addition, for these experiments, the reference sequence is also of a moderate length ($m = 10^7$), and the alphabet size is small; therefore, we expect that $C > \lg m$, suffix arrays perform better within the flat range.

For the GPU being used (NVIDIA GTX 580), the hardware limit on the number of threads X is large and of the same order as the latency.[5] Therefore, the suffix array's transition (where the running time starts depending on n) happens for smaller values of n than the suffix tree. This fact is also consistent with the empirical observations. As n increases, both performance curves ultimately bend up. The bend occurs at different points for the two algorithms, and as predicted above, the suffix array curve bends sooner than the suffix tree curve.

After the transition of algorithm performance to be linear with n, the runtimes of the two algorithms increase with different rates. For the first case, the running time for suffix array increases faster than suffix tree. If we are in the second case, for these machines, we expect $L/X < \lg m$, since it is expected to be small. Therefore, for both the first and the second cases, suffix array running time grows faster than the suffix tree running time. In the figure, as predicted, the slope of the curve for suffix arrays is steeper than the slope for suffix trees after the transition. Even though the curve does not go that far, we can speculate that eventually, for large enough n, suffix trees will outperform suffix arrays.

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

IV. Fast Fourier Transform (FFT)

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

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi k \frac{n}{N}}, \qquad k = 0, ...N-1. \quad (5)$$

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

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

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

Next we analyze the FFT algorithm on GPUs. At each of the $\lg N$ steps, $N/2$ 2-point FFTs are computed in parallel, each by a thread. Therefore, the work is $T_1 = O(N \lg N)$ and the span is $T_\infty = O(\lg N)$. For each stage of the butterfly, we have to perform $O(N)$ memory accesses, but these are predictable at regular intervals. Therefore, they can be grouped into chunks of size C, and the total number of memory transfers is $M = O(N \lg N/C))$. Using Eq. (1) and by refining the last term (based on relationship between N and $\mathcal{T}$) as in suffix trees/arrays, we get the running time of

$$T_P = O\left(\max(\frac{N \lg N}{P}, \lg N, \frac{\lg N \cdot L}{C}, \frac{N \lg N \cdot L}{CXP})\right). \quad (6)$$

Comparison with empirical data: We now compare our analytical observations with the empirical result of Govindaraju et al. [14], who evaluated the FFT algorithm with radix-2 Stockham algorithm on NVIDIA GPUs. Here, we would like to highlight one interesting experiment, where they varied the memory clock rate. Note that increasing the memory clock rate is equivalent to shrinking the memory latency L. In Eq. (6), L affects the runtime only as long as the algorithm runtime is dominated by the memory complexity term (the last two terms). As L shrinks, eventually it is small enough that the algorithm runtime is no longer dominated by the memory complexity; instead it is dominated by the computation complexity (the first term). Therefore, after a certain point of increasing memory frequency, the algorithm runtime no longer depends on L, and further increases in frequency have no affect. Figure 2 shows the running times for 2 problem sizes as the memory frequency increases. The graph validates the above observation; after a certain point, increasing the memory clock rate (decreasing L) does not affect the runtime, and the curve becomes flat.

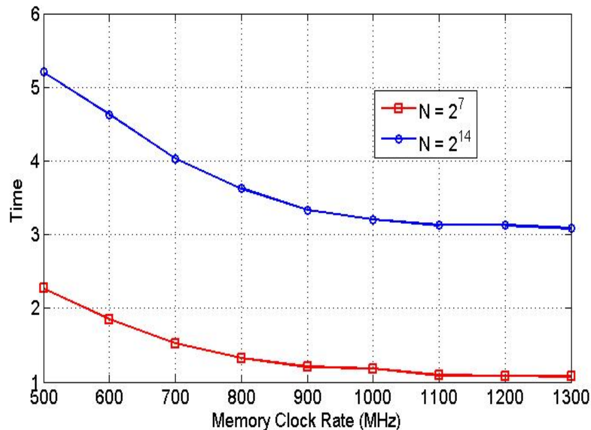


Fig. 2: Runtime of FFT algorithm with various memory frequencies on an NVIDIA GTX280. The FFTs are performed for two problem sizes $N = 2^7$ and $N = 2^{14}$. The y-axis is the runtime plotted on an arbitrary scale, as the runtime data are converted from GFLOPs from Govindaraju et al. [14]. The x-axis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

V. Merge Sort

Merge sort is generally considered to be the preferred technique for external sorting, where the sequence being sorted is stored in a large external memory and the processor only has direct access to a much smaller local memory. In particular, it is asymptotically optimal in the disk access model [15]. In this section, we analyze the merge sort of Satish et al. [1].

The idea of the merge sort algorithm is to divide the input sequence into blocks of size Z, sort them in parallel locally within core groups so as to utilize the fast memory using odd-even merge sort [16], and finally recursively merge them using the ***blocked merge***. The blocked merge is an important subroutine of merge sort. It merges two sorted sequences A and B into a new sorted sequence S of size n that contains all the elements of A and B. We first analyze the algorithm for blocked merge and then use it to analyze merge sort.

A. Blocked Merge

The idea of this algorithm is to decompose the overall problem into many smaller problems, each of which is small enough to fit in fast memory. Therefore, sequences A and B are cut into smaller sequences as follows: first divide the sequences A and B into sections of size $Z/2$; say that the boundary elements are $a_1, a_2, ..., a_{2n/Z}$ and $b_1, b_2, ..., b_{2n/Z}$ respectively. We then, in parallel, search for these boundary elements in the other sequence. If we include the boundary elements and these new binary search locations, this divides both sequences into $4n/Z$ sections, each of size at most $Z/2$ and the corresponding sections match up. That is, we can individually merge the ith section in A to the ith section in B and this leads to the correct final sequence S. The process of creating this partition involves n/Z parallel binary searches; leading to a total of $T_\infty = O(\lg n)$ span, $T_1 = O(\frac{n}{Z} \lg n)$ work, and incurs $M = O(\frac{n}{Z} \lg n)$ memory transfers, since the memory accesses cannot be grouped due to unpredictable access patterns.

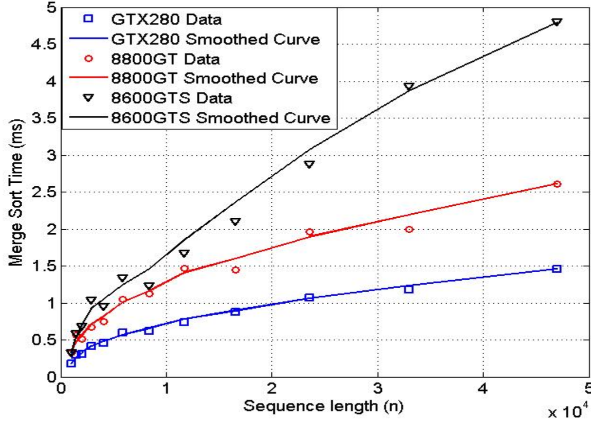
Once subsequences of both A and B fit in memory, $n < Z$, we can merge the many subsequences in parallel. For each element $x \in A \cup B$ we compute the rank $rank(x, S)$, which is equal to $rank(x, A) + rank(x, B)$. If $x \in A$, it's rank in A is simply its index in A, since A is sorted. Its rank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

Since there are a total of n/Z blocks, which can all be merged in parallel, the total span is $T_\infty = O(\lg Z)$, with $T_1 = O(n \lg Z)$ work and $M = O(n/C)$. Therefore, adding the cost of partitioning and the local merges, and applying Eq. (1), we get

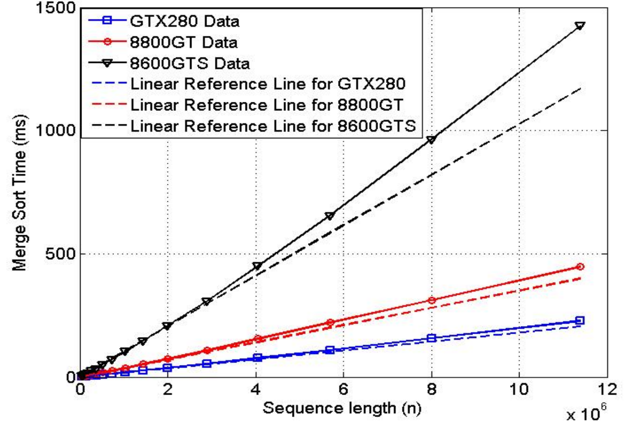
$$T_P = O\left(\max(\frac{n \lg Z + \frac{n}{Z} \lg n}{P}, \lg n, \frac{(\frac{n}{C} + \frac{n}{Z} \lg n)L}{\mathcal{T}P})\right). \quad (7)$$

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

$$T_P = O\left(\max(\frac{n \lg Z}{P}, \lg n, \frac{nL}{C\mathcal{T}P})\right). \quad (8)$$



(a) Runtime for small sequence size.



(b) Runtime for large sequence size.

Fig. 3: Merge sort on multiple GPUs (data from [1]); The solid lines are smoothed curves from data and dotted lines are linear references. (a) For small n, the runtime increases slower than linearly with n. (b) For large n, the runtime increases faster than linearly with n.

B. Merge Sort

The odd-even merge sort is based on a bitonic sorting network and proceeds in $O(\lg n)$ phases for a sequence of size n. In each phase, there is $O(n \lg n)$ work and $O(\lg n)$ span. Therefore, the total work is $O(n \lg^2 n)$ and the total span is $O(\lg^2 n)$. For subsequences of size Z, the odd-even sorting step takes $O(\lg^2 Z)$ span, $O(Z \lg^2 Z)$ work, and $O(Z/C)$ memory transfers. Since there are a total of $O(n/Z)$ blocks to be sorted in parallel, the total span is $O(\lg^2 Z)$, total work is $O(n \lg^2 Z)$ with $O(n/C)$ memory transfers. Given that $Z = \Omega(Z)$, $T_1 = O(n \lg^2 Z)$, $T_\infty = O(\lg^2 Z)$.

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

Therefore, ultimately we can combine the sorting and the merging step. Since $O(\lg(n/Z)) = O(\lg n - \lg Z) = \Omega(\lg Z)$, we can substitute into Eq. (1) and by refining the last term according to problem size to get the runtime:

$$T_P = O\left(\max\left(\frac{n \lg Z \lg \frac{n}{Z}}{P}, \lg n \lg \frac{n}{Z}, \frac{\lg \frac{n}{Z} \cdot L}{C}, \frac{n \lg \frac{n}{Z} \cdot L}{CXP}\right)\right). \tag{9}$$

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

Comparison with empirical data: Empirical results in Figures 3a and 3b are replotted from the data represented in [1]. First let us look at small values of n in Figure 3a. In this range, we see that the running time increases very slowly with n, as expected, since it depends on the third term in Eq. (9). If we look at large values of n, however, in Figure 3b, we see that the running time increases a little bit faster than linearly with n (the dashed lines show linear growth for comparison), which we can speculate is approximately $n \lg (n/Z)$. The TMM model is able to predict the growth of running time for both small and large sequences. In addition, it indicates that increasing the hardware limit on threads (increasing X) is likely to increase the area where the growth is slow.

VI. List Ranking

List ranking is a classic problem where we want to compute the rank of each element in a list in parallel. Here we analyze the performance of Wyllie's list ranking algorithm [17] in the TMM model. In Wylie's algorithm, each element's rank is computed in parallel by repeated pointer jumping; the successor pointer of each element in the list is repeatedly updated to its successor's successor, while also updating the rank estimate. Given a linked list of n elements, the algorithm finishes in $T_\infty = O(\lg n)$ span, thereby making the total work $T_1 = O(n \lg n)$.

Rehman [18] implement this algorithm on GPU, assigning one thread to each element in the list. Each thread accesses $O(\lg n)$ elements, and these memory accesses cannot be grouped, since they are relatively far away from each other. Therefore, the number of memory transfers is $M = O(n \lg n)$. Refining the last term by considering all possible n, we get the runtime from Eq. (1):

$$T_P = O\left(\max\left(\frac{n \lg n}{P}, \lg n, \lg n \cdot L, \frac{n \lg n \cdot L}{XP}\right)\right). \tag{10}$$

At a small list size when $n < \min(LP, XP)$, the third term in Eq. (10) dominates the performance. Therefore the analytic runtime is linear with $\lg n$. As n increases, performance may be dominated by memory complexity due to the last term or computation complexity due to the first term; it depends on the relationship between L and X. Specifically, when $L > X$, and $n > XP$, the memory complexity dominates; when

$L < X$ and $n > LP$, the computation complexity dominates. However, in both cases, the running time increases with $n \lg n$.

Comparison with empirical data: These observations are borne out with the empirical data which is replotted in Fig. 4 from the data presented in [18]. The runtime increases slowly for small values of n and faster for larger values of n.

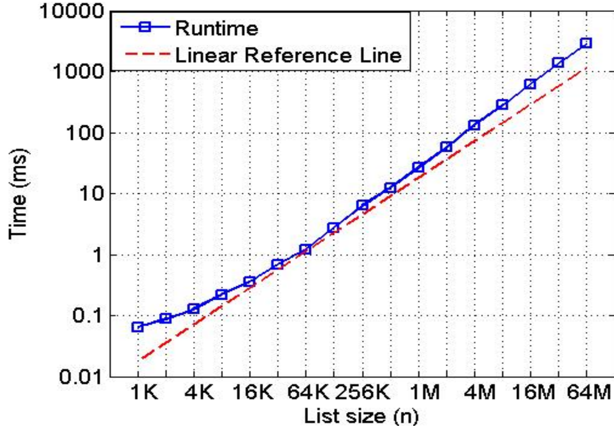


Fig. 4: Runtime of Wyllie's algorithm on NVIDIA GTX 280 (data from [18]). The curve indicates that the runtime grows slowly for small n and faster for larger n (dotted line is a linear reference). Note that the graph is a log-log plot in order to expose the trends over a wide range of n.

VII. All-Pairs Shortest Paths

Given a graph $G = (V, E)$ with n vertices and m weighted edges, an ***all-pairs shortest paths*** algorithm calculates the shortest weighted path from every vertex to every other vertex. Here we consider the dynamic programming algorithm [19] that uses repeated matrix multiplication. The graph is represented as an adjacency matrix A where A_{ij} represents the weight of edge (i, j). A^l is a transitive matrix where A^l_{ij} represents the shortest path from vertex i to vertex j using at most l intermediate edges. $A^1 = A$ and A^2 can be calculated from A^1 using squaring (similar to matrix multiplication):

$$A^2_{ij} = \min_{0 \le k < n}(A^1_{ij}, A^1_{ik} + A^1_{kj}). \tag{11}$$

In order to calculate all-pairs shortest paths, we simply calculate A^{n-1} using repeated squaring. The adjacency matrix is stored on off-chip global memory, and matrix multiplication is performed by dividing the matrix into blocks and allowing each multiprocessor to operate on individual blocks.

This algorithm was analyzed in the original TMM paper [9] with $T_1 = O(n^3 \lg n)$, $T_\infty = O(n \lg n)$ and $M = O(n^3 \lg n / (\sqrt{Z}C))$. Using these values, the speedup, defined as T_1/T_P on P processors, is

$$S_P \quad = \quad \Omega\left(\min(P, n^2, \frac{\sqrt{Z}C\mathcal{T}}{L} \cdot P)\right) \tag{12}$$

Comparison with empirical data: In [9], the authors conducted experiments on an NVIDIA GTX480 and show that, as the TMM bounds indicate, increasing the number of threads per core $\mathcal{T}$ improves speedup while the third term dominates and then the speedup flattens out, no longer depending on $\mathcal{T}$, as the first term starts dominating.

In this paper, we reproduce that experiment and conduct additional experiments to make some more observations. The results are shown in Figure 5. We wish to plot speedup T_1/T_P; however, since a single core execution time T_1 on a GPU is not well-defined, we 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"; however, the shape of the curves are representative of the speedup achievable relative to a fixed serial execution time.

1) We conducted experiments on the latest NVIDIA GTX680 *Kepler* architecture (shown in Figure 5b) in addition to the *Fermi* architecture GTX480 (shown in Figure 5a). The figures are qualitatively similar — as $\mathcal{T}$ increases, the speedup increases for a while and then flattens out — as predicted by the TMM model.

2) Quantitatively, the machines have different characteristics. GTX680 has a smaller hardware limit on the number of threads per core X.[6] However, it also requires fewer threads per core $\mathcal{T}$ to achieve almost flattened speedup, indicating that memory latency L is small compared to other parameters and can be hidden with fewer threads. However, it appears that the speedup curve is not completely flat even at the maximum number of threads, indicating that it may be beneficial to increase the hardware limit on the number of threads further.

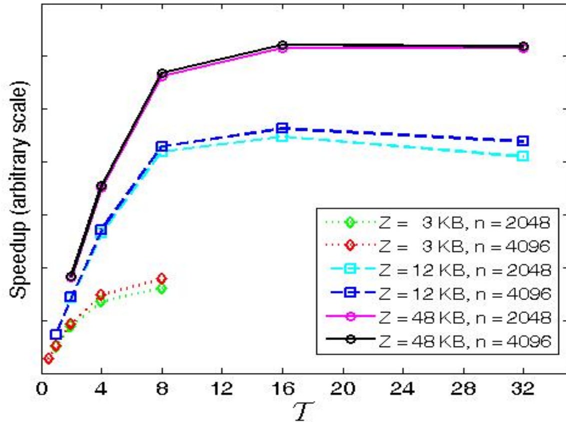
3) We also conducted experiments to evaluate the effect of local memory size Z by artificially decreasing the memory by filling it with dummy data. The TMM model indicates that reducing Z should decrease speedup, which is validated by the experiments on both GPUs. The line for $Z = 3$ KB stops at $\mathcal{T}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

This set of experiments indicates that the TMM model can be used to understand the importance of a variety of parameters on algorithm performance, not only $\mathcal{T}$ and L, but also Z. More importantly, the TMM model can be informative with respect to understanding limits on performance that are due to constraints on machine parameter ranges. For example, the maximum number threads per core, X, is lower in the Kepler architecture than in the Fermi architecture, while Figure 5b indicates that a larger value of X for Kepler could provide a performance benefit for this all-pairs shortest paths algorithm.

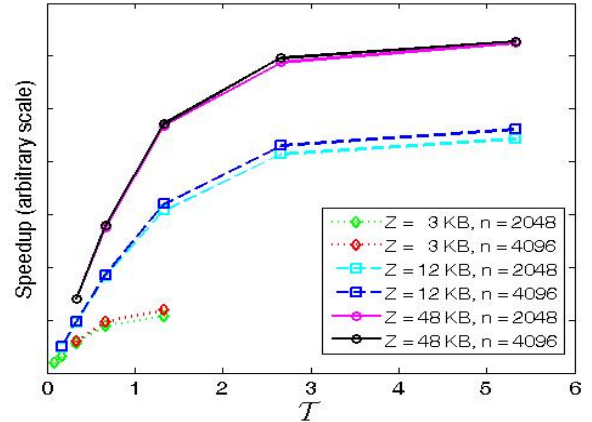
VIII. Analysis of Additional Algorithms

In this section, we briefly present the bounds on other classic algorithms in the TMM model. These bounds are listed

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



(a) Speedup on NVIDIA GTX480 (Fermi).



(b) Speedup on NVIDIA GTX680 (Kepler).

Fig. 5: Speedup of all-pairs shortest paths algorithm on two generations of NVIDIA GPUs. On GTX480, with memory size 12 KB and 48 KB, using more than 16 threads hides the memory latency completely; on GTX680, due to the hardware limit on $\mathcal{T}$, latencies are not fully hidden, and the speedup curve is still climbing slowly.

in Table II. Due to space constraints, we are not able to describe these algorithms, but most of them are either well-known primitives (such as reduce, scan, merge) or classic algorithms for well-understood problems (such as connected components, minimum spanning tree, sorting). For completeness, we include the bounds of the algorithms we analyzed in this paper. Note that the bounds on M shown are for large n; we are not presenting the detailed analysis for all ranges of n as we did above. n is generally the problem size; for graph algorithms, n is the number of nodes and m is the number of edges; for string matching, n is the number of query strings, m is the length of the reference string, and k is the maximum query length.

TABLE II: Analysis for some more classic algorithms.

Algorithms	Work T_1	Span T_∞	Mem. Ops M
Reduce [20]	n	$\lg n$	n/C
Scan [20]	n	$\lg n$	n/C
Merge [1]	$n \lg Z$	$\lg Z$	n/C
Merge Sort [1]	$n \lg Z \lg \frac{n}{Z}$	$\lg Z \lg \frac{n}{Z}$	$\frac{n}{C} \lg \frac{n}{Z}$
Odd-even Sort [16]	$n \lg^2 n$	$\lg^2 n$	$\frac{n}{C} \lg \frac{n}{Z}$
ConnectedComp [21]	$(m+n) \lg n$	$\lg^2 n$	$\frac{m+n}{C} \lg n$
MST-Boruvaka [22]	$m \lg n$	$\lg n$	$m \lg n$
Suffix Tree [13]	nk	k	nk
Suffix Array [13]	$nk \lg m$	$k \lg m$	$nk \lg m/C$
FFT [14]	$n \lg n$	$\lg n$	$n \lg n/C$
List Ranking [18]	$n \lg n$	$\lg n$	$n \lg n$
APSP [9]	$n^3 \lg n$	$n \lg n$	$n^3 \lg n/(\sqrt{Z}C)$

IX. Related Work

In this section, we review some related work on models for algorithm analysis, with emphasis on the models for parallel programs, particularly on GPUs. The most fundamental model that is broadly used to analyze sequential algorithms is the Random Access Machine (RAM) model [23], where we assume that all operations, including memory accesses, take unit time. Disk Access Machine (DAM) analysis [15] is the oldest model that incorporates the memory complexity into the analysis of algorithms by counting the number of memory transfers from slow to fast memory. Since then, many other models that consider memory accesses have been developed [24], [25]. Parallel Random Access Machine (PRAM) [26] is analogous to the RAM model for parallel algorithms, and many algorithms have been designed and analyzed in this model. The PRAM model does not consider the memory hierarchy of modern parallel machines, and therefore, a number of models have been designed to analyze memory performance of parallel algorithms [10], [27].

Recently, several models have been proposed to analyze the performance of algorithms on a GPU. In this paper, we used the TMM model designed by Ma et al. [9]. Kirtzic et al. [28] propose a Parallel GPU Model (PGM), which is essentially an adaption of the Bulk-Synchronous Parallel (BSP) model [29], and equates a superstep in BSP with a function unit of GPU program, such as the computation kernel, and data passing between CPU and GPU. However, this model does not explicitly model the memory subsystem and assumes uniform cost access to all levels of memory. Nakano [10] proposes the Hierarchical Memory Machine (HMM) model, which consists of multiple Discrete Memory Machines (DMMs) representing shared memory and a single Unified Memory Machine (UMM) representing global memory. The HMM model does consider both shared memory accesses and the grouping of global memory accesses. Haque et al. [6] propose a Many-core Machine Model (MMM) based on the Graham-Brent theorem, which is quite similar to the TMM model, but does not model the impact of threading for hiding memory latency.

There are also a number of non-asymptotic models that capture GPU performance. Sim et al. [30] present the GPU-Perf framework, which quantitatively estimates performance along four dimensions: inter-thread instruction-level parallelism, memory-level parallelism, computing efficiency, and serialization effects. These four metrics help to identify performance bottlenecks. Ma et al. [31] design an analysis frame-

work for many-core architectures, bridging the gap between asymptotic models and calibrated models that quantitatively predict runtime. The framework jointly addresses parallelism, latency-hiding, and occupancy; helps to reduce the configuration space for tuning kernels; and reflects performance trends as the problem size and other parameters scale. Zhang et al. [5] 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.

X. Conclusion

In this paper, we analyze some classic algorithms in the TMM model and compare the analytical results with empirical results reported in the literature. This comparison indicates that the TMM model is effective at explaining empirical performance for highly-threaded many core GPUs, especially the NVIDIA GPUs. In particular, the TMM model seems to be effective at predicting the effect of changing the problem size, thread count, and machine characteristics like memory latency and local memory size on the running time of algorithms. There are several directions of future work. First, most of the algorithms we analyze are relatively simple algorithms with ample parallelism. We could use the TMM model to understand the performance of more complex algorithms. Second, the TMM model only captures the performance of a single machine with a 2-level hierarchy. We would like to extend it to consider algorithms running on systems containing more than one GPU. Finally, there are other models that have been proposed for analyzing algorithms on GPUs. We would like to explore them and investigate their effectiveness.

Acknowledgments

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

References

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

[2] 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.

[3] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig, "Streaming algorithms for biological sequence alignment on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 9, pp. 1270–1281, 2007.

[4] 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.

[5] 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.

[6] S. A. Haque, M. M. Maza, and N. Xie, "A many-core machine model for designing algorithms with minimum parallelism overheads," in *Proc. of High Performance Computing Symp.*, 2013.

[7] 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.

[8] L. Ma and R. D. Chamberlain, "A performance model for memory bandwidth constrained applications on graphics engines," in *Proc. of the 23rd Int'l Conf. on Application-specific Systems, Architectures and Processors*, 2012.

[9] L. Ma, K. Agrawal, and R. D. Chamberlain, "A memory access model for highly-threaded many-core architectures," *Future Generation Computer Systems*, vol. 30, pp. 202–215, January 2014.

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

[11] L. Ma, K. Agrawal, and R. Chamberlain, "Theoretical analysis of classic algorithms on highly-threaded many-core GPUs," in *Proc. of 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2014, pp. 391–392.

[12] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," in *Proc. of the ACM-SIAM Symp. on Discrete Algorithms*, 1990.

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

[14] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete Fourier transforms on graphics processors," in *Proc. of ACM/IEEE Supercomputing Conf.*, 2008.

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

[16] P. Kipfer and R. Westermann, "Improved GPU sorting," in *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, Mar. 2005, ch. 46.

[17] J. C. Wyllie, "The complexity of parallel computations," Ph.D. dissertation, Cornell University, Ithaca, NY, USA, 1979.

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

[19] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[20] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proc. of 22nd ACM SIGGRAPH/EUROGRAPHICS Symp. on Graphics Hardware*, 2007, pp. 97–106.

[21] J. Soman, K. Kothapalli, and P. J. Narayanan, "Some GPU algorithms for graph connected components and spanning tree." *Parallel Processing Letters*, vol. 20, no. 4, pp. 325–339, 2010.

[22] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, "Fast minimum spanning tree for large graphs on the GPU," in *Proc. of Conf. on High Performance Graphics*, 2009, pp. 167–171.

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

[24] A. Aggarwal, A. K. Chandra, and M. Snir, "Hierarchical memory with block transfer," in *Proc. of Symp. on Foundations of Comp. Sci.*, 1987.

[25] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. of Symp. on Foundations of Comp. Sci.*, 1999.

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

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

[28] J. S. Kirtzic and O. Daescu, "A parallel algorithm development model for the GPU architecture," in *Proc. of Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, 2012.

[29] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[30] J. Sim, A. Dasgupta, H. Kim, and R. W. 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.

[31] L. Ma, R. Chamberlain, and K. Agrawal, "Performance modeling for highly-threaded many-core GPUs," in *Proc. of the 25th Int'l Conf. on Application-specific Systems, Architectures and Processors*, 2014.