Performance Modeling for Highly-threaded Many-core 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*—Highly-threaded many-core GPUs can provide high throughput for a wide range of algorithms and applications. Such machines hide memory latencies via the use of a large number of threads and large memory bandwidth. The achieved performance, therefore, depends on the parallelism exploited by the algorithm, the effectiveness of latency hiding, and the utilization of multiprocessors (occupancy). In this paper, we extend previously proposed analytical models, jointly addressing parallelism, latency-hiding, and occupancy. In particular, the model not only helps to explore and reduce the configuration space for tuning kernel execution on GPUs, but also reflects performance bottlenecks and predicts how the runtime will trend as the problem and other parameters scale. The model is validated with empirical experiments. In addition, the model points to at least one circumstance in which the occupancy decisions automatically made by the scheduler are clearly sub-optimal in terms of runtime.**

***Keywords*—GPGPU, Performance Model, Threaded Many-core Memory (TMM) Model, All-pairs Shortest Paths (APSP)**

I. Introduction

Highly-threaded many-core Graphics Processing Units (GPUs) are designed to maximize throughput by hiding memory latencies through fast context-switching between thousands of threads.[1] They have been extensively exploited as powerful compute engines for a wide range of algorithms and applications, such as sorting [1], hashing [2], linear algebra [3], dynamic programming [4], graph algorithms [5], and many other classic algorithms [6]. In distributed computing, notably streaming computing [7], [8], GPUs are also used to accelerate certain modules of applications [9]. An important issue that remains, however, is a comprehensive understanding of what algorithmic and architectural features have significant impact on the performance of particular algorithms. Implementing an algorithm and coding it to run correctly is not very difficult, but getting it to perform well can still be quite a challenge.

Performance of algorithms on GPUs largely depends on the suitability of the underlying algorithm, the vulnerability of the program to memory behaviors, and the efficiency of scheduling on many-core architectures. An algorithm is well suited for GPUs only when it has sufficient parallelism and is not unduly bounded by memory latencies. A program runs efficiently only when it organizes a large number of threads concurrently proceeding with the computation while not incurring too much memory traffic. A scheduling scheme works perfectly only when it manages and distributes the resources among the thread hierarchy in a well balanced manner so that all the streaming multiprocessors run with full occupancy. These factors all jointly impact the runtime. Ultimately, the achieved performance for an algorithm is a complicated interaction between a variety of parameters, some determined by the algorithm, some set by developers, and others imposed by the architecture of the particular machine being used. The interactions between these parameters, as well as the impact of each on the algorithm's performance, are often not well understood.

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

As to asymptotic models, Ma et al. [10] designed the Threaded Multi-core Memory (TMM) model, in which a number of classic algorithms are analyzed in terms of both their computational complexity and their memory complexity assuming perfect scheduling [11], [12]. Kirtzic et al. [13] proposed the Parallel GPU Model (PGM), which is essentially an adaption of the Bulk-Synchronous Parallel (BSP) model [14], and equates a superstep in BSP with a function unit of a GPU program. This model does not explicitly model the memory subsystem and assumes uniform cost access to all levels of memory. Nakano [15] proposed the Hierarchical Memory Machine (HMM) model, which consists of multiple Discrete Memory Machines (DMMs) representing shared memory and a single Unified Memory Machine (UMM) representing global memory. The HMM model does consider both shared memory accesses and the grouping of global memory accesses. Haque et al. [16] proposed a Many-core Machine Model (MMM) based on the Graham-Brent theorem, which is quite similar to the TMM model, but does not model the impact of threading for hiding memory latency.

As to calibrated performance models, Hong et al. [17] propose an analytical model to capture an estimate of the cost of memory operations by counting the number of parallel memory requests in terms of memory-warp parallelism (MWP) and computation-warp parallelism (CWP). However, their assumption of no cache misses is not always realistic.

[1] If a thread stalls on a memory operation, some other thread can be scheduled in its place.

Sim et al. [18] extend this MWP-CWP model and present the GPUPerf framework. This framework quantitatively estimates performance along four dimensions: inter-thread instruction-level parallelism, memory-level parallelism, computing efficiency, and serialization effects. These four metrics help to identify performance bottlenecks and suggest what types of optimizations should be done. Ma et al. [2], [19] design an analytic model for memory-limited kernels especially as impacted by cache and various configuration parameters that can be used to tune kernel execution. This model also reflects warp scheduling effect at the thread block level rather than instruction level, thus it is simpler than the model in [18]. Liu et al. [20] describe a general performance model that predicts the runtime of a biosequence database scanning application fairly precisely. Their model incorporates the relationship between problem size and performance, but does not look into microarchitecture-level parameters like how the size of thread blocks and grids[2] influences the runtime. Baghsorkhi et al. [21] measure performance factors in isolation and later combine them to model the overall performance via work flow graphs so that the interactive effects between different performance factors are modeled correctly. Their compiler-based approach still doesn't provide suggestions for run-time configuration of kernels. Govindaraju et al. [22] propose a cache model for efficiently implementing three memory intensive scientific applications with nested loops. It is helpful for applications with 2D-block representations while choosing an appropriate block size by estimating cache misses. He et al. [23] focus on the access patterns of gather and scatter operations, which can suffer from low memory bandwidth utilization, and design a probabilistic cache model to predict cache misses. Zhang et al. [6] 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.

In the present work, we utilize both asymptotic analysis and calibrated performance prediction on many-core GPUs, in effect drawing on the concepts of [10] and [19]. We develop an integrated analytical framework combining both, analyzing algorithm efficiency and predicting the achievable execution time based on a quantification of ***parallelism***, ***latency-hiding***, and ***occupancy***. Within the context of the asymptotic analysis, in addition to the computational complexity expressed in terms of ***work*** T_1 — the total amount of computation, or, in other words, its running time on 1 processor — and ***span***[3] T_∞ — the amount of computation on the critical path or, in other words, the running time on an infinite number of processors, we also consider the memory complexity determined by the number of memory transfers M from slow memory to fast memory as a critical performance measure. In general, GPUs attempt to mask memory transfers to/from slow memory by executing a substantial number of concurrent threads, whereby nominally there is always a set of threads ready for execution (i.e., not waiting on a memory reference to complete). However, one cannot launch arbitrarily large numbers of threads due to limited on-chip resources, which are managed by the scheduler. At the same time, simply seeking a large thread count per thread block may not always provide high occupancy on streaming multiprocessors and therefore cannot guarantee good performance. As a result, we model GPU scheduling mechanisms and use them as a factor bridging the gap between the asymptotic model and calibrated model, between the theoretical performance and real runtime.

Our model is useful in a number of aspects:

1) It is able to identify the performance bottleneck of a particular algorithm, and judge whether the algorithm is more likely to be performance bound by memory accesses or by computation.
2) It predicts performance trends as the problem size (or other parameters) scale up. This can be quite helpful when comparing and ordering different algorithms with various parameter settings.
3) It can explore and reduce the design and configuration space for tuning kernel execution on GPUs. Choosing how to decompose the problem into subproblems and picking the right thread block/grid size for better scheduling constitutes a gigantic search space. The model is able to project the possible runtime given different inputs and prune the space accordingly.
4) It is helpful for identifying performance improvement opportunities along two dimensions, scheduling and algorithm design. Oftentimes, general algorithms may suffer from insufficient parallelism or bad access patterns, or both to a different extent. Sub-optimal configuration of kernel launch can also impede the performance. Guided by the model, algorithms can be designed to maximize the parallelism while at the same time minimizing frequent and irregular long-latency memory accesses. The scheduling scheme should balance among the choices of sub-problem size, thread block/grid size, and also the workload and resources consumed per thread. Increasing threads in a block may reduce the active blocks that can be launched simultaneously so that the occupancy drops as the total number of threads is reduced. However, enlarging the sub-problem size and, for each, assigning a bigger thread block can reduce the passes that are needed to solve the entire problem. This trade-off relation is quantified in our model.
5) It highlights the sub-optimality of existing GPU scheduling schemes in some scenarios. Chasing high occupancy, the current GPU scheduler dispatches thread blocks onto a certain number of streaming multiprocessors in a greedy way depending on the resource usage of each thread block. However, a good occupancy does not necessarily guarantee the best runtime. We illustrate a set of use cases where artificially increasing the requested amount of shared memory results in substantial performance gains.

The paper is organized as follows. Section II describes how the analytical modeling framework is established capturing the characteristics of GPU architectures. Section III provides the analysis of a classic algorithm, all-pairs shortest paths (APSP), under the proposed model. Section IV compares the model prediction with empirical data for validation. Section V illustrates the use of the model to identify unexpected behavior.

[2]In a GPU kernel launch, threads are organized in a number of blocks; thread blocks are organized in a number of grids.

[3]Also called depth, time, or critical-path length in the literature.

The paper ends with some concluding remarks in Section VI.

II. Analytical Model

In this section, we will describe a pair of existing models for algorithm performance on GPUs (developed by our group), the first an asymptotic model and the second a calibrated performance model. We follow this discussion by combining these two models, resulting in a calibrated performance model with greater coverage than either of the previously existing models.

A. Asymptotically Modeling Algorithm Execution Time

Based on the TMM model [10], highly-threaded many-core GPUs can be abstracted as consisting of a number of core groups Q (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 take unit time. 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.[4] 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 calculating M. In addition, $\mathcal{T}$ is an additional parameter for the ***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 TMM model parameters.

TABLE I. TMM model parameters.

	Description
Q	number of core groups
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 $ML/(\mathcal{T}P)$.

B. Calibrated Modeling of Runtime

As a general rule, calibrated performance models differ from asymptotic performance models in several ways. These distinctions might include the explicit inclusion of performance-impacting factors that are only important over some range of the model's input domain (especially, for example, for smaller input sizes) in addition to the scale factors that enable calibrated models to make specific quantitative performance predictions.

The model presented in [19], which we will extend in the subsection that follows, characterizes algorithm performance in terms of the following factors: algorithmic complexity, f_{app}, caching, f_{cache}, and scheduling, f_{sched}. The algorithmic complexity factor is expressed via a function $f_{\text{app}}(\overrightarrow{\text{algo}}, \overrightarrow{\text{inpt}})$, defined in terms of an algorithm parameter vector $\overrightarrow{\text{algo}}$ and an input size vector $\overrightarrow{\text{inpt}}$. $\overrightarrow{\text{algo}}$ includes parameters from the algorithm design and implementation, e.g., size of a sub-block computation, etc. $\overrightarrow{\text{inpt}}$ takes the parameters relevant to the input problem size and/or working set size. The form of f_{app} is, of course, algorithm-specific. It can be intuitively regarded as a general adapter of the model to different problem sizes, algorithms, and even to different implementations of each algorithm.

The caching factor, f_{cache}, reflects the impact of cache misses on runtime. A Boolean indicator A_C is used to encode whether or not the kernel's working set fits in on-chip memory spaces, either shared memory or L1 cache. We complete the cache performance model by expressing the cache factor as a linear combination of execution times that are blended by cache hit rate (r_H) and miss rate (r_M):

$$f_{\text{cache}} = \begin{cases} 1 & \text{if } A_C \\ r_H + r_M \times G & \text{otherwise,} \end{cases} \quad (2)$$

where G reflects the multiplicative slowdown experienced with very low cache hit rates. In principle, one would like to express G in terms of the relative performance of the cache and the global memory. This is essentially the same thing modeled with the parameter L in the TMM model.

The scheduling factor, f_{sched}, captures the performance impact due to the scheduling of thread blocks on streaming multiprocessors.

$$f_{\text{sched}} = \frac{\lceil \frac{B_r}{B_a \times P/Q} \rceil \times B_a \times P/Q}{B_r} \quad (3)$$

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

Here, B_r is the number of requested thread blocks, B_a is the number of active thread blocks on each multiprocessor and P/Q is the number of multiprocessors. Not all requested blocks B_r can be scheduled and executed on all the multiprocessors at the same time. If $B_r > B_a \times P/Q$, multiple passes are needed to consume all the requested blocks of work. From [19], the number of active blocks B_a is described in equation (4) in terms of the shared memory required by the application S_B, the quantity of shared memory on each multiprocessor Z, the number of threads requested per thread block T_r, the processor registers required by the application $R_T \times T_r$, the quantity of registers available per multiprocessor R, the maximally allowed thread blocks B_{max}, and the maximally allowed threads T_{maxMP}.

$$B_a = \min\left(\left\lfloor \frac{Z}{S_B} \right\rfloor, \left\lfloor \frac{R}{R_T \times T_r} \right\rfloor, \left\lfloor \frac{B_{\max}}{P/Q} \right\rfloor, \left\lfloor \frac{T_{\text{maxMP}}}{T_r} \right\rfloor\right) \tag{4}$$

The above expression for f_{sched} in (3) reflects the block scheduling process on the multiprocessors, with the time determined by the multiprocessor with the largest number of blocks assigned to it (expressed via the ceiling function $\left\lceil \frac{B_r}{B_a \times P/Q} \right\rceil$). As we will see below, this yields a distinctive zigzag pattern in the predicted performance as the number of requested blocks is varied. Essentially, this factor encapsulates the occupancy of the GPU.

Overall, the runtime is modeled to be proportional to the product of the three factors as shown below:

$$\textit{Time} \propto f_{\text{app}}(\overrightarrow{\text{algo}}, \overrightarrow{\text{inpt}}) \times f_{\text{cache}} \times f_{\text{sched}} \quad \text{if } A_T \tag{5}$$

where A_T is a Boolean indicator which is true when a sufficiently large number of threads are launched so that the memory latency is assumed to be fully hidden.

C. Extended Model

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

1) The TMM model explicitly includes the impact of memory references on execution time. In particular, the memory complexity explicitly reflects the cost of memory behavior and, at the same time, the caching effect in terms of C — how many memory operations or data accesses can be grouped and Z — how much data can be cached and shared.] This implies that T_P takes over not only the functionality of f_{app} in the model of [19] but also the functionality of f_{cache}.
2) The model of [19] is constrained to the circumstance where there are sufficient threads to mask memory latency. The TMM model has no such constraint, and therefore the condition variable A_T in equation (5) can be eliminated.

The second step in combining the two models is to substitute the appropriate[5] portions of equation (1) into equation (5).

$$\begin{aligned} \textit{Time} &\propto T_P \times f_{\text{sched}} && (6)\\ &\propto \max\left(\frac{T_1}{P}, \frac{ML}{\mathcal{T}P}\right) \cdot \frac{\lceil \frac{B_r}{B_a P/Q} \rceil \cdot B_a P/Q}{B_r} && (7)\\ &\propto \max\left(T_1, \frac{ML}{\mathcal{T}}\right) \cdot \left\lceil \frac{QB_r}{B_a P} \right\rceil \cdot \frac{B_a}{QB_r} && (8) \end{aligned}$$

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

III. Application of the Integrated Analytical Model for Algorithm Analysis

In this section, we pick a classic algorithm — dynamic programming via adjacency matrix — for solving the all-pairs shortest paths problem, as a vehicle for empirically investigating the extended performance model. We develop expressions for the work, T_1, and the number of memory transactions, M, as a function of the problem size. These are then substituted into equation (8).

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 [24] 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):

$$A^2_{ij} = \min_{0 \le k < n} (A^1_{ij}, A^1_{ik} + A^1_{kj}). \tag{9}$$

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

This algorithm was analyzed in the original TMM paper [10]. The work $T_1 = n^3 \lg n$ is the same as with traditional PRAM analysis. To analyze the memory cost, we need to count the number of memory transfers between global memory and shared memory. Algorithms are tailored to highly-threaded, many-core architectures generally by using fast on-chip memory to avoid accesses to slow off-chip global memory, coalescing[6] to diminish the time incurred to access slow memory, and threading to hide the latency of accesses to slow memory. Due to its large size, the entire adjacency matrix is stored on off-chip global memory. Following traditional block-decomposition techniques, the matrix multiplication is performed by dividing the matrix into sub-blocks[7] with dimension S_D such that the total number of sub-blocks is $(n/S_D)^2$, and allowing each thread block on a multiprocessor

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

[6]C reads can grouped in just one memory transfer if consecutive C threads read data from continuous addresses.

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

to operate on an individual sub-block of the data. Individual threads read in the required input sub-blocks, perform the computation of equation (9) for their assigned sub-block, and write the sub-block out to global memory. This happens $\lg n$ times by repeated squaring. There are a total of n^2 elements and each element is read for the calculation of n/S_D other blocks. However, due to the regularity in memory accesses, each block can be read fully coalesced. As a result, the number of memory operations for one matrix multiply is $(n^2/C)(n/S_D) = n^3/(S_D C)$. one B core group, we get B = (Z). Therefore, for $\lg n$ matrix multiplication operations,

$$M = \frac{n^3 \lg n}{S_D C}. \tag{10}$$

Substituting the above expressions for T_1 and M into (8) yields the following:

$$\textit{Time} \propto \max\left(n^3 \lg n, \frac{n^3 \lg n \cdot L}{S_D C \mathcal{T}}\right) \cdot \left\lceil \frac{QB_r}{B_a P} \right\rceil \cdot \frac{B_a}{QB_r}. \tag{11}$$

There is an intrinsic relation between the sub-block dimension, S_D, and the number of requested thread blocks, B_r, according to how the problem is partitioned and assigned by the algorithm. There are $(n/S_D)^2$ total sub-blocks, each assigned to a single thread block, $B_r = (n/S_D)^2$, i.e. $S_D = n/\sqrt{B_r}$.

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

$$\textit{Time} \propto \max\left(n^3 \lg n, \frac{n^2 \lg n \cdot L\sqrt{B_r}}{C\mathcal{T}}\right) \cdot \left\lceil \frac{QB_r}{B_a P} \right\rceil \cdot \frac{B_a}{QB_r}. \tag{12}$$

The expression above is informative in a number of aspects.

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

$$\textit{Time} \propto n^3 \lg n \cdot \left\lceil \frac{QB_r}{B_a P} \right\rceil \cdot \frac{B_a}{QB_r}. \tag{13}$$

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

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

$$\textit{Time} \propto \frac{n^2 \lg n \cdot L\sqrt{B_r}}{C\mathcal{T}} \cdot \left\lceil \frac{QB_r}{B_a P} \right\rceil \cdot \frac{B_a}{QB_r}. \tag{14}$$

In this case, the runtime is predicted to be linear with $\sqrt{B_r}/\mathcal{T}$. This means that when performance is bounded by memory latency, enlarging the sub-block size, S_D, (i.e., effectively reducing the requested number of thread blocks, B_r) or increasing the average thread count per core, $\mathcal{T}$, can reduce the runtime.[8]

3) Considering the impact of the second two terms of (8), note that the number of active blocks, B_a, is determined by the scheduler according to register usage, shared memory usage, and fixed device capability [19]. Performance is maximized when the requested number of blocks, B_r, is an integer multiple of the product of B_a and P/Q, thereby balancing the number of blocks allocated to each multiprocessor and maintaining a high occupancy. According to (3), continuously varying B_r generates runtimes with a zigzag pattern as illustrated in Figure 1. As can be readily observed, the runtime is minimized at regular intervals when B_r happens to be a multiple of 15. As B_r grows to larger values, the influence of B_r on occupancy diminishes.

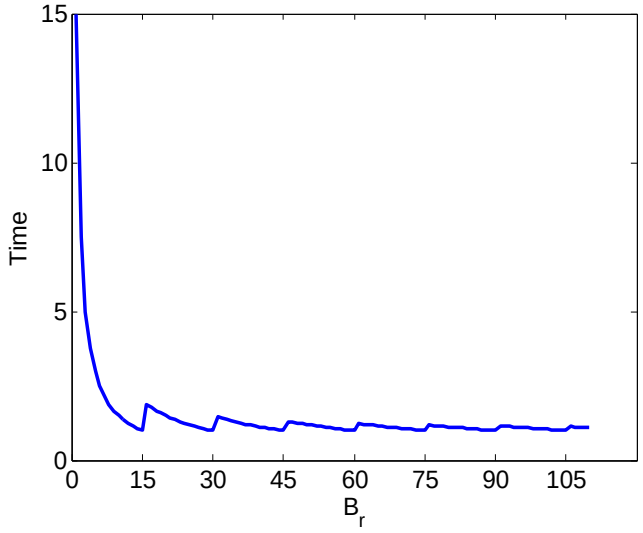

Fig. 1. Execution time variation with requested number of blocks, B_r. For this example, $B_a = 1$, $P/Q = 15$ (e.g., as in an NVIDIA GTX480), and the max term in (8) is artificially set to 1.

IV. Empirical Validation

In this section, we validate the predictions drawn from the analysis above through empirical measurements. We use an NVIDIA GTX480 with 15 multiprocessors, each of which has 32 cores and supports up to 1536 threads sharing the same 48 KB shared memory. We implement the dynamic programming algorithm via adjacency matrix multiplication for solving the all-pairs shortest paths problem, and set up a testbench with which we can vary the following parameter settings: problem size, n, sub-block dimension, S_D, requested blocks B_r, and threads per core, $\mathcal{T}$. The relation between runtime and problem size n has been well studied before and will not be investigated here. Instead, we will focus on

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

verifying the effects of other parameters present in the model and of particular interest to the new extensions to the model: S_D, B_r, and $\mathcal{T}$.

Note that the values of some of the architecture parameters (e.g. Z and C) can be obtained from the specification of the architecture being used. Generally for NVIDIA GPUs, C is 32 if reads of threads are grouped; Z can be configured either to 16 KB or 48 KB for Fermi and later architectures. While the value of architecture parameters, such as the memory latency L, can not be quantitatively determined, however, for a given architecture, these parameters will not change. In (14), C and L only contribute as a scale factor and can be represented by fixed coefficients as shown in (15).

A. Effect of $\sqrt{B_r}/\mathcal{T}$

According to (13) and (14), we infer that runtime should be linear with $\sqrt{B_r}/\mathcal{T}$ when memory bound and stay constant when compute bound. In Figure 2, we fit a linear curve to the measured execution time of several runs, varying the settings for B_r and $\mathcal{T}$ as follows:

$$Time = a_1 \cdot \frac{\sqrt{B_r}}{\mathcal{T}} + a_0 \tag{15}$$

with $a_1 = 0.957$, $a_0 = 53.9$. The horizontal line represents the execution time when the application is compute bound (the empirical support for which is all clustered near the origin, it is plotted across the entire graph for easier visibility). For the curve fit of (15), $r^2 = 0.9916$, showing good linearity. The measured and predicted runtimes align with each other quite well.

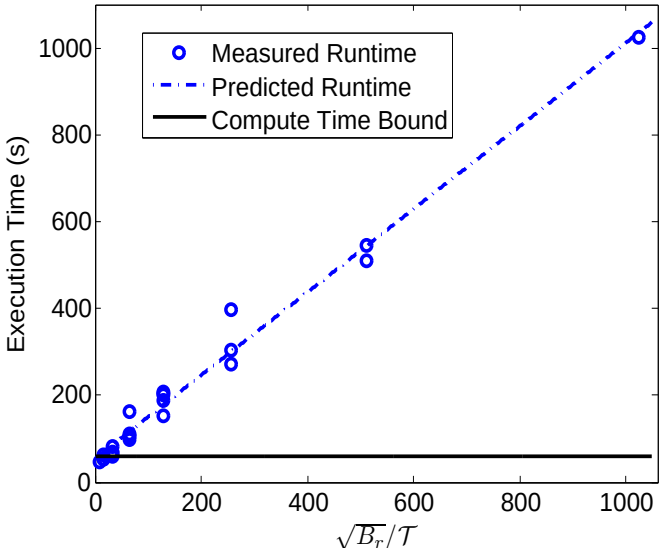

Fig. 2. Runtime and model prediction in terms of $\sqrt{B_r}/\mathcal{T}$ for all-pairs shortest paths problem with 8192 vertices. Measurements are from various runtime configurations of $(B_r, \mathcal{T}, S_D)$, therefore with different B_a. Specifically, $B_r = (n/S_D)^2$, B_a is determined by equation (4).

Given a fixed problem size n that is reasonable on a fixed machine (we solve a graph of 8192 vertices on a GTX480), reducing $\sqrt{B_r}/\mathcal{T}$ will eventually transition the application performance from being constrained by memory latency to being constrained by computation. This observation is illustrated in Figure 2. When we either choose a smaller B_r indirectly by increasing the sub-block dimension S_D, or configure the kernel to run with more threads $\mathcal{T}$, the runtime keeps dropping until a level where it turns flat. This is the point where transition happens, and after which the runtime is bounded by computation and independent of $\sqrt{B_r}/\mathcal{T}$.

B. Effect of $\mathcal{T}$

Next, we move on to investigate the effects of $\mathcal{T}$, the average threads per core, on runtime. According to equations (13) and (14), runtime should be inversely related to $\mathcal{T}$ when $\mathcal{T}S_D < L/C$ and stay constant if $\mathcal{T}S_D > L/C$. Figure 3 illustrates this relationship fairly clearly. At small values of $\mathcal{T}$, runtime drops drastically as we increase $\mathcal{T}$. At the same time, the curves with larger S_D (smaller B_r) are more steeply sloped and flatten out for a lower value of $\mathcal{T}$. Runtime for the trials using $S_D = 64$, 32, and 16 converge to the same (flat) level at $\mathcal{T} = 8$, 16, and 32, respectively. All these observations are consistent with and can be explained by the model. In the range where $\mathcal{T}$ is low, latencies are not well hidden. When $\mathcal{T}$ gets big enough so that latencies are completely hidden, further increases in $\mathcal{T}$ do not bring any marginal benefits in terms of runtime. Larger S_D enables a smaller value of $\mathcal{T}$ as it is the product of both S_D and $\mathcal{T}$ that matters for latency hiding purposes.

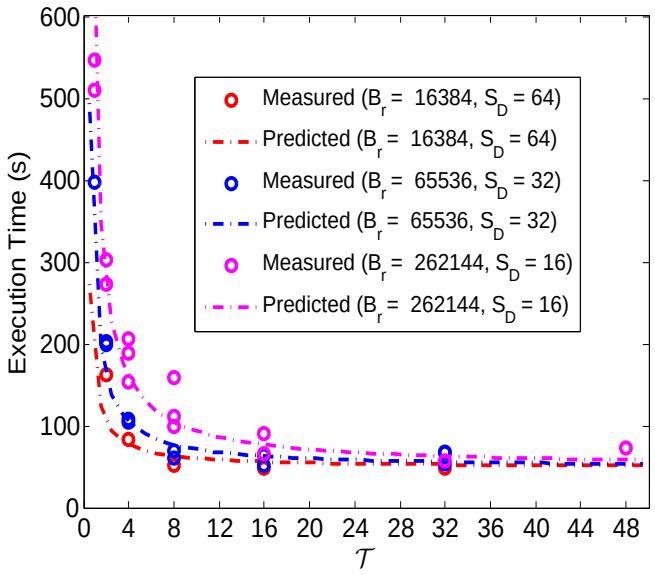

Fig. 3. Empirically measured and model predicted runtimes in terms of $\mathcal{T}$ for all-pairs shortest paths problem with 8192 vertices. Measurements are from various runtime configurations of $(B_r, \mathcal{T}, S_D)$, therefore with different B_a. Specifically, $B_r = (n/S_D)^2$, B_a is determined by equation (4).

C. Effect of B_r

We continue by examining the effects of B_r on the application runtime. For a fixed value of $\mathcal{T}$, again, according to equations (13) and (14), runtime should increase with $\sqrt{B_r}$ when limited by memory latency and stay constant when compute bound. Figure 4 shows how runtime changes with B_r for several distinct values of $\mathcal{T}$. Due to the sub-blocking

mechanism, integral division of the matrix restricts the dimension of sub-blocks, S_D, to be processed per kernel, therefore the value options for B_r are limited. Nonetheless, we still see a reasonably good alignment between the model's predictions and the empirical measurements from the implementation for the range of $\mathcal{T}$ attempted. Curves for the larger values of $\mathcal{T}$, for example $\mathcal{T} = 16$ or 32, tend to converge to the flat level implied by sufficient latency hiding.

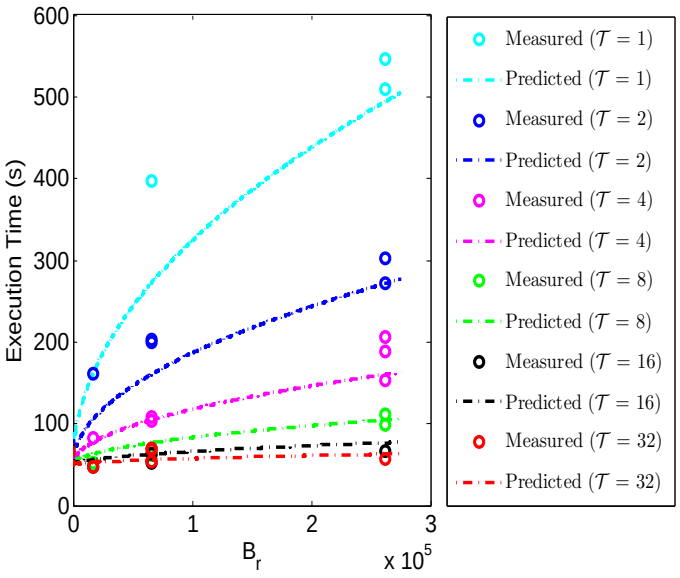

Fig. 4. Empirically measured and model predicted runtimes in terms of B_r for all-pairs shortest paths problem with $n = 8192$ vertices. Measurements are from various runtime configurations of $(B_r, \mathcal{T}, S_D)$, therefore with different B_a. Specifically, $B_r = (n/S_D)^2$, B_a is determined by equation (4).

V. Discovering Unexpected Behavior

While the model predictions described above are not always perfect (see, for example, data points in Figure 4 for very small values of $\mathcal{T}$), generally they do a very good job of explaining how performance trends with various performance-impacting factors. We will next illustrate the use of the model to discover an unexpected condition, in which the measured empirical performance was substantially different than the model prediction, and how that exposes additional uncertainty in realized performance of GPU applications in practical settings.

When launching a kernel on the GPU, the programmer specifies a configuration of that kernel, which includes things such as count of thread blocks, threads per thread block, etc. Other parameters, such as the number of active blocks, are automatically set as a function of the explicitly provided configuration according to equation (4). For an specific example run ($n = 8192$, $S_D = 32$, $B_r = 65,536$, $\mathcal{T} = 4$, $B_a = 4$), the measured execution time of 404 s was 4 times longer than the predicted execution time of 105 s. As part of our investigation into this anomaly, we artificially increased the shared memory requested by the application, thereby coercing it to use a B_a of 1 instead of the automatically determined value of 4. When we ran this altered version of the application (which we verified still provided the correct result), the execution time was 108 s, much more in line with the model's prediction.

The above anomaly occurred more often than just in this individual case. Figure 5 shows execution time vs. $\mathcal{T}$ for several cases of automatically determined numbers of active blocks and artificially lowered numbers of active blocks ($B_a = 1$ or 2). In many cases, requesting more memory than was truly needed resulted in substantial performance gains.

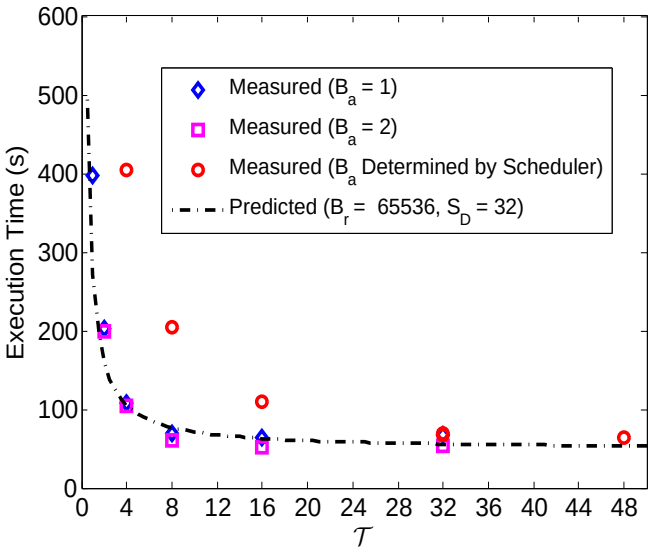

Fig. 5. Empirical measures of all-pairs shortest paths runtime for scenarios when $B_a = 1$, $B_a = 2$ and B_a is determined automatically by the scheduler. The APSP problem with 8192 vertices is divided into sub-blocks of dimension 32.

While we do not have a satisfactory explanation for the execution time realized when B_a is at its default value, we do conclude that there is sufficient uncertainty inherent in the performance achievable on modern GPUs that the use of well understood performance models can at the very least help to identify circumstances where the application is not performing as it should. The scheduler works in a way that the number of active blocks scheduled on each multiprocessor cannot be directly controlled. Yet, as illustrated in Figure 5, there are circumstances where it is clearly beneficial to the application to be able to control the active blocks scheduled. As illustrated by our experience, however, it can be indirectly changed by altering the shared memory requested by each kernel.

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

VI. Conclusion

In this paper, we bridge the gap between asymptotic models and calibrated models, between theoretical performance predictions and real execution time, for performance analysis of algorithms on highly-threaded many-core GPUs. We develop an integrated analytical framework extending two existing

models, combine them by modeling GPU scheduling mechanisms and incorporating both the computation complexity and memory complexity as critical performance-impacting measures. By doing so, our analytical framework is able to capture the parallelism, latency-hiding, and occupancy together in one model, reflecting performance bottlenecks, reducing the configuration space for kernel execution, and predicting achievable execution times as well as how execution time will trend as the various parameters scale.

We analyze a classic algorithm — all-pairs shortest paths — under this analytical framework, and compare the analytical results with empirical results. This comparison indicates that our model is effective at explaining empirical performance for highly-threaded many-core GPUs. In particular, it accurately predicts the effect of changing the sub-block dimension, thread count, requested blocks, and local memory size on the running time of algorithms. In addition, the model points to at least one circumstance in which the occupancy decisions automatically made by the scheduler are clearly sub-optimal.

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

[4] 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.

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

[6] 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.

[7] R. D. Chamberlain, M. A. Franklin, E. J. Tyson, J. H. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. B. Shands, and N. Singla, "Auto-pipe: Streaming applications on architecturally diverse systems," *Computer*, vol. 43, no. 3, pp. 42–49, 2010.

[8] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain, "Deadlock avoidance for streaming computations with filtering," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*. ACM, 2010, pp. 243–252.

[9] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh, "Scalable framework for mapping streaming applications onto multi-GPU systems," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 1–10, 2012.

[10] 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.

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

[12] L. Ma, K. Agrawal, and R. D. Chamberlain, "Analysis of classic algorithms on GPUs," in *Proc. of the 12th ACM/IEEE Int'l Conf. on High Performance Computing and Simulation (HPCS)*, 2014.

[13] 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.

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

[15] K. Nakano, "The hierarchical memory machine model for GPUs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, 2013.

[16] 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 Symposium*, 2013.

[17] 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.

[18] J. Sim, A. Dasgupta, H. Kim, and R. W. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *Proc. of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012, pp. 11–22.

[19] 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.

[20] 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.

[21] 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 (PPoPP)*, 2010, pp. 105–114.

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

[23] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in *Proc. of ACM/IEEE Supercomputing Conf.*, 2007.

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