Transit: A Visual Analytical Model for Multithreaded Machines

Ang Li[*,†], Y.C. Tay[†], Akash Kumar[†], and Henk Corporaal[*]

[*]Eindhoven University of Technology, Eindhoven, Netherlands

[†]National University of Singapore, Singapore

ang.li@tue.nl, *dcstayyc@nus.edu.sg*, *akash@nus.edu.sg*, *h.corporaal@tue.nl*

ABSTRACT

With the extraordinary growth of cores and threads in today's multithreaded machines, analyzing and tuning the performance of such platforms becomes a challenging task. In this paper, we propose an intuitive and visualizable model for analyzing the performance of contemporary highly concurrent multithreaded machines. Based on flow balancing between service demand and service supply of the memory system, the model draws an intuitive figure to characterize machine state, identify bottlenecks and determine optimization directions. The tractability of the model is highlighted as it only requires two parameters from the workload. Our model achieves 90% and 83% prediction accuracy for computation throughput on Fermi and Kepler GPUs over the 16 applications from Rodinia benchmark.

Keywords

Performance modeling; Multithreaded machine; GPUs; Performance optimization

1. INTRODUCTION

Moore's Law has continued to show promise, but the end of clock-frequency scaling for uniprocessors has driven mainstream computation towards the multi-core era [1]. Multicore processors offer enormous computing power, but insufficient exploitable parallelism and long-latency remote communication, typically off-chip memory access, restrict the attainable performance [2]. Consequently, *multithreading* [3] has also been proposed as an effective solution. It raises processor utilization through thread-level parallelism (TLP) and hides memory delay via fast context switching. Later, with the rapid growth of cores in a processor, the number of threads has increased dramatically. Nowadays, a single GPU chip encapsulates up to 2,880 scalar cores and can accommodate over 30,000 active threads simultaneously.

Obviously, tuning performance for such massively multithreaded platforms becomes a difficult challenge. Although modest speedup could be attained through basic functional porting, programmers have to spend significant time and effort to identify and alleviate the system bottlenecks before fully extracting the hardware potential. This is especially the case when little is known about the underlying implementation of the target machine. Therefore, many programmers and designers have to search exhaustively via profilers or simulators in a huge design space, or rely entirely on empirical guidelines.

Analytical model offers an alternative approach. It either models a particular architecture that requires numerous parameters to grasp detailed machine features in order to predict performance precisely, e.g. [4, 5], or it models a general machine that is easy to understand and manipulate so as to highlight new behaviors, explain observed phenomena and derive intuition, e.g. [6, 7].

This paper falls into the second category – we propose a high-level, visualizable and throughput-oriented analytical model for general multithreaded machines. Based on flow balancing between service demand and supply of the memory system, our model clearly describes machine state, locates performance bottlenecks and indicates optimization directions. The major objective is to provide a visualizable modeling tool for gaining insight and deriving intuition.

2. TRANSIT MODEL

In this section, we present the **transit model**. We first describe how the components of the multithreaded machine system are organized. We then present how to construct the model and how to draw transit figures. Finally, we summarize the input and output of the model.

2.1 Model Components

In the transit model, a computer machine is decomposed into a *computation* and a *memory system*, denoted as **CS** and **MS**. CS refers to computation units including multiprocessors, coprocessors and special accelerators while MS refers to the memory hierarchy including local cache, shared cache, off-chip DRAM, etc. For flexibility, the scope of MS can be scaled along the memory hierarchy, from the top register level to the bottom hard-disk or Internet level, in a different context. For example, if MS refers to the off-chip DRAM, then CS refers to the entire processor chip.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HPDC'15, June 15–20, 2015, Portland, Oregon, USA.

Copyright © 2015 ACM 978-1-4503-3550-8/15/06 ...$15.00.

http://dx.doi.org/10.1145/2749246.2749265.

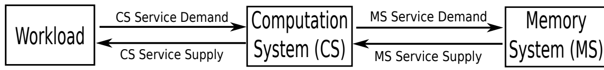

Figure 1: System Organization. A computer system is partitioned into double layers – MS stands isolated from the workload.

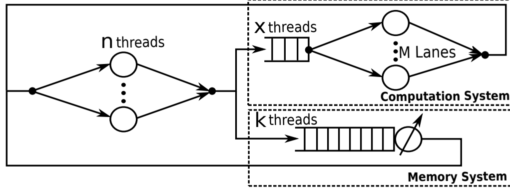

Figure 2: Multithreaded Machine Model

From the application's standpoint, it is CS that accomplishes the desired jobs; MS, however, plays an assistant role since the application logic does not impose any data movements among various memories – most of the time, the application logic postulates the memory space to be flat and unified. Therefore, the CS throughput is often viewed as the primary performance metric while MS throughput remains the secondary.

As depicted in Fig.1, MS stands isolated from the workload. Since it is CS that executes the user logic, the workload has service demand over CS, which is the theoretical attainable throughput of CS . However, due to performance bottlenecks inside the machine, the actual service supply is less or equal to such service demand. Meanwhile, CS requires MS to store the necessary data, so CS has service demand upon MS. Similarly, the actual service supply of MS to CS is less or equal to the service demand of CS.

2.2 Model Construction

With the system organization as a preamble, let us proceed to the model. As shown in Fig.2, the multithreaded machine is modeled as an *interactive queuing network* [8, 9], a special case of *closed networks* [10]. The reason for being "*closed*" is that the total number of threads is usually dictated by the availability of hardware resources or application configuration; while a new thread is only initiated when an in-flight thread terminates. The memory hierarchy is modeled as as an *aggregate queuing system* and the computation system is modeled as a *single-queue-multiple-server network*, inside which each server indicates a unique *execution lane* (also known as *thread slot* [11] or *logical processor* [12]) that is capable of performing a unit-cost operation in a single cycle or time slot. The n active threads are regarded as the users of the two systems, and is thus postulated to be independent of each other. Meanwhile, we also assume that the workload for each thread is roughly homogeneous, just like the GPU's single-instruction-multiple-threads (SIMT) kernels. However, this assumption could be relaxed if we emphasize average-value-analysis (AVA) [9].

A thread has two states: *thinking* and *waiting* by following the interactive model's terminology. It is thinking when being processed in CS. After an average of Z cycles[1], the thinking thread aborts CS and proposes a memory request. The thread is then suspended in MS for L cycles.

[1] Z is also known as *arithmetic intensity* [13] of the host application, which is a ratio of computation operations to memory operations.

Table 1: Symbol Table

Symbol	Meaning
n	Total threads in the machine
k	Threads in MS
x	Threads in CS
$f(k)$	Service supply of MS to CS
$g(x)$	Service demand from CS to MS
Z	Arithmetic intensity (thinking time)
R	Maximum MS throughput
M	Width of concurrent execution lanes
π	CS transition point
δ	MS transition point
L	Average memory access latency

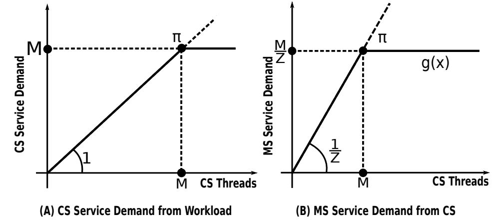

Figure 3: Computation System: Z acts as a scaling factor that transforms the service demand of the computation system to the service demand on the memory system.

Upon fulfillment of the memory request, the thread exits MS and enters CS again, starting a new *turnaround* ($Z + L$ cycles). Before actually being processed in CS, a thread might buffer in a waiting queue. It is assumed that both the waiting queue and the memory queue are sufficiently large to hold all the pending and outstanding threads. This assumption is justifiable as when threads are being blocked, it is equivalent to them waiting in an abstract queue.

Consider the CS service demand in Fig.1, if the throughput of a single thread being executed in one lane during a cycle is normalized as one unit, then for an M-lane system with x threads (Fig.2), the throughput is:

$$G(x) = \begin{cases} x & for\ x < M \\ M & for\ x \geq M \end{cases} \quad (1)$$

Such a shape (Fig.3-A) has been confirmed by several existing works on both multithreaded CPUs [14] and GPUs [15]. Now consider how CS service demand can be transformed to MS service demand in Fig.1. With computation intensity Z, we know that for each Z cycles on average, a memory fetch is prompted. Therefore, the MS service demand (from CS), or the average number of memory fetches with x threads in CS (Fig.2) is

$$g(x) = \begin{cases} \frac{x}{Z} & for\ x < M \\ \frac{M}{Z} & for\ x \geq M \end{cases} \quad (2)$$

as shown in Fig.3-B. Due to dependency, if such demand could not be fulfilled by MS, performance of CS suffers. Here, $\boldsymbol{Z}$ acts as a *scaling factor* that transforms CS service demand to MS service demand. We mark the special point $x = M$ as the *transition point* ($\boldsymbol{\pi}$) of CS, beyond which CS saturates.

For MS, the service supply throughput in Fig.1 is generally similar to Fig.4-A: the beginning phase is nearly linear it is a closed network [9]; the ending phase flattens out as

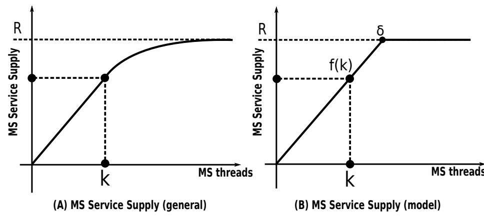

Figure 4: Memory System: the transition region is aggregated as a transition point.

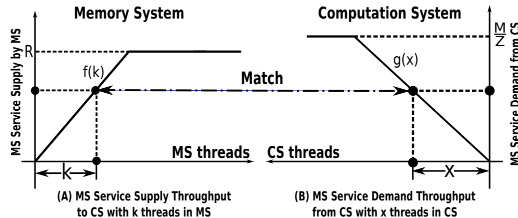

Figure 5: Memory system service supply (A) and demand (B). Note, (A) is the same as Fig.4-B. (B) is obtained by reversing the horizontal axis direction of Fig.3-B.

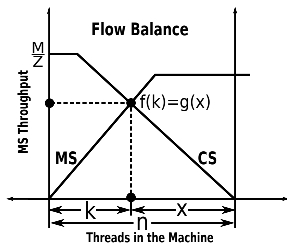

Figure 6: Transit Figure: the equilibrium between service demand and service supply of the memory system. It implies the current machine state: within the total n threads, k of them are in the memory system and x are in the computation system.

throughput approaches bottleneck capacity. However, for tractability, it is also modeled as a roofline shape (Fig.4-B). We argue that this abstraction is already sufficient to capture the characteristics of MS since only the transition region is aggregated as a *transition point* ($\boldsymbol{\delta}$). In fact, the roofline like MS throughput function has also been observed in real machines [16].

If we **reverse** the horizontal axis direction of MS service demand throughput function $g(x)$ (Fig.3-B), it becomes a figure like Fig.5-B. Now focus on MS, we have its *service supply curve* $\boldsymbol{f(k)}$ by itself (Fig.5-A) and *service demand curve* $\boldsymbol{g(x)}$ imposed by the CS (Fig.5-B). Based on the *flow balance property* [10], in a steady state of the machine,

$$\boldsymbol{f(k) = g(x)} \tag{3}$$

Therefore, if the two figures are integrated as shown in Fig.6, their intersection is just the **equilibrium between service demand and supply** (or MS *inflow* and *outflow* [9]), which is exactly the current throughput of MS. Consequently, the CS throughput is $\boldsymbol{Zr}$. We label this visualization **transit figure** because it clearly illustrates the present bottlenecks of the machine and the corresponding optimization directions that are effective to mitigate or remove the bottlenecks.

2.3 Model Input & Output

Input – As aforementioned, in the transit model there are three architecture-related parameters R, L, M and two workload-related parameters Z and n (see Table.1). As the raw memory latency is generally very difficult to change, in the following optimization section, L is viewed as constant. The tuning of the other four parameters and their impact on the shape of the transit figures are shown in Fig.5.
Output – Given the five input parameters, the transit figure can show the tendency of MS throughput and CS throughput. In order see them directly from the figure, we propose the following principles:

- **Principle 1:** If the intersection of $f(k)$ and $g(x)$ goes up, then MS throughput increases.
- **Principle 2:** If the intersection goes up and Z is unchanged, then CS throughput increases.
- **Principle 3:** If Z is increasing and the intersection is on the right side of CS transition point, then CS throughput increases.

In the following subsection, we show how these principles can be leveraged to rapidly locate bottlenecks and determine optimization directions.

2.4 Performance Bound

In this subsection, we describe four types of performance bound. For each bound, we show the effective optimization approaches. Since CS throughput is the primary measure, we only discuss those approaches that can increase CS throughput.

2.4.1 Thread Bound

Identify – Thread bound addresses the state of the machine where insufficient threads are allocated. Therefore, neither CS nor MS can reach its maximum throughput: in the transit figure, the MS throughput function $f(k)$ and CS throughput function $g(x)$ intersect at their sloping parts, as shown in Fig.11.
Optimization – Based on Principle 2, to increase CS throughput, we could raise the intersection point of $f(k)$ and $g(x)$. To achieve this goal, we could push $g(x)$ to the right by enlarging thread volume n (via Fig.7), as described in Fig.11. Or based on Principle 3, we could also increase arithmetic intensity Z (via Fig.8) as the intersection is on the right side of the CS transition point π. The transit figure is shown in Fig.12.

2.4.2 Memory Bound

Identify – Memory bound models the state of the machine when the memory system obtains its maximum throughput: $f(k) = R$. It indicates that in the transit figure (Fig.13), $f(k)$ and $g(x)$ intersect at the flat part of $f(k)$ but the sloping part of $g(x)$.
Optimization – Obviously, tuning n does not affect the height of the intersection or Z. The feasible ways seem to lower the sloping part of $g(x)$ or lift the flat part of $f(k)$, which corresponds to upgrading Z (via Fig.8) and R (via Fig.9). The transit figures are shown in Fig.13 and Fig.14.

2.4.3 Computation Bound

Identify – Computation bound is the state of the machine that $g(x) = M/Z$, which implies that in the transit figure (Fig.15), $f(k)$ and $g(x)$ intersect at the flat part of $g(x)$ but at the sloping part of $f(k)$.

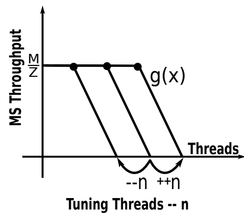

Figure 7: $\boldsymbol{n}$ is the thread volume in the machine. It dictates the relative position of $g(x)$.

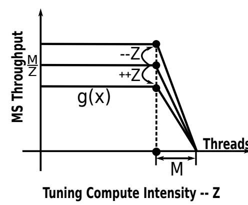

Figure 8: $\boldsymbol{Z}$ is the arithmetic intensity. It determines the scaling factor of $g(x)$.

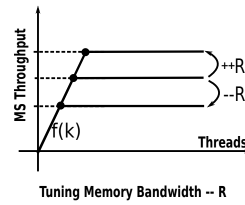

Figure 9: $\boldsymbol{R}$ is the maximum sustainable MS throughput. It determines the altitude of $f(k)$.

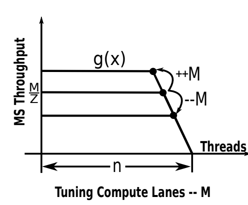

Figure 10: $\boldsymbol{M}$ is the width of concurrent execution lanes. It determines the altitude of $g(x)$.

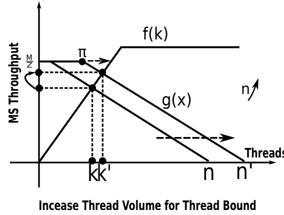

Figure 11: Thread Bound. $f(k)$ and $g(x)$ intersect at their sloping parts. If we slide $g(x)$ to the right by increasing n (via Fig.7) and draw a new curve, with the intersection goes up, by Principle 1, MS throughput increases. Meanwhile, as Z is unchanged, with Principle 2, CS throughput also increases. This method is effective until π touches $f(k)$ as then the machine becomes computation bound.

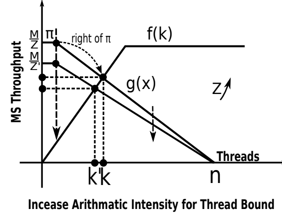

Figure 12: Thread Bound. $f(k)$ and $g(x)$ intersect at their sloping parts. If we tilt $g(x)$ by increasing arithmetic intensity Z (via Fig.8), as the intersection is on the right of π, with Principle 3, CS throughput is essentially rising albeit MS throughput is dropping. This approach fails when π reaches $f(k)$. At that moment, the machine becomes computation bound.

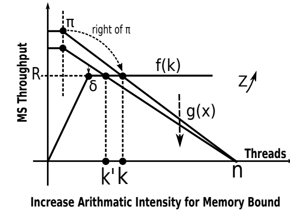

Figure 13: Memory Bound. $f(k)$ and $g(x)$ intersect at the flat part of $f(k)$ but the sloping part of $g(x)$. Since the intersection is on the right side of π, based on Principle 3, we can enhance CS throughput by increasing Z (via Fig.8). Since the height of the intersection is unchanged, by Principle 1, MS throughput keeps constant. The machine becomes thread bound when $g(x)$ coincides δ.

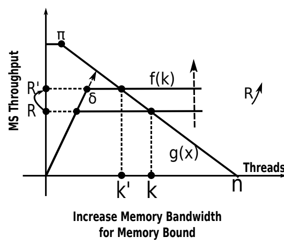

Figure 14: Memory Bound. $f(k)$ and $g(x)$ intersect at the flat part of $f(k)$ but the sloping part of $g(x)$. Based on Principle 2, we can raise the intersection point to increase CS throughput, which is equivalent to lifting the flat part of $f(k)$, or increasing R (via Fig.9). With Principle 1, it also promotes MS throughput. This approach is effective until δ arrives at $g(x)$, at which the machine becomes thread bound.

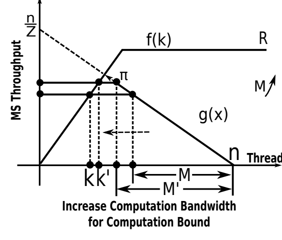

Figure 15: Computation Bound. $f(k)$ and $g(x)$ intersect at the flat part of $g(x)$ but the sloping part of $f(k)$. This is already a good state as all M lanes are leveraged albeit some memory bandwidth are wasted. To further increase CS throughput, we can lift the flat part of $g(x)$ by increasing M (via Fig.10) based on Principle 2. As the intersection goes up, by Principle 1, MS throughput also increases. This approach fails when π touches $f(k)$.

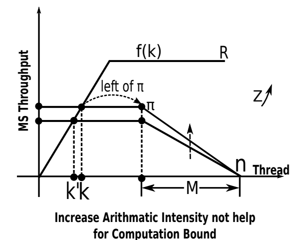

Figure 16: Computation Bound. $f(k)$ and $g(x)$ intersect at the flat part of $g(x)$ but the sloping part of $f(k)$. Here, increasing Z (via Fig.8) does not raise CS throughput because the interaction falls on the left side of the CS transition point π, which violates Principle 3. Also, since the height of the intersection is unchanged, by Principle 1, MS throughput keeps the same.

Optimization – Generally, this is the ideal state of the machine since CS throughput is already the maximum. However, we can still obtain a higher CS throughput by increasing M (via Fig.10), as shown in Fig.15. Note, it is not profitable to increase Z (via Fig.8), because in the scenarios of computation bound, the intersection point is on the left part of CS transition point π, which does not fulfill the requirement of Principle 3, see Fig.16.

2.4.4 Capacity Bound

Identify – Capacity bound describes a very special state of the machine when *workload balance* equals *machine balance* [17]. It requires $M/Z = R$, which is also the ridge point of the Roofline model [6]. In this condition (Fig.17), both $f(k)$ and $g(x)$ intersect at their flat parts. Here, the thread volume n is no longer important if it is sufficient to saturate both CS and MS.

Optimization – This is the **optimal state** for algorithm-architecture or software-hardware codesign regarding thread

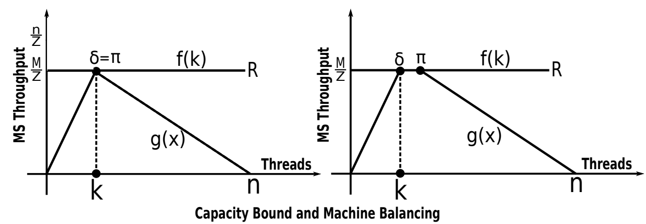

Figure 17: Capacity Bound or **Machine Balance**. This is the optimal case as both CS and MS attain its best performance. Meanwhile, due to capacity bound, some threads may be idle.

parallelism as both CS and MS bandwidth are fully leveraged ($f(k) = R, g(x) = M/Z$). Therefore, no more optimization is required. To break the balance and further improve performance, several factors have to be tuned simultaneously (e.g. M, n and R at the same time).

3. EXPERIMENT

We validate the transit model on a Fermi (Tesla-C2075) and a Kepler GPUs (GTX-690). Although we test on GPUs, the model can be applied on other multithreaded machines, e.g. UltraSPARC T2, Intel Xeon Phi, etc. The Rodinia Benchmark [18] is exploited for validation.

First we collect the machine-related input parameters L, M, R by profiling $f(k)$ and $g(x)$ via microbenchmarking. We view off-chip DRAM as MS. To plot $f(k)$, a CUDA version Stream Benchmark [19] is refined and utilized. In terms of $g(x)$, a microbenchmark is developed based on the method proposed by [20]. Due to space limitation, we cannot display the plots in the paper. However, we show the measured coordinates of δ and π, which can be used to determine $f(k)$ and $g(x)$ uniquely: For Tesla C2075, δ(SP) = (1536, 8.93), δ(DP) = (768, 9.29), π(SP) = (576, confirmed by [20], 24.8), π(DP) = (384, 14); For GTX690, δ(SP) =(2048, 16.25), δ(DP) = (1024, 18.86), π(SP) = (2048, 80), π(DP) = (384, 8). SP stands for single precision while DP stands for double precision. The units are thread and GB/s.

Then we need the workload-related parameters Z and n. For simplicity, the CUDA command-line profiler is leveraged to gather and calculate. The equation is shown below:

$$\begin{cases} n = & occupancy * max_resident_threads \\ Z = & instruction_issued/(dram_read + dram_write) \end{cases}$$

where the symbols on the right side are the names of the profiler counters.

We compare the predicted computation/memory throughput with the values measured by the profilers. A script is developed to gather n and Z, and plot the corresponding transit figure automatically. The results are shown in Fig.18 (for Tesla C2075) and Fig.19 (for GTX690).

As can be seen, for a majority of the applications, the dark star (measured memory throughput) is near the intersection (predicted by the model) except *gaussian, lavaMD* and *particlefilter*. We find that *gaussian* and *particlefilter* exhibit a very irregular memory access pattern so the actual memory throughput function $f(k)$ is far poorer than the ones we profiled. *lavaMD* is the only application that adopts double precision in Rodinia. Also, in the native machine code (SASS) generated by *cuobjdump*, we find that most of the stores are 128 bit-width while most of the loads are 64 bit-width. This explains why such few threads can generate extraordinary memory access performance. Finally, note that the abscissa value of δ(SP) is 1536 for Tesla C2075, which is also the maximum allowable threads per SM. This explains the linear behavior of $f(k)$ for most applications.

Overall, using the computation throughput (PCT & RCT in Fig.18) as the metric, our model achieves 90.4% prediction accuracy for Rodinia benchmarks on Tesla C2075 and 82.6% on GTX690 without the three outliers (*gaussian, lavaMD* and *particlefilter*). The major reason for Kepler showing a poorer accuracy than Fermi is the deviation of the profiled $f(k)$, as we did not consider coalescing degree to keep the model simple and general. However, Kepler is much more sensitive to memory efficiency than Fermi as the core number in Kepler SM is much larger than Fermi (192 vs. 32).

4. RELATED WORKS

Many performance models have been proposed for multithreaded machines [21, 22, 7, 23]. Saavedra-Barrera et al. [21] set up a Markov-Chain to yield a formula for processor efficiency with respect to the number of threads. They characterize three operating regimes: linear (efficiency being proportional to thread volume), transition and saturation (efficiency depending only on remote reference rate and switch cost). The destructive impact on cache due to multithreading is also involved. Agarwal [22] presents an analytical model for a multithreaded machine that covers cache interference, interconnection network contention and context-switching overhead. He concludes that two to four threads are already sufficient to yield full processor utilization if the working-set size is much smaller than caches. Guz et al. [7] propose an analytical model targeting the trade-off between thread volume and the effectiveness of shared cache. A performance valley is identified between the cache efficiency zone and multithreaded efficiency zone for applications that are sensitive to cache efficiency. Chen et al. [23] focus on shared cache contention of multithreaded machines and construct a stochastic model based on circular sequence profiling of the threads.

All of these models, however, predominately focus on the temporal behavior of a *typical thread* or *average thread* (per-thread temporal behavior). Our model stays different in that it stresses on the spatial distribution of the thread population at a general state (all-thread spatial behavior). Meanwhile, most of the existing models (e.g. [4, 5]) are aimed at time prediction, so are mostly devoted to the precise modeling of low-level details. Such an effort requires large amount of parameters and the model itself can be time consuming to learn and implement. In comparison, our model is more straightforward as it focuses on providing high-level intuition and is visualizable as an intuitive figure.

The model that is most similar to the transit model is the Roofline model [6]. In comparison, both models utilize a roofline shape to describe system throughput. The major difference is that the transit model focuses specially on multithreaded machines. We add the thread volume, which is probability the most crucial parameter for a multithreaded machine, as a new dimension and emphasize the spatial distribution of the threads – the machine state. Secondly, in contrast to the Roofline model which attributes all optimizations as the impact to *operation intensity*, we separate a multithreaded machine into two parts so that each of them can be profiled, varied and analyzed independently.

5. CONCLUSIONS

This paper has introduced an intuitive, flexible and visualizable model for analyzing the performance of modern highly concurrent multithreaded machines. Motivated by the observation that the spatial distribution of threads between the computation and memory systems can also describe machine state and the fact that throughput performance is mostly proportional to the number of internal threads before saturation, we construct a **cross-roofline** like model called **transit** to illustrate the present machine state, identify performance bottlenecks and provide optimization intuition.

6. ACKNOWLEDGMENTS

This work was supported by Singapore Ministry of Education Academic Research Fund Tier 1 (No.R263-000-B02-112). The authors would like to thank Marissa E Kwan Lin from NUS for her useful comments on the draft paper.

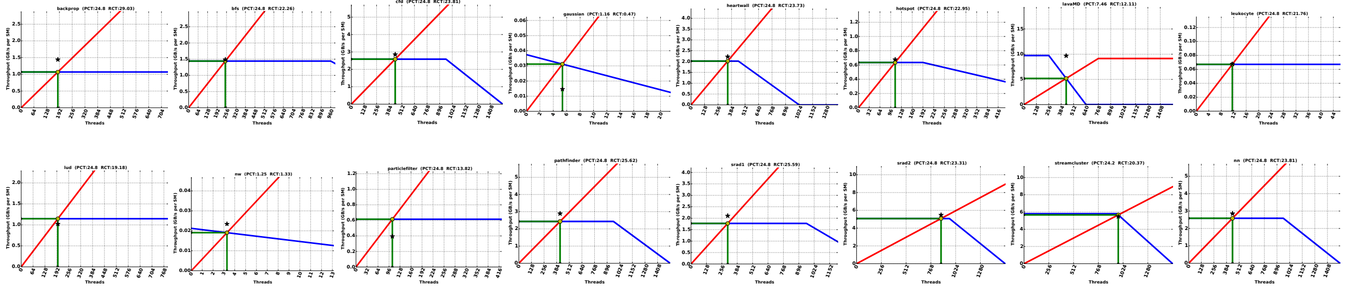

Figure 18: Testing Rodinia Benchmarks on NVIDIA Tesla C2075. The star denotes the measured memory throughput. "PCT" is the predicted computation throughput (GFLOPS) while "RCT" is the real measured computation throughput (GFLOPS).

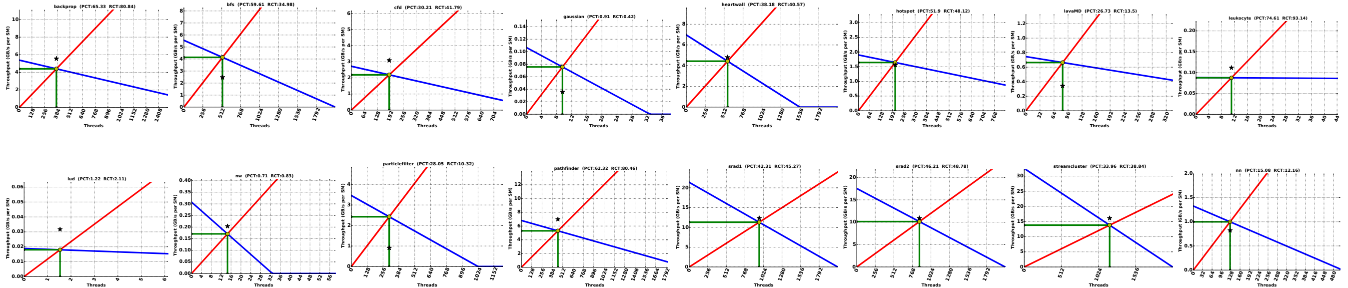

Figure 19: Testing Rodinia Benchmarks on NVIDIA Kepler GTX690. The star denotes the measured memory throughput. "PCT" is the predicted computation throughput (GFLOPS) while "RCT" is the real measured computation throughput (GFLOPS).

References

[1] S. W. Keckler et al. "GPUs and the future of parallel computing". In: *Micro, IEEE* 31.5 (2011), pp. 7–17.

[2] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[3] R. A. Iannucci. *Multithreaded computer architecture: A summary of the state of the art*. Springer, 1994.

[4] D. J. Sorin et al. "Analytic Evaluation of Shared-memory Systems with ILP Processors". In: *Proc.ISCA*. IEEE Computer Society, 1998, pp. 380–391.

[5] S. Hong and H. Kim. "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness". In: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, pp. 152–163.

[6] S. Williams, A. Waterman, and D. Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[7] Z. Guz et al. "Many-core vs. many-thread machines: stay away from the valley". In: *Computer Architecture Letters* 8.1 (2009), pp. 25–28.

[8] P.-S. Chen. "Queueing network model of interactive computing systems". In: *Proceedings of the IEEE* 63.6 (1975), pp. 954–957.

[9] Y. C. Tay. "Analytical performance modeling for computer systems". In: *Synthesis Lectures on Computer Science* 4.3 (2013), pp. 1–141.

[10] E. D. Lazowska et al. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[11] H. Hirata et al. "An elementary processor architecture with simultaneous instruction issuing from multiple threads". In: *ACM SIGARCH Computer Architecture News*. Vol. 20. 2. ACM. 1992, pp. 136–145.

[12] K. Hwang. *Advanced computer architecture*. Tata McGraw-Hill Education, 2003.

[13] S. W. Williams. *Auto-tuning performance on multicore computers*. ProQuest, 2008.

[14] G. T. Byrd and M. A. Holliday. "Multithreaded processor architectures". In: *Spectrum, IEEE* 32.8 (1995), pp. 38–46.

[15] Y. Zhang and J. D. Owens. "A quantitative performance analysis model for GPU architectures". In: *High Performance Computer Architecture, 17th International Symposium on*. IEEE. 2011, pp. 382–393.

[16] J. Nieplocha et al. "Evaluating the potential of multithreaded platforms for irregular scientific computations". In: *Proceedings of the 4th International Conference on Computing Frontiers*. ACM. 2007, pp. 47–58.

[17] J. D. McCalpin. "A survey of memory bandwidth and machine balance in current high performance computers". In: *IEEE TCCA Newsletter* (1995), pp. 19–25.

[18] S. Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *Workload Characterization. International Symposium on*. IEEE. 2009, pp. 44–54.

[19] NVIDIA. *CUDA port of the Stream benchmark*. 2010. URL: `https://devtalk.nvidia.com/default/topic/381934/stream-benchmark/`.

[20] V. Volkov. "Better performance at lower occupancy". In: *Proceedings of the GPU Technology Conference, GTC*. Vol. 10. 2010.

[21] R. Saavedra-Barrera, D Culler, and T. Von Eicken. "Analysis of multithreaded architectures for parallel computing". In: *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM. 1990, pp. 169–178.

[22] A. Agarwal. "Performance tradeoffs in multithreaded processors". In: *Parallel and Distributed Systems, IEEE Transactions on* 3.5 (1992), pp. 525–539.

[23] X. E. Chen and T. Aamodt. "Modeling cache contention and throughput of multiprogrammed manycore processors". In: *Computers, IEEE Transactions on* 61.7 (2012), pp. 913–927.