X: A Comprehensive Analytic Model for Parallel Machines

Ang Li*, Shuaiwen Leon Song†, Eric Brugel‡, Daniel Chavarría-Miranda†, Akash Kumar§, and Henk Corporaal*

* Eindhoven University of Technology, The Netherlands
† Pacific Northwest National Laboratory, USA
‡ The State University of New Jersey, USA
§ Technische Universität Dresden, Germany
ang.li@tue.nl, shuaiwen.song@pnnl.gov, akash.kumar@tu-dresden.de,
brugel18@gmail.com,daniel.chavarria@pnnl.gov, h.corporaal@tue.nl

***Abstract*—To continuously comply with Moore's Law, modern parallel machines become increasingly complex. Effectively tuning application performance for these machines therefore becomes a daunting task. Moreover, identifying performance bottlenecks at application and architecture level, as well as evaluating various optimization strategies, are becoming extremely difficult when the entanglement of numerous correlated factors is being presented. To tackle these challenges, we present a visual analytical model named "X". It is intuitive and sufficiently flexible to track all the typical features of a parallel machine. Different from the conventional analytic models that focus on the temporal state of a representative core or thread, our proposed X-model concentrates on the spatial state of parallel machines – the distribution of concurrent threads among different subsystems of these machines, while predicting the overall throughput based on such state. One major highlight of our model is its tractability as it only requires a small number of essential parameters from the application and architecture. Meanwhile, it is able to effectively help users investigate the combined-effects of different types of parallelism: the instruction-level-parallelism (ILP), the thread-level-parallelism (TLP), the memory-level-parallelism (MLP) and the data-level-parallelism (DLP). Through our X-model, developers and architects can quickly draw an intuitive figure called X-graph to identify performance bottlenecks, as well as play "what-if " scenarios to evaluate the effectiveness of the proposed optimization techniques by investigating their individual and combined effects.**

I. Introduction

Despite the fact that Moore's Law has continued to show promising, the mainstream computing has been leveraging multiprocessors and parallel applications extensively for superior performance, due to the end of frequency scaling for uniprocessors. However, decades of practical experience demonstrated that analyzing and optimizing performance for the complex modern parallel architectures still remains a challenging task, especially concerning the huge design space with divergent types of parallelism to exploit. Therefore, developers often found themselves lost when exploring a large number of design options and their combined effects. For instance, as one of the most popular throughput-oriented many-core architectures, GPU is well-known for its ability to initiate thousands or even millions of concurrent threads. A performance metric called "occupancy" is then proposed to measure the ability of a workload to utilize the available thread slots on a GPU for peak performance. However, programmers who attempt to pursue high occupancy for better performance then become confused, as some research literatures later indicate that maximizing occupancy may lead to register spilling and inferior cache performance [1]. They become even more hesitated when other research demonstrate that if there are plenty of instruction-level-parallelism, better performance can be achieved with lower occupancy [2].

These challenges emerge because developers often constrain themselves to address a very specific performance issue for a machine component (e.g. registers, caches, main memory, etc) without much indication for better understanding of the global systematic effects. In other words, as modern parallel architectures become increasingly complicated, most performance factors are not independent with each other but are often inter-correlated or even inter-conflicted. Therefore, a high-level and easy-to-use performance analysis tool, that can provide comprehensive information for identifying performance bottlenecks and demonstrate the performance variation characteristics when a particular factor is altered, is highly desired.

In this paper, we present such a performance analysis tool called "X-model", which is a high-level and visualized analytic model for general parallel machines. It can help developers understand the observed phenomena and derive new optimization strategies. Based on the spatial state of the parallel machine, the model is able to comprehensively investigate the combined effects of various types of parallelism: the instruction-level-parallelism (ILP), the thread-level-parallelism (TLP), the memory-level-parallelism (MLP) and the data-level-parallelism (DLP); and it only requires very few essential parameters from application and architecture for the model construction. With our X-model, developers and architects can easily draw an intuitive figure called "X-graph" to identify performance bottlenecks and discern potential optimizations. More significantly, by drawing an X-graph, designers and researchers can easily find out, in a visualized and conceptual way, whether a proposed technique by a manuscript is effective for resolving the problem it targets and why, as well as what else can be done subsequently. This paper thus makes the following contributions:

- We propose a high-level visualizable analytic model

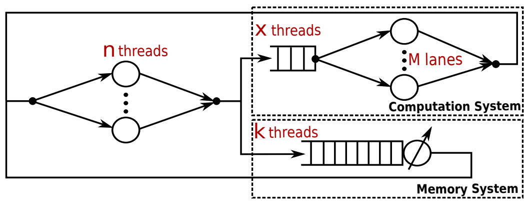

Figure 1. Baseline Multithreaded Machine Model.

for parallel machines that can comprehensively analyze the joint-effects of numerous factors such as MLP, ILP, TLP and DLP (Section III-(A)).

- We propose an approach to integrate shared cache into the X-model (Section III-(B)) to form X-graphs that can reflect complex cache effects (Section III-(C)). Based on these X-graphs, interesting performance insights are derived (Section III-(D)).
- We provide a thorough case study on how to leverage the X-model for evaluating different performance optimization options for real applications. We demonstrate that our model can identify the limiting factors, suggesting potential optimization techniques, reasoning and bounding the effectiveness of a technique, and explore new opportunities for further optimizations.

II. Background: The Transit Model

Before describing the *X-model* in detail, we first introduce the *Transit Model* [3] that we proposed previously for visualizing simple performance analysis for a multithreaded machine. Although X-model is built upon the Transit model, we further extend it to include important features such as analyzing various types of parallelisms and expressing sophisticated cache effects on modern architectures. These features are essential, and can significantly affect the overall performance of modern parallel machines.

In the Transit model, a multithreaded machine is partitioned into a computation system (CS) and a memory system (MS). Their boundary is flexible depending on the requirements. The CS throughput is viewed as the primary performance metric while MS throughput is also of interest. As shown in Fig.1, the multithreaded machine is modeled as an interactive queuing network. There are totally n threads in the machine, in which x of them are in CS and $n-x=k$ threads in MS. The CS is a single-queue-multiple-server system. Each server denotes an in-order computation lane that can perform one computation operation in a cycle. The MS is an aggregated queuing system. During execution, a typical thread executes in one of the M lanes of CS for Z cycles on average, and proposes a memory request. It then enters MS for L cycles to do the data fetching. After retrieved, the thread enters CS again, starting a new turnaround. The major parameters used in this paper are listed in Table I.

CS: As shown in Fig.1, with x threads occupying x lanes in CS, the utilization of CS would be x/M. As one computation lane generates one operation per cycle, the CS throughput function $g(x)$ can be expressed as $g(x) = min(x, M)$, which is a roofline-like figure shown in Fig.2-(B). Since there is one memory request per Z cycles on average, in total there are $g(x)/Z$ memory requests per cycle. This is the demand throughput from CS to MS. Note that we reverse X-axis' direction for further integration and utilization. π in Fig.2-(B) represents the CS transition point, at which CS begins to get saturated.

Table I
Major Parameters Used In This Paper

n	Total threads in the parallel machine
k	Threads in the memory system (MS)
x	Threads in the computation system (CS)
$f(k)$	MS supply throughput to CS
$g(x)$	MS demand throughput from CS
Z	Compute intensity (ops/bytes ratio)
E	Instruction-level-parallelism degree
R	Maximum sustainable MS throughput
M	Computation lanes
π	CS transition point (when CS is saturated)
δ	MS transition point (when MS is saturated)
L	Average MS access latency
h	Shared cache hit rate
ψ	Position of cache peak

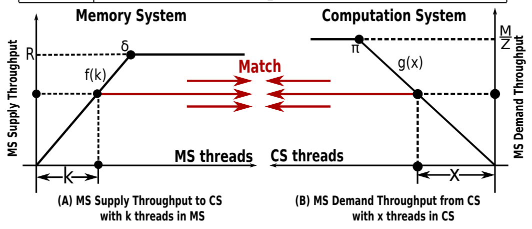

Figure 2. (A): MS supply throughput function $f(k)$ and (B): CS throughput demand function $g(x)/Z$ to MS.

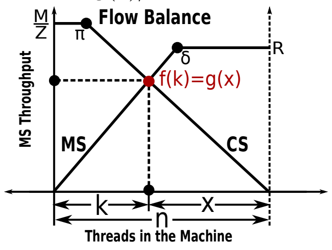

Figure 3. **Transit Figure**: the intersection of $f(k)$ and $g(x)$ represents the equilibrium between service demand and supply of MS. It indicates the spatial machine state: within the total n threads, k of them are in MS and x in CS.

MS: With k threads filling up k pipeline-slots in MS shown in Fig.1, if the MS pipeline delay is L, the utilization of MS can be described as k/L. Consequently, the MS throughput function $f(k)$ would be kR/L. This is also a roofline-like figure shown in Fig.2-(A). It illustrates the supply throughput from MS to CS. δ represents the MS transition point, at which MS starts to get saturated.

Based on the *flow balance property* [4], for a steady state of the system, $f(k) = g(x)$. Therefore, if we combine Fig.2-(A) and Fig.2-(B), a cross-roofline figure can be obtained, shown in Fig.3. This is called a *transit figure*. The intersection point of $f(k)$ and $g(x)$ is the equilibrium between the demand throughput and supply throughput of MS, which is exactly the current MS throughput, or $f(k_0)$ if k_0 is used to describe the k value at the intersection. Consequently, the CS throughput is $Z * f(k_0)$.

The **inputs** of the transit model are three architecture-related parameters R, L, M and two application-related parameters Z and n (described in Table I). In the transit model, since the raw memory latency L is very difficult to change in practice, it is postulated to be constant; the other four are changeable. The **output** of the model is the machine performance, or the delivered throughput of CS and MS. Three principles are proposed to evaluate the CS and MS throughput in the transit figure:

- ***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 compute intensity Z is increasing and the intersection is on the right side of CS transition point π, then CS throughput increases.*

The other focus of the transit model is on illustrating various state transitions of the multithreaded machine based on different types of performance bounds, including thread-bound, computation-bound, memory-bound and capacity-bound. Please refer to [3] for more detailed description.

III. The X-Model

In this section, we present the X-model. We use the letter "X" to label the model because it illustrates the general shape of the model — a cross-roofline. Unlike the original roofline model which is built generally for sequential machines, the X-model is a dynamic, high-level and visualized analytic model for parallel machines. Moreover, with only six parameters from application and architecture, and based on the present spatial state of a parallel machine, X-model can help users comprehensively explore the combined effects of various types of parallelism, including TLP, ILP, MLP, and DLP. This is very different than the transit model, in which only simple performance analysis (e.g. computation/memory/thread/capacity bound analysis) can be conducted. Furthermore, the X-model integrates the shared cache effects into the parallel machine shown in Fig.1 to form a more complete model for matching the complex modern multi- and many-core architectures, in which cache effects directly impact the overall performance. Next, we demonstrate how to operate our X-model for performance analysis and evaluation. Then, we discuss how to model and integrate the cache effects in the X-model. All the parameters discussed in the following subsections are shown in Table I.

A. Operating X-Model For Analysis and Evaluation

1) **Memory-Level-Parallelism (MLP):** As shown in Fig.4-(B), $\boldsymbol{L}$ is the average memory access latency. In the transit model, L is viewed as a constant parameter. In fact, the reciprocal of L is just the average per-thread memory throughput. Before MS throughput function $f(k)$ hits its upper bound R (or reaches the MS transition point δ), $1/L$ is the slope of $f(k)$. Since L is a constant, the sloping part of the curve is a straight line. Beyond the MS transition point δ ($k >= \delta$), $f(k)$ becomes flat as MS is already overloaded with the increasing number of k threads.

In the X-model, as $1/L$ is the average per-thread throughput and R is the overall throughput, then $\frac{R}{1/L} = RL$ essentially indicates the number of threads needed to saturate the MS, or the **MLP of the machine**. Usually, with R being fixed, the larger latency L, the more threads (a larger k) are required to fill the pipeline slots and hide the latency (Fig.4-(B)). Alternatively, with L being fixed, the larger throughput

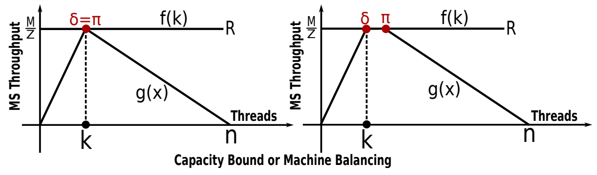

Figure 5. Capacity Bound or **Machine Balance**: both CS and MS attain its best performance. Meanwhile, due to the shortage of the machine capacity, some threads may be idle. π is the CS transition point and δ is the MS transition point.

R implies that more threads are necessary to approach R (Fig.4-(A)), which is just the MLP. On the other hand, the utilized MLP, or the **MLP of the workload**, is proportional to k, which is the number of threads in MS.

2) **Instruction-Level-Parallelism (ILP):** The effect of **ILP of the machine**, which is also the ILP of CS since MS does not have the ILP concept, is difficult to be illustrated in the X-graph because of its entangled relationship with the TLP in CS. Their combined effect is the number of computation lanes (M) in CS. Since most of the modern parallel machines adopt dynamic scheduling, both ILP and TLP of the workload can be exploited via these lanes simultaneously. Note that for a real machine, the ability to exploit ILP and TLP heavily relies on the underlying hardware design (see Section IV).

ILP of the workload is more important. It indicates the parallelism inside the scope of a single thread, or how many computation lanes a thread can leverage at the same time. In the transit model, ILP of the machine is assumed as one, meaning that a thread only occupies a single lane. In the X-model, a variable E is employed to describe the ILP degree of the workload. As shown in Fig.4-(E), we modify the CS curve $g(x)$ to address ILP. With a larger E, relatively less threads are required in CS (a smaller x) to fill up the available lanes and saturate CS. Note that compared to Z (compute intensity in Fig.4-(D)), E defines the slope of $g(x)$ while Z acts as a scaling factor when integrating CS and MS curves (see Section II and Fig.2) for the X-graph.

3) **Thread-Level-Parallelism (TLP):** Regarding the **TLP of the workload**, the X-model is similar to the transit model. It is simply n (Fig.4-(F)). However, the **TLP of the machine** in the X-model is quite different. It is defined as the minimum number of threads to hit the *capacity bound* or *machine balance*. As shown in Fig.5, two different scenarios of the machine balance are illustrated, at which both CS and MS attain its best performance. The capacity bound or machine balance describes the optimal state for software-hardware co-design since both CS and MS bandwidth are fully leveraged ($f(k) = R$, $g(x) = M/Z$) [3]. Unlike the right figure in Fig.5, the left one does not have any idle threads in neither CS nor MS. Therefore, its n is the TLP of the resident parallel machine.

4) **Data-Level-Parallelism (DLP):** For the **DLP of the workload**, it is defined as a metric that measures the number of computation operations performed per data element, which is the ratio between computation operations and memory operations of the workload, or Z (compute intensity or arithmetic intensity) shown in Fig.4-(D). Meanwhile, the **DLP of the machine** indicates the intrinsic characteristic of the machine, which can be represented as M/R. Essentially,

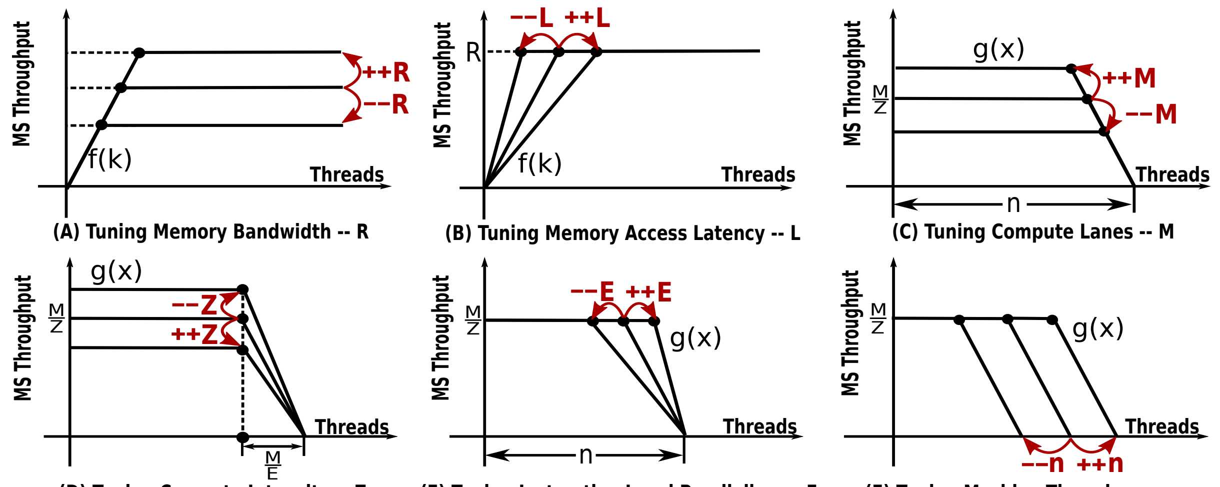

Figure 4. Operating X-Model.

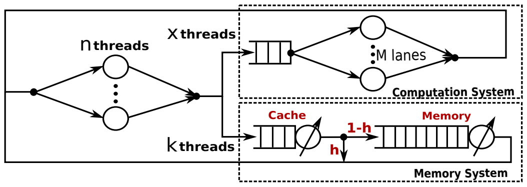

Figure 6. The new parallel machine model equipped with Shared Cache.

the relative relationship between DLP of the workload and DLP of the machine can be summarized as: *if DLP of the workload is less than DLP of the machine ($Z < M/R$), the system is memory bound; otherwise ($Z \geq M/R$), it is computation bound.* Note, DLP of the machine (M/R) is just the ridge point of the roofline figure [5]. To some extent, it indicates the level of difficulty for programmers to achieve the peak computation performance for the underlying architecture.

B. The X-Model with Cache Effects

In this subsection, we present the X-model integrated with shared cache effects. We first model a MS with shared cache integrated. Based on the obtained new MS throughput curve, we then show the complete X-model. Finally, we describe two novel observations revealed by the X-model.

1) **Modeling Cache Effects:** In the transit model, a basic assumption is that threads in a multithreaded machine are independent of each other and there is no cache interference among the threads. In addition, the average memory access latency L is fixed. Based on these assumptions, a roofline-like figure for the MS throughput function $f(k)$ is generated (Fig.2-(A) in Section II). In the X-model, we relax these restrictions and replace the roofline-like $f(k)$ with a more practical throughput curve that can better address the cache effects.

As shown in Fig.6, on top of the transit model, an intermediate cache system is placed ahead of the main memory in MS. If the hit rate of the shared cache is h, a memory request would have a probability of h to be quickly returned from the cache while a probability of $(1-h)$ to be slowly returned from the main memory. Therefore, if we use $L_\$$ to denote cache latency and L_m to denote off-chip memory latency, the average MS latency L with k threads in MS (i.e. L_k) would be:

$$L_k = h * L_\$ + (1-h) * L_m \tag{1}$$

and the new MS throughput function $f(k)$ with k threads would be

$$f(k) = k/L_k \tag{2}$$

The remaining question is to find a proper cache model that supports multithreading. We adopt the one proposed by Jacob et. al. [6] to accomplish this. If there are k threads accessing the cache, each thread shares on average $S_\$/k$ of the cache storage. The hit rate seen by a thread hence can be represented as:

$$h(\frac{S_\$}{k}) = 1 - (\frac{S_\$}{\beta k} + 1)^{-(\alpha-1)} \tag{3}$$

where α and β are the constant parameters describing the locality of the workload. Meanwhile, the main-memory throughput is still bounded by R. Therefore,

$$L_m = max\{L, \frac{k}{R}\} \tag{4}$$

where L is the constant memory latency before MS is saturated, as discussed before. Combining Eq.(1), (2), (3) and (4), we remodel the MS throughput function $f(k)$ as

$$f(k) = k/[L_\$ + (max\{L, \frac{k}{R}\} - L_\$)(\frac{S_\$}{\beta k} + 1)^{1-\alpha}] \tag{5}$$

A sample figure for the new $f(k)$ is shown in Fig.7. At the beginning, with the efficient utilization of the cache, the MS throughput increases almost linearly with the expanded MS threads, and eventually reaches a peak. We label this peak as **cache peak** where $k = \psi$. However, once the aggregated working set for the increased k threads exceeds the cache capacity, thrashing occurs and performance starts to degrade ($k > \psi$). Note that with a hit rate h, there are on average $h * k$ threads in the cache and $(1-h) * k$ threads in the main memory. At this time, the $(1-h) * k$ threads

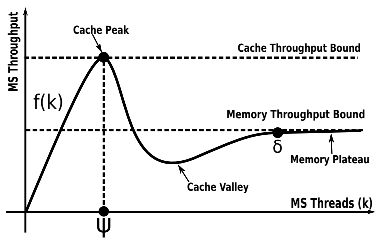

Figure 7. Throughput functions $f(k)$ for a MS with cache integrated.

are not sufficient to saturate the main-memory system. In other words, the MLP of MS cannot be fully exploited by $(1-h)*k$ threads (see Section III-A-(1)). This explains why there is a performance valley after the cache peak: *the cache throughput drops so quickly without the memory throughput increases fast enough*. We label this valley as **cache valley**. Beyond the ridge point of the cache valley, the main-memory starts to play the major role for performance with the cache impact diminishing. With the further expanded threads, $f(k)$ increases again with MS continuously exploited. Once the thread number reaches the MS transition point δ where MS is saturated, $f(k)$ remains stable afterwards. We label this stable throughput as the **memory plateau**.

In Fig.8, we summarize three operations to tune the cache-integrated MS throughput function $f(k)$ (i.e. Eq.(5)). The first operation is workload-locality related. As shown in Fig.8-(A), by tuning α and β, we can obtain three representative shapes of $f(k)$ corresponding to three different cache-sensitivity conditions: *cache insensitive (CI)*, *moderately cache sensitive (MCS)* and *highly cache sensitive (HCS)*. The CI applications present the same curve (Curve 1 in Fig.8-(A)) as the $f(k)$ function of MS without cache. For both MCS and HCS applications, there is a cache peak. However, the cache peak of MCS applications (Curve 2) is lower and flatter than that of HCS applications (Curve 3). In addition, the MCS cache peak can be reached with less threads. Beyond the cache peak, there is a cache valley for MCS applications and possibly for HCS applications, depending on the hit rate and MLP of the MS. However, the valley of HCS, if exists, is not as deep as that in MCS due to the less significant cache effects towards performance in MCS.

The other two operations are architecture related. Fig.8-(B) shows the condition of tuning cache capacity ($S_{\$}$ in Eq. (5)). Three curves correspond to *no cache*, *a small cache* and *a big cache*. Although the variations of the shapes are very similar to Fig.8-(A), they are not exactly the same: the change of $S_{\$}$ is more like a scaling transform of the cache peak and valley, and the displacement of the curves is quite uniform. Finally, Fig.8-(C) illustrates the scenario of tuning raw cache access latency ($L_{\$}$ in Eq. (5)). Although this cannot be easily done theoretically, it can significantly affect the height of the cache peak and the depth of the cache valley. Comparing Curve 2 (*a slow cache*) with Curve 3 (*a fast cache*), it is clear that a fast cache is always beneficial, as it strengthens the positive cache effects by increasing the cache peak, while mitigates the negative effects through raising or smoothing the cache valley. Also note that the positions of the cache peak and valley do not change on the x-axis when tuning $L_{\$}$.

C. X-graphs Reflecting Cache Effects

With the new $f(k)$, we are able to draw a complete X-graph. As shown in Fig.9-(A), the X-graph is more comprehensive and accurate than the transit graph shown in Fig.3. It also highlights one of the major advantages of the X-model over the Roofline model [5]: it compartmentalizes the machine into MS and CS. Therefore, when the cache effects or other effects (e.g. scratchpad memory, MSHRs, etc.) are needed to be reflected in the model, a new $f(k)$ based on a specific condition can be supplied without the interference from CS.

Note that we use the MS throughput as the y-axis in our X-graph instead of the CS throughput, albeit CS throughput seems more convenient for performance lookup. This is because, unlike $f(k)$, $g(x)$ is generally a regular roofline. If converting a complex cache-effect integrated $f(k)$ (shown in Fig.8) into the CS space by multiplying Z, the process can be complicated. Therefore, doing so simplifies the model.

D. Interesting Insights Gained From the X-graph

In this subsection, we will demonstrate two interesting insights on performance observed from the X-graph:

- An unstable intersection point exists in the X-graph but cannot be actually observed in practice;
- If Z is small and E is large, the workload may suffer from sharp performance degradation at certain critical point.

1) **Unstable Intersection:** Slightly different from Fig.9-(A), $f(k)$ and the $g(x)$ intersect at three points in Fig. 9-(B): σ, σ' and σ''. The key observation gained from this X-graph is that the intersection $\boldsymbol{\sigma}$ is essentially unstable and cannot be observed on real parallel machines, because any perturbation will cause the equilibrium (Fig.3) to move away. The dashed part indicates the unstable region.

Consider the scenario that the current intersection is σ. At this time, k will be increased by one if a thread happens to leave the computation system and issues a memory request. Consequently, the MS throughput reduces as $f(k)$ decreases with a larger k (the descending dash-line part of $f(k)$). Meanwhile, since $x + k = n$ is fixed, x decreases by one. Although this decrease also causes $g(x)$ to reduce a bit (at the sloping part of $g(x)$), the reduced magnitude of $g(x)$ is smaller than that of $f(k)$ because the dropping slope of $f(k)$ is steeper than that of $g(x)$. Therefore, there is more throughput loss of MS than CS. Starting from the equilibrium σ, $f(k)$ becomes smaller than $g(x)$ after this process, causing more threads to leave CS than entering, since MS is the bottleneck currently (i.e. $f(k) < g(x)$). This leads to a further increase of k and triggers the same process again. Such process repeats until $f(k) = g(x)$, reaching a stable interaction σ''.

From the same initial state σ, the other possibility is that a thread happens to obtain the fetched data and aborts MS. This decreases k by one, which leads to the throughput increase for both MS and CS. However, as the slope of $g(x)$ is steeper than $f(k)$, after the process, $f(k) > g(x)$, making CS to be the performance bottleneck and more threads are likely to leave MS and being blocked in CS. Consequently, k decreases further, which will trigger the same process

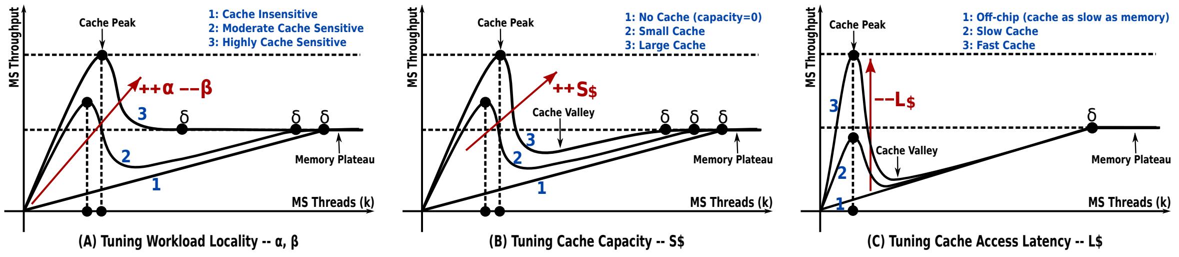
MS Throughput
Cache Peak
1: Cache Insensitive
2: Moderate Cache Sensitive
3: Highly Cache Sensitive
++α --β
δ
Memory Plateau
MS Threads (k)
(A) Tuning Workload Locality -- α, β
1: No Cache (capacity=0)
2: Small Cache
3: Large Cache
++S$
Cache Valley
(B) Tuning Cache Capacity -- S$
1: Off-chip (cache as slow as memory)
2: Slow Cache
3: Fast Cache
--L$
(C) Tuning Cache Access Latency -- L$

Figure 8. The three operations to tune the cache-integrated MS throughput function $f(k)$: (A) tuning working load locality; (B) tuning cache capacity; (C) tuning cache access latency.

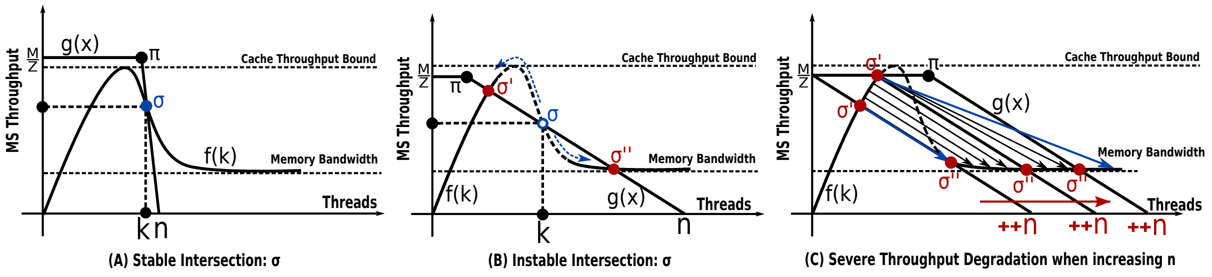
MS Throughput
g(x)
π
σ
Cache Throughput Bound
f(k)
Memory Bandwidth
Threads
k n
(A) Stable Intersection: σ
σ'
σ''
k
n
(B) Instable Intersection: σ
++n
(C) Severe Throughput Degradation when increasing n

Figure 9. A complete X-graph reflecting cache effects. It illustrates three scenarios: (A) stable intersection; (B) unstable intersection; and (C) severe performance degradation when increasing n.

again. Such a process repeats until $f(k) = g(x)$. Under this condition, however, the machine state shifts leftwards and eventually settles at σ'.

To summarize, *any perturbation to k will cause the machine state to diverge from σ*. However, the intersection in Fig.9-(A) can be converged as the slope of $g(x)$ is steeper than that of $f(k)$. A perturbation is then revised under this condition, making this intersection stable. To explain this using a mathematical form, the stable scenarios in Fig. 9-(B) need to meet the following **derivative** relationship:

$$|\frac{\partial g(x)}{\partial x}| > |\frac{\partial f(k)}{\partial k}| \tag{6}$$

The remaining question for Fig.9-(B) is: at which point (σ' or σ'') will the machine eventually converge to? It is hard to say from the model itself. Mostly it depends on the thread distribution: if there are more threads in CS (x is large), CS is likely to have a higher throughput, which matches the good performance of MS with comparatively less threads in MS (k is smaller with a larger x under $x+k = n$). Under this scenario, the machine stabilizes at σ'. However, if there are less threads in CS, the lower throughput of CS also matches the poor performance of MS since excessive threads congest the cache, causing severe thrashing and resource shortage (e.g. MSHRs). The machine then stabilizes at σ''. Clearly σ'' is undesirable as the performance is poorer.

2) **Severe Performance Degradation:** We further explore the two stable intersections in Fig.9-(B). As the machine state may be settled at either σ' or σ'', from σ' to σ'' the performance drops quite significantly. If we add more threads to the machine (increase n, see Fig.4-(F)), as shown in Fig.9-(C), the positions of σ' and σ'' also move accordingly. However, when σ' coincides with the CS transition point δ, σ' starts to remain constant. At this moment, the parallel machine is already computation bound although the cache can deliver higher throughput. The arrows in Fig.9-(C) indicate the magnitude of performance degradation that the machine might suffer from when increasing n: the *minimum* is from σ' to σ'', which occurs when $g(x)$ is tangent to $f(k)$. The *maximum* is $\frac{M}{Z} - R$, which attains when there are infinite threads in the machine.

In summary, there are two forms of the X-model: *the regular one with cache* and *the simpler one without*. Generally, if users do not need to consider the cache effects, the basic X-model is more straightforward and simple. However, for the majority of the complex modern architectures, dealing with cache-level effects and optimizations seems more common. In Section VI, we will show a case study using the regular X-model with cache effects.

IV. Guidelines For Plotting X-Graph

In this section, we provide guidelines on how to draw an X-graph that represents the integration of features from workload and architecture. Our X-model provides good abstraction for both the understudied architecture and the application. From the perspective of an architecture, it extracts three machine-related parameters M, R, L, based on which an *architectural X-graph* can be drawn first and it only requires to profile the hardware once. In this paper, to showcase the ability of our model to address complex architectures, we choose to use one of the most popular throughput-oriented many-core architecture—GPU, for the purpose of evaluation and illustration. However, the same methodology can be applied to other parallel machines. Fig.10 shows the samples of architectural X-graphs based on the three major GPU generations (i.e., Fermi, Kepler and Maxwell) under single (SP) and double precisions (DP). To profile $f(k)$ (i.e. L and R) for the architectural X-graph, we use a modified CUDA version of the Stream Benchmark [7]. To profile the $g(x)$ (i.e. M), we developed a microbenchmark based on the method proposed in [2].

From the perspective of an application, the X-model extracts three application-dependent parameters Z, E, n, based on which the architectural X-graph shown in Fig.10 can be

Table II
Experiment Platforms. "LDS" is the number of load/store units per SM. "Scher" indicates the number of warp-schedulers per SM. "Disp" is the number of warp-dispatch-units per SM. δ(SP) is the transition point for the MS throughput with single-precision floating-point like data size (4 bytes) and fully coalescing access. The unit is warps – GB/s, e.g. 48/147 means MS throughput function saturates at 147 GB/s with 48 warps. δ(DP) is for 8 bytes data size with coalescing. There are at most 32 warps per thread block, so the X-axis stops at 32.

GPU	Arch	SM $\times$ SP	LDS	Freq	Mem Band	Dri/Rtm	Max Warps	Schr	Disp	δ(SP)	δ(DP)
GTX570	Fermi-2.0	15x32	16	1464 MHz	152 GB/s	6.5/4.0	48	2	2	48/147	24/152
Tesla K40	Kepler-3.5	15x192	32	876 MHz	288 GB/s	6.0/6.0	64	4	8	64/180	48/200
GTX750Ti	Maxwell-5.0	5x128	32	1137 MHz	86.4 GB/s	6.5/6.5	64	2	4	56/82	28/83

tuned to specifically address an application. The X-model provides a convenient way to enable independent evaluation on architecture using a series of different application features. It also provides a way to predict application performance on an unreachable or nonexistent platform if those hardware features can be provided (R, M, L, and the ability to exploit ILP, TLP DLP and MLP). To draw an application X-graph, we first parse the application code/instructions via compiler/assembler. Once the ILP (i.e. E) is obtained, we then tune g(x) according to Fig.4-(E), which corresponds to choose a curve from the $g(x)$ series with different Es, shown in Fig.10. Depending on the value of n, we can change the relative distance between $f(k)$ and $g(x)$, refer to Fig.4-(F). Finally, when Z is available, we can divide CS throughput by Z to convert the $f(k)$ and $g(x)$ curves into the same MS throughput space (as can be seen, the left y-axis is MS throughput and the right y-axis is CS throughput, they are not in the same space). Thus, their intersection is just the current machine state, or present MS throughput. Following these three steps above, we can obtain the X-graph for a application running in a specific architecture. We will show some examples of applications' X-graphs in the next section.

V. Validation

In this section, we validate the X-model on the Kepler platform (listed Table II). We use 12 practical applications `bfs`, `backprop`, `stencil`, `gesummv`, `hpccg`, `heartwall`, `leukocyte`, `nw`, `nn`, `spmv`, `atax`, `lud` from common-used benchmarks including Rodinia [8], Parboil [9], Polybench [10] and [11]. Based on the guideline introduced in the previous section, we take the Kepler architectural X-graph (Fig.10-B) as the start-point and tune the $g(x)$ curve according to the application features, which are E, n and Z. To obtain these software-related parameters, we parse the SASS assembly code of the application. Regarding ILP or E, we use a new approach that is different from the existing one based on CFG analysis for a general machine [12]. Since Kepler, GPUs start to embed scheduling information in the SASS assembly code to simplify the hardware scheduler's task and reduce energy. We thus developed a tool to read this scheduling information from the cubin file and count the average number of instructions that are issued simultaneously, which is the ILP. Note the ILP obtained here is always less than or equal to two because the scheduling information is within the scope of a single warp and does not tell how many warp schedulers (4 for Kepler shown in Table II) will select instructions from the single warp at runtime. In order to be accurate, we weight the ILP values for each code-block by the number of iterations for that block. Similarly, we also count the ratio between the number of total instructions and off-chip memory instructions for all the basic code-blocks, and weight by the number of loop iterations to calculate the value of computation intensity (Z). Finally, we calculate how many warps can be allocated simultaneously on a SM (i.e. the occupancy), which is the just the value of n. We developed a script to draw the X-graph and compared the predicted computation and memory throughput (i.e. the MS and CS throughput at the intersection of $f(k)$ and $g(x)$) with the values measured by the CUDA profiler. The results are shown in Fig.11.

As can be seen, for most of the applications, the dark star (measured memory throughput) is quite near the intersection (predicted by the X-model). Note that for SP scenarios, MS saturates at 2048 threads (64 warps), which is also the the maximum allowable threads per SM. This explains the linear behavior of $f(k)$ in most applications. `hpccg` is a DP application. Overall, using the computation throughput (PCT and RCT in Fig.11) as the metric, our model achieves 84.1% prediction accuracy. Consider only three parameters are extracted from the application, this is already quite accurate. The major factor that may impact the accuracy, is believed to be the coalesced memory access, as we do not count the coalesced access effect of MS.

VI. Case Study

In this section, we show an example on how to leverage the X-model for evaluating different performance optimization options for real applications. We use a memory-intensive benchmark named `gesummv` from Polybench [10] as the target kernel. The platform we take for showcasing is Fermi GTX570, shown in Table II. Note that this case study is to show the usage of the X-Model in detail; the general guideline is the same for other applications and platforms. Initially, 16 warps or 512 threads are allocated per thread block, which means all the 48 warp-slots per SM are fulfilled (with three thread blocks). The occupancy is 1. Besides, 16KB L1 cache on each SM is allocated by default.

To accurately reflect the present machine state for `gesummv`, we draw its X-graph based on the method discussed in Section IV. As shown in Fig.12, the isolated yellow points are the trace-points of $f(k)$ profiling via the bypassing approach in [13]. The green curve is the plot of $f(k)$ generated by connecting and smoothing these trace-points. We can observe that $f(k)$ and $g(x)$ intersect at the dropping slope of $f(k)$, which indicates that the L1 cache is thrashing currently and the machine shows a suboptimal performance. Under this thrashing condition, an intuitive tuning approach is to increase the L1 cache size, as discussed in Fig.8-(B). Fig.13 shows the new X-graph in which the L1 is increased from 16KB to 48KB. However, very limited performance gain is observed after such tuning (only about 0.1GB/s MS throughput gain). The

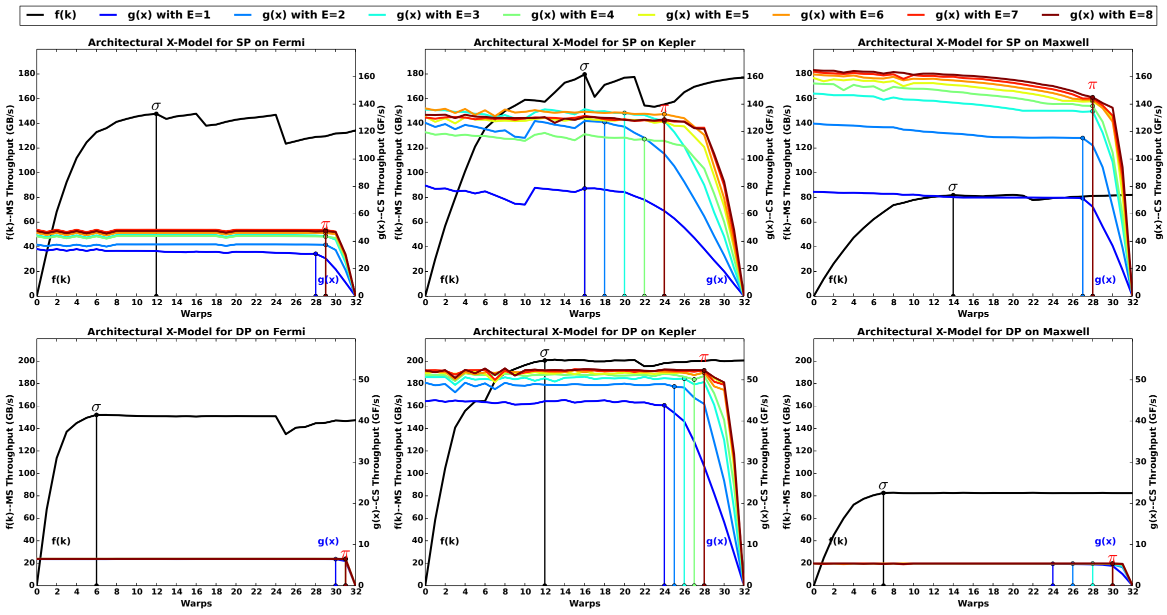

Figure 10. X-graphs for three different GPU architectures under single and double precisions.

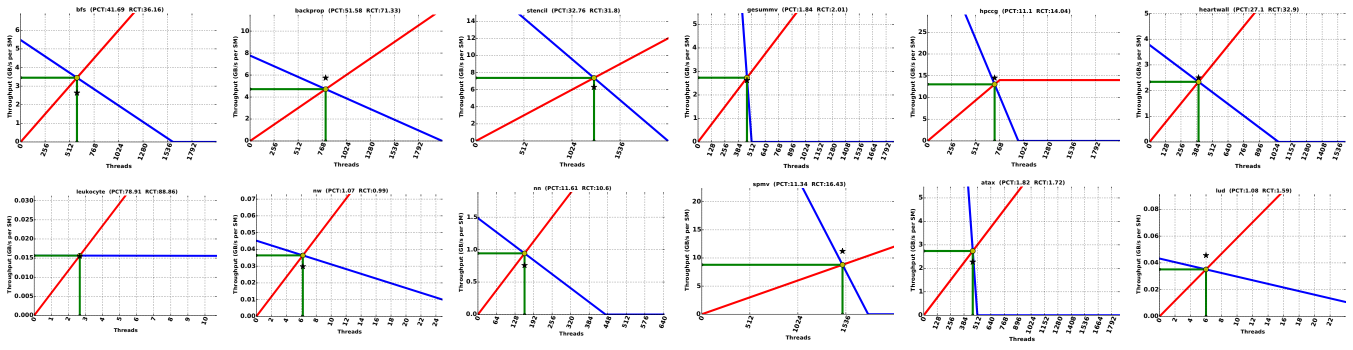

Figure 11. Validation Results on Kepler Platform.

intersection is still at the dropping slope of $f(k)$, which indicates that the reason behind such poor performance improvement is not that the application is cache insensitive, but because the cache thrashing condition is still severe due to resource contention (e.g., limited MSHRs and miss queue entries) or bad data locality. However, compared with the 16KB L1 scenario (Fig.12), the cache peak of 48KB L1 in Fig.13 is much higher, which also implies that: (1) If the cache thrashing can be effectively resolved (e.g. via cache bypassing), the achievable performance can be much higher. In other words, the potential performance can be increased by reducing capacity misses through larger cache. (2). Our cache enlarging operation in Fig.12 is correct. The X-model here highlights its first usage: **investigating machine states and identifying the limiting factors for performance**.

To further improve the performance of the scenario shown in Fig.13, we generate other tuning approaches by evaluating each model-tuning operations illustrated in Fig.4 and Fig.8, with the intention of increasing CS/MS throughput. After eliminating the ones that cannot improve CS/MS throughput under this thrashing condition (e.g., manipulating computation lanes M), we propose four optimization strategies for `gesummv`: *thread throttling* (Fig.14), *cache bypassing* (Fig.15), *increasing compute intensity* (Fig. 16) and *reducing ILP* (Fig.17). They correspond to the operations of decreasing n, increasing R, increasing Z and decreasing E, respectively. Here, we show the second usage of the X-model: **deriving and selecting the potential optimization approaches.**

Thread throttling [14], [15] is to restrict the number of concurrent threads on a SM to adapt the cache capacity or memory bandwidth [1]. Cache bypassing [16], [13] is to keep a limited number of threads accessing the cache while others bypass the cache to a lower memory hierarchy (in our case, bypass L1 to L2). Although both techniques are demonstrated to be effective for cache thrashing in various existing work, the explanation on when specific techniques would achieve the most performance gain as well as when they are going to fail, is unknown. The X-graphs in Fig.14 and Fig.15, however, can help us explain these directly. They show that the intersection goes up in both graphs under thread throttling and cache bypassing. The best performance is achieved when $g(x)$ coincides with the cache peak ψ in Fig.14 and when R rises to the same level as the cache peak in Fig.15. Eventually, further thread throttling or bypassing beyond the cache peak will start to degrade the performance again. Here, we show the third usage of the X-model: **reasoning and bounding the effectiveness of a technique.**

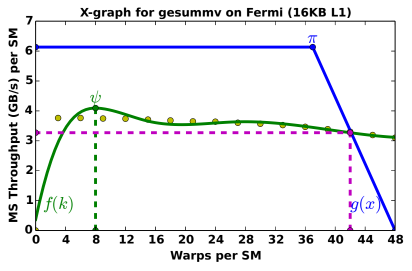

Figure 12. The X-graph for gesummv on GTX570 with default 16KB L1 and 48 warps.

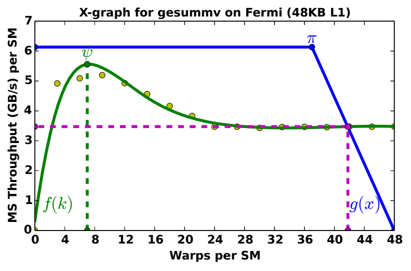

Figure 13. The X-graph for gesummv on GTX570 with 48KB L1 cache size and 48 warps.

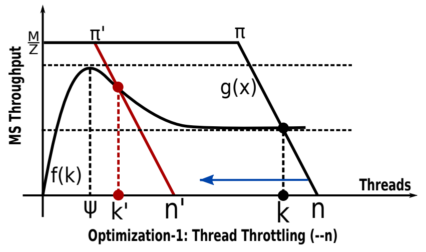

Figure 14. Thread Throttling. As the intersection goes up and Z is unchanged, based on Principle 2, both CS and MS performance increase.

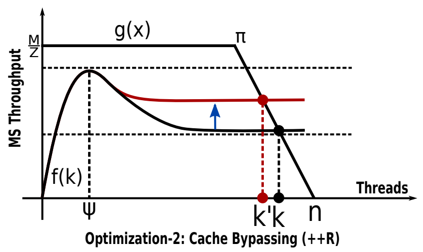

Figure 15. Cache Bypassing. As the intersection goes up and Z is unchanged, based on Principle 2, both CS and MS performance increase.

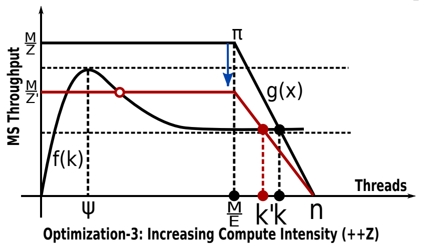

Figure 16. Increasing Z. As Z is increased and the intersection goes up only slightly, based on Principle 3, CS throughput is enhanced but MS throughput improvement is very limited.

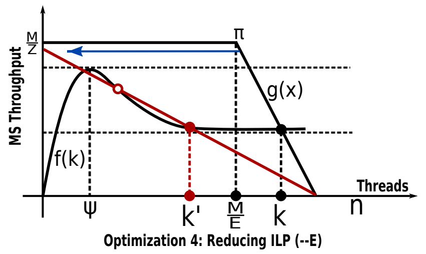

Figure 17. Reducing E. As the intersection goes up and Z is unchanged, based on Principle 2, both CS and MS performance increase.

Furthermore, compared to thread throttling and cache bypassing, the two much less obvious tuning options are illustrated in Fig.16 and Fig.17. Fig.16 shows that although increasing Z can enhance the CS throughput for gesummv (as Z is increased, based on Principle 3, CS throughput is increased), the improvement for MS throughput is very limited (i.e. the height difference between the two intersections is tiny). Note that Z value of an application is mostly decided by its algorithm. Therefore, to increase Z, algorithm modification is often required. Fig.17 shows something very interesting that has not been explored by any existing literature as a performance tuning method: *reducing ILP level (E) of an application can potentially increase the MS and CS throughput under cache trashing effect*. We leave the exploration on this new observation from our X-model as the future work. Nonetheless, we show the last usage of the X-model here: **exploring new opportunities for performance improvement**.

Finally, shown in Fig. 18, we validate the tuning approaches suggested by the X-model above, including larger cache size, thread throttling and cache bypassing on a real GPU platform. We also show the performance of disabling L1 as a reference. The performance results are normalized to the default condition with 16KB L1 cache. Overall, using 48KB L1 cache achieves 7% speedup; thread throttling achieves 8% and 26% speedup for 16KB and 48KB L1 scenarios respectively; and cache bypassing achieves 22% and 36% speedup under two cache sizes respectively. These figures demonstrate that the tuning approaches offered through the X-model are effective with regard to performance optimization for a real parallel machine.

VII. Related Work

In this section, we discuss two existing analytic models that are widely-known and related to our X-model: the roofline model [5], [17], the valley model [18], [19], and the MWP-CWP model for GPUs [20], [12].

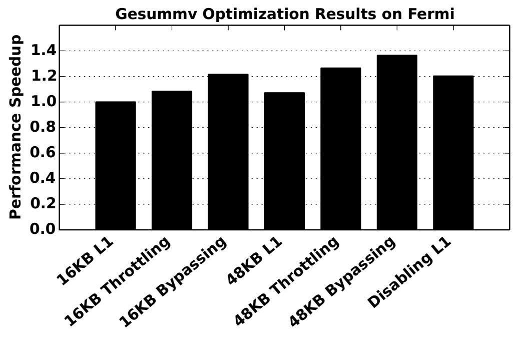

Figure 18. Validation of the tuning insights provided by the X-model.

Roofline Model: The roofline model [5], [17] draws a roofline-like figure to show the variation of machine throughput with respect to the arithmetic intensity of the workload, which is essentially the relative relationship between DLP of the workload and DLP of the machine (i.e. Z and M/R). Both models aim at providing a visualizable and intuitive throughput model. However, the X-model is significantly different in three aspects. First and most important, the roofline model is generally for **sequential machine** and only addresses the influence of Z. The X-model, however, is for **parallel machines**. We address the impacts from various types of parallelism including ILP, TLP, MLP and DLP. Second, the roofline model is constructed based on **bottleneck analysis** whereas the X-model is built upon **flow balancing**. The roofline model is basically **static** for a certain machine, and by profiling Z of a workload, users can decide if the workload is memory-bound or computation-bound. The X-model, however, tracks the spatial state of the machine with a specific workload, which is the equilibrium between CS and MS. Any change of the parameters leads to the variation of the X-graph. Therefore, the X-model is **dynamic**. Finally, the X-model is much more **flexible** than the roofline model. In the roofline model, there is only one curve representing both MS and CS. In our X-model, we separate the MS curve from the CS curve so that each of them can be profiled, varied and analyzed independently. Therefore, X-model makes it possible to investigate complex architectures (e.g., the complicated cache effects) by replacing $f(k)$ and $g(x)$ with more sophisticated and accurate shapes.

Valley Model: In [18], [19], Guz et. al. proposed an analytic model to describe the interaction between thread volume and shared cache for a multithreaded-manycore machine. Specially, they identified a *performance valley* between the cache efficiency zone and multithreaded efficiency zone for applications showing super-linear degradation of the hit rate with increased threads.

Although our modeling process for the cache effects in Section III-B is analogous, the X-model itself is dramatically different. First, the valley model assumes that MS always remains the major bottleneck of the machine. We do not have this assumption so that factors such as ILP degree (E) can affect the cache performance, as discussed in Section VI. Second, the valley model assumes that allocated threads in the machine (i.e., n) share the cache storage. However, we argue that in the steady state of a parallel machine, within a certain time interval, only a fraction of the threads (MS threads) are essentially accessing MS. Therefore, the cache sharing should be only among these MS threads ($\boldsymbol{k}$) instead of all threads of the machine ($\boldsymbol{n}$), as reflected in Equation 3. Third, the memory latency in valley model is fixed. That is why they introduced a bound from the CS part. In our X-model, the memory latency is changeable as the overall throughput is less than R. Finally, the CS and MS threads in the valley model are combined. The model focuses on their **joint effect** based on the MS bound assumption. As a comparison, the X-model separate the parallel machine into two curves and concentrates on their **relative effect**. Therefore, the X-model can offer more insights like the instable equilibrium and the sharp performance degradation discussed in Section III-D.

MWP-CWP Model: MWP-CWP model [20], [12] is proposed to model execution time for GPUs specifically. It involves complex architectural level parameters and requires the support of simulation tools and PTX code, and it lacks the flexibility to play "what-if" scenarios for evaluating the effectiveness of different optimization techniques. Our X-model eliminates the "only GPU" part, so that it can be applied for general parallel machines. Although the intention of our model is to provide high-level evaluation for the present state of a parallel machine and propose useful intuition for optimizations, it can also be extended for execution time prediction if needed.

VIII. Conclusion

In this paper, we propose a performance model named "X", which is a high-level and visualized analytic model for general parallel machines. Based on the spatial state of the machine, the X-model is able to comprehensively investigate the combined effects of various types of parallelism and the complex cache effects. With the model, developers and architects can easily draw an X-graph to identify performance bottlenecks, discern potential optimizations and derive novel intuitions.

References

[1] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: optimizing thread-level parallelism for GPGPUs," in *Proc. PACT*. IEEE, 2013.

[2] V. Volkov, "Better performance at lower occupancy," in *Proc. GTC*, vol. 10. San Jose, CA, 2010.

[3] A. Li, Y. C. Tay, A. Kumar, and H. Corporaal, "Transit: A visual analytical model for multithreaded machines," in *Proc. HPDC*. ACM, 2015.

[4] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[5] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, 2009.

[6] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge, "An analytical model for designing memory hierarchies," *TC*, vol. 45, no. 10, 1996.

[7] NVIDIA. CUDA port of the stream benchmark. [Online]. Available: https://devtalk.nvidia.com/default/topic/381934/stream-benchmark/

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IISWC*. IEEE, 2009.

[9] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[10] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *InPar*. IEEE, 2012.

[11] J. Dongarra, "Toward a new metric for ranking high performance computing systems," *Sandia Report*, no. SAND2013-4744 312, 2013.

[12] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *Proc. PPoPP*. ACM, 2012.

[13] A. Li, G.-J. v. d. Braak, A. Kumar, and H. Corporaal, "Adaptive and Transparent Cache Bypassing for GPUs," in *Proc. SC*. ACM, 2015.

[14] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient gpu computing," in *Proc. MICRO*. IEEE, 2014.

[15] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proc. MICRO*. IEEE Computer Society, 2012.

[16] W. Jia, K. Shaw, M. Martonosi *et al.*, "MRPB: Memory request prioritization for massively parallel processors," in *Proc. HPCA*. IEEE, 2014.

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

[18] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-core vs. many-thread machines: stay away from the valley," *CAL*, vol. 8, no. 1, 2009.

[19] Z. Guz, O. Itzhak, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Threads vs. caches: modeling the behavior of parallel workloads," in *Proc. ICCD*. IEEE, 2010.

[20] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proc. ISCA*. ACM, 2009.