Themis: Predicting and Reining in Application-level Slowdown on Spatial Multitasking GPUs

*Wenyi Zhao, *†Quan Chen, ‡Hao Lin, ‡Jianfeng Zhang, *Jingwen Leng,
*Chao Li, *Wenli Zheng, *Li Li, *†Minyi Guo
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
†Shanghai Institute for Advanced Communication and Data Science, Shanghai Jiao Tong University, China
‡Alibaba Group, China
wenyizhao@sjtu.edu.cn, {chen-quan, leng-jw, lichao, zheng-wl, lilijp, guo-my}@cs.sjtu.edu.cn,
bixuan@taobao.com, xingdian@alibaba-inc.com

***Abstract*—Predicting performance degradation of a GPU application when it is co-located with other applications on a spatial multitasking GPU without prior application knowledge is essential in public Clouds. Prior work mainly targets CPU co-location, and is inaccurate and/or inefficient for predicting performance of applications at co-location on spatial multitasking GPUs. Our investigation shows that hardware event statistics caused by co-located applications, which can be collected with negligible overhead, strongly correlate with their slowdowns. Based on this observation, we present Themis, an online slowdown predictor that can precisely and efficiently predict application slowdown without prior application knowledge. We first train a precise slowdown model offline using hardware event statistics collected from representative co-locations. When new applications co-run, Themis collects event statistics and predicts their slowdowns simultaneously. Our evaluation shows that Themis has negligible runtime overhead and can precisely predict application-level slowdown with prediction error smaller than 9.5%. Based on Themis, we also implement an SM allocation engine to rein in application slowdown at co-location. Case studies show that the engine successfully enforces fair sharing and QoS.**

***Index Terms*—Spatial Multitasking GPU, Co-location, Slowdown prediction**

I. Introduction

Large-scale applications, such as graph processing [37] and deep neural networks [22], are now more and more computationally demanding. GPUs are particularly suitable for these applications from both the performance and total cost of ownership (TCO) perspectives. As such, private data centers (e.g. Google's [11]) and public multi-tenant Clouds (e.g. Amazon EC2 [3]) have adopted GPU-outfitted servers.

While more and more streaming multiprocessors (SMs) are integrated into a GPU (from 16 SMs in a a Nvidia K40 GPU, to 56 in P100, to 80 in V100), a single application cannot always fully utilize all the resources [28]. For instance, a compute-intensive GPU kernel would consume most of the SMs but waste global memory bandwidth, while a data-intensive kernel wastes computational power of SMs. To improve resource utilization, prior work [1], [28], [32] proposed spatial multitasking GPU, where the SMs can be explicitly allocated to different GPU applications using preemption technique [25]. As a result, applications with complementary resource requirement can be co-located on the same GPU, which leads to better aggregated throughput [15].

On the other side, co-located applications can experience performance slowdown compared to their solo-run owing to the contention in shared resources such as shared cache capacity and global memory bandwidth. As such, the knowledge on the slowdown of each application is crucial for co-locating them. For instance, cloud providers may bill the customers based on the slowdowns of their applications due to the co-location (aka. fair pricing). Furthermore, accurate slowdown prediction may enable better SM allocation that avoids over-provision while still guarantees QoS or fair sharing. Failing to guarantee QoS or fair pricing/sharing, cloud tenants may resist the co-location according to Service Level Agreement (*SLA*), wasting the opportunity to improve the aggregated throughput.

We observe that the application's slowdown at co-location on spatial multitasking GPU can be broken down into two parts: scalability slowdown, resulting from the reduction of SMs assigned to it, and interference slowdown, resulting from the contention on L2 cache and global memory bandwidth. It is nontrivial to predict the slowdown since GPU sharing introduces varying amount of contention among the co-located applications and the applications often have different scalabilities as well as different sensitivities to the interference on the shared L2 cache and global memory bandwidth.

Prior work on slowdown prediction mainly targets CPU co-location [18], [24], [33], [35] and can be divided into two categories: profiling-based and model-based. Profiling-based techniques (e.g., Bubble-Flux [33]) periodically phase-in and phase-out each application to collect its characteristics. While the phase-in/-out operations on CPU are lightweight, they require heavy SM preemption support on GPU, which we show significantly degrades aggregated performance (inefficient). On the other hand, prior model-based techniques (e.g., ASM [24]) ignore the impact of the number of SMs allocated to an application on its performance, which makes the prediction inaccurate.

Our investigation shows that a kernel's **scalability**, **sensitivity to resource contention**, and **"pressure" on shared resources** affect its slowdown at co-location. The pressure varies when the kernel is allocated different numbers of

Quan Chen and Minyi Guo are co-corresponding authors of this paper.

SMs (Section III). We further find that some hardware event statistics that can be collected online with negligible overhead strongly correlate with these features. Based on the above observations, we propose **Themis** to precisely and efficiently predict application-level slowdown at co-location on spatial multitasking GPUs, without degrading aggregated performance. We train an unified kernel slowdown model in Themis using correlated hardware event statistics as inputs. The training data is collected by co-locating representative GPU applications under multiple SM allocations (Section V). When new GPU applications co-run, Themis periodically collects event statistics, and predicts their slowdowns simultaneously using the kernel slowdown model.

Moreover, we propose a novel proactive slowdown prediction method to predict the slowdown of an application with an un-tried SM allocation (Section VII). Based on this method, an SM allocation engine reallocates SMs periodically to rein in application slowdown to satisfy user requirements.

The main contributions of this paper are as follows.

- **Comprehensive slowdown analysis of co-located applications on a spatial multitasking GPU**. We breakdown application slowdown into scalability slowdown and interference slowdown. The analysis lays the foundation for our accurate slowdown prediction methodology.
- **A precise online slowdown prediction with negligible overhead**. We show that prior profiling-based slowdown prediction techniques, originally designed for CPU with fast context switch support, are not applicable on GPUs due to the large SM preemption overhead.
- **A novel method to handle prediction of un-tried SM allocations**. We show previous posterior prediction methods do not work owing to the unavailability of event statistics under un-tried SM allocations, for which we use constrained piecewise regression.

Our evaluation shows that Themis has negligible runtime overhead and precisely predicts application slowdown with error smaller than 9.5%. In addition, our case studies show that the SM allocation engine based on Themis can precisely rein in application slowdown to enforce fair sharing and QoS.

II. Related Work

Co-locating applications in datacenters has been an active research area because it can improve the server utilization.

On the one hand, researchers attempted to mitigate resource contention to improve performance and fairness [8], [12], [38]. FST [12] estimated performance unfairness in a shared memory multi-core system and eliminated unfairness due to bandwidth contention. Zhuravlev et. al [38] proposed a scheme that classified threads into multiple classes, and scheduled threads accordingly to eliminate the interference on CPU. Similarly, there were also efforts on contention mitigation on GPUs [27]. Park et al. proposed to dynamically change the multitasking mode for better performance on GPUs [21]. Wang et. al [30] proposed a hybrid method to mitigate the interference between CPU and GPU.

These prior work mostly rely on heuristics or contention related metrics to perform mitigation. On the other hand, qualitatively predicting performance interference between applications at co-location has been identified as a key challenge in datacenter and Cloud.

For CPU co-location, Bubble-Up [18] created bubbles that have different pressures on the memory subsystem and profiles each application individually to measure its sensitivity to resource contention. Bubble-Flux [33] and ASM [24] employed online profiling techniques. However, these techniques are either inadequate for GPUs owing to different architectural features or invasive that requires modification of applications' normal execution. Such invasiveness can also translate to unaffordable profiling overhead on spatial multitasking GPUs (in Section III). GDP [16] is a transparent interference prediction technique, but is built on latency-optimized CPUs and challenging to be extended for throughput-optimized GPUs.

For GPU co-location, Baymax [7] and Prophet [6] predicted performance interference among co-located GPU applications for a temporally shared GPU while we consider spatial multitasking GPUs. Zhao et al [36] studied SM optimization on spatial multitasking GPUs. They found memory-bound applications exhibit a complex and program-specific slowdown behavior when SM allocation changes, for which they used a trial-and-failure approach without relying on an actual model. Compared with them, Themis is accurate, spatial multitasking GPU-specific, and transparent with negligible performance overhead.

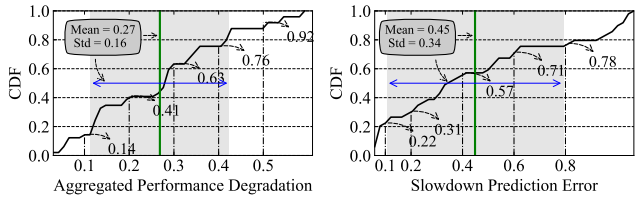
There is also prior work that builds performance models with analysis based approaches or machine learning based approaches for GPU architectures. Themis borrow their insights but extend them in the context of co-location performance prediction. Zhang et. al [34] developed a microbenchmark-based performance model for Nvidia GPUs. The model identifies GPU program bottlenecks and allows programmers to predict the benefits of potential program optimizations and architectural improvements. Wu et. al [31] trained a neural network model that predicts application performance on various GPUs.

III. Background and Motivation

Our investigation seeks to answer three research questions in this section. First, without application knowledge beforehand, can prior work precisely and efficiently predict application slowdown when multiple applications are co-located on a spatial multitasking GPU? Second, if cannot, what are the reasons of the poor accuracy and/or efficiency? Third, what are the challenges to enable precise and efficient application-level slowdown prediction without prior knowledge?

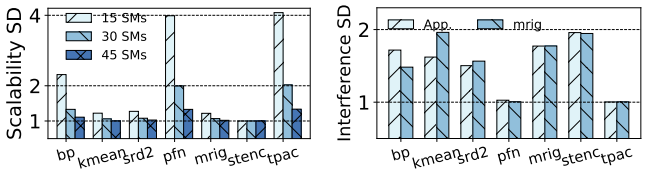
A. Investigation Setup and Metrics

In our investigation, we co-locate applications from two widely-used GPU benchmark suites, Rodinia [5] and Parboil [23], on a spatial multitasking GPU, and evenly allocate the SMs to them. We use GPGPU-Sim [4] to simulate a 60-SM spatial multitasking GPU as the experimental platform following the same implementation reported in prior work [2],



(a) Inefficiency of Bubble-Flux (b) Inaccuracy of ASM

Fig. 1: Cumulative distribution of aggregated performance degradation/slowdown estimation error of Bubble-Flux/ASM.



(a) Scalability slowdown (b) Interference slowdown

Fig. 2: Scalability of the benchmarks and their interference slowdowns when they are co-located with *mrig*.

[25], [28]. The benchmarks have various scalabilities and pressure on different shared resources. More details on the hardware and benchmarks are described in Section VI. The reason we do not perform the investigation on a real-system GPU (e.g., Nvidia V100) is that prior work relies on SM preemption and per-SM event collection. These features are still not supported by real-system GPUs.

To quantify an application's slowdown at co-location, we first run it alone and collect its solo-run execution time T_{solo}. Let T_{colo} represent its execution time when it is co-located with other applications. Equation 1 calculates its *slowdown* (denoted by SD) and normalized *progress* (denoted by PG) at co-location. Note that, $SD \geq 1$ and $0 < PG \leq 1$.

$$SD = \frac{T_{colo}}{T_{solo}}; \quad PG = \frac{T_{solo}}{T_{colo}} = \frac{1}{SD} \tag{1}$$

Based on Equation 1, let $SD_{measured}$ and SD_{pred} represent an application's measured slowdown and predicted slowdown respectively. Equation 2 calculates the error of slowdown prediction (denoted by Err).

$$Err = \frac{|SD_{pred} - SD_{measured}|}{SD_{measured}} \tag{2}$$

Assume k ($k > 1$) applications are co-located on a spatial multitasking GPU. Let PG_1, ..., PG_k denote their progresses calculated in Equation 1. Same to prior work [13], Equation 3 calculates the aggregated application performance ST_{colo}. The larger ST_{colo} is, the better the GPU performs.

$$ST_{colo} = \sum_{i=1}^{k} PG_i \tag{3}$$

Then, **the goal of this work is to minimize slowdown prediction error Err without decreasing ST_{colo}.**

B. Limitations of Prior Work

Fig. 1 shows the inefficiency and inaccuracy of Bubble-Flux [33] and ASM [24] respectively, on predicting the application slowdown at colocation on spatial multitasking GPU.

As shown in Fig. 1(a), the average aggregate performance degradation (compared with the case where no prediction technique is used) across 49 pairwise colocations (more details in Section VI) is 0.27 (dropping from 1.22 to 0.95) when applying Bubble-Flux, making colocation less attractive. Bubble-Flux has large overhead because **its fundamental design logic is to periodically profile each of the co-located applications, which can only be achieved through expensive SM preemption on GPU [25]**.

ASM has low prediction accuracy for application co-location on spatial multitasking GPUs. Observed from Fig. 1(b), the average prediction error of ASM for the 49 co-locations is 45% (up to 105% in the worst case). This is because ASM predicts an application's slowdown only based on the interference on shared cache and memory bandwidth, **ignoring the slowdown due to the reduction of SMs allocated to it**. This design choice is not a problem for CPU co-location because a CPU application uses the same number of cores when it runs alone and when it is co-located with other applications. However, a GPU application uses much fewer SMs (e.g., half of all the SMs) when it is co-located with others on a spatial multitasking GPU.

C. Key Factors that Affect Kernel Slowdown

To better understand why ASM has low accuracy, we first breakdown a kernel k's slowdown at co-location. A GPU application has multiple kernels and we can calculate an application's slowdown based on the slowdowns of its kernels (Section IV). Assume k is co-located with other applications and is allocated n SMs, its slowdown SD in Equation 1 can be reformulated to be Equation 4, where T_n represent k's processing time when it runs alone but only uses n SMs.

$$SD = \frac{T_{colo}}{T_{solo}} = \frac{T_{colo}}{T_n} \times \frac{T_n}{T_{solo}} = SD_{scal} \times SD_{inter} \tag{4}$$

In Equation 4, T_n/T_{solo} is k's slowdown only due to fewer SMs, T_{colo}/T_n is k's slowdown only due to shared resource contention. We refer T_n/T_{solo} and T_{colo}/T_n as *scalability slowdown* (SD_{scal}) and *interference slowdown* (SD_{inter}), respectively. ASM has poor accuracy because it ignores scalability slowdown at co-location on spatial multitasking GPUs.

As an example, for each test application in Table III, Fig. 2(a) shows its scalability slowdown when it runs with 15, 30 and 45 SMs, and Fig. 2(b) shows its interference slowdown when it is co-located with an memory-intensive application *mrig* using 30 SMs. In Fig. 2(b), the x-axis shows the benchmarks, the bar "App." shows its corresponding interference slowdown.

Observed from Euqation 4 and Fig. 2, we identify three key factors that have to be considered to precisely predict a kernel's

slowdown when it is co-located with other applications on a spatial multitasking GPU.

(1) The number of SMs. It seriously affects a kernel's scalability slowdown SD_{scal}. As shown in Fig. 2(a), a kernel runs slower when fewer SMs are allocated to it. Furthermore, **kernels have different scalability slowdowns when they are allocated the same number of SMs.**

(2) Contention on shared resources. A kernel has smaller interference slowdown when the contention on the shared resource is lighter. For instance, *mrig* has smaller interference slowdown when it is co-located with *tpac* that has light "pressure" on shared L2 cache and global memory bandwidth.

(3) Sensitivity to resource contention. Kernels are slowed down by different times due to the contention on shared resources. For instance, *kmeans* and *tpac* have different interference slowdowns when they are co-located with *mrig*. A kernel's sensitivity to resource contention affects its interference slowdown under a given pressure on shared resources.

D. Challenges of Application-level Slowdown Prediction

According to the above analysis, it is non-trivial to precisely predict application-level slowdown at co-location. Specifically, there are three key challenges.

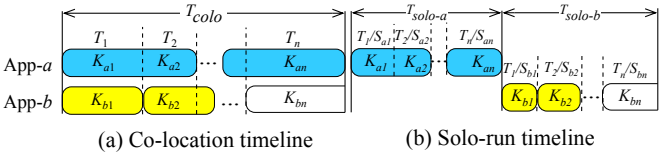
- An application's slowdown at co-location varies. A GPU application often has multiple kernels with various characters, thus suffers from unstable slowdown.
- Precise slowdown prediction has to consider both interference slowdown and scalability slowdown. However, **an application's sensitivity to the shared resource contention and its scalability are not known beforehand.**
- Context switch overhead is heavy on GPU [25]. The technique that profiles each application to collect its characters through frequent context switches on CPU (e.g., Bubble-Flux) [24], [33] is not applicable on GPU. **We have to obtain scalability and sensitivity information of an application online without expensive profiling.**

IV. The Themis Methodology

To address the above challenges, we propose **Themis**, an **efficient** and **precise** application-level slowdown predictor for application co-location on spatial multitasking GPUs with negligible runtime overhead.

Themis predicts the slowdown of a GPU application at co-location in three steps: (1) divides its execution into phases with stable slowdown; (2) predicts its slowdown in each phase; (3) calculates its application-level slowdown. As an example, Fig. 3 shows the execution timelines of *App-a* and *App-b*, when they co-run on a spatial multitasking GPU and when they run alone. Themis predicts their slowdowns simultaneously in a time duration of T_{colo} in Fig. 3(a) as follows.

First, Themis divides T_{colo} into phases of duration $T_1, ..., T_n$, where the break points are at the end of every kernel. In each phase, the slowdowns of both *App-a* and *App-b* are stable, because the characteristics of a kernel is relative stable. Prior work [32] reports similar finding.



(a) Co-location timeline (b) Solo-run timeline

Fig. 3: The scenarios when *App-a* and *App-b* are co-located and when they run alone.

In the second step, Themis predicts *App-a* (and *App-b*)'s slowdown in each phase based on hardware event statistics and a pre-trained kernel slowdown model (Section V). The model is trained to be a neural network based on the observation that hardware event statistics of a kernel, which can be collected online with negligible overhead, are highly correlated with its inherent characteristics (scalability and sensitivity) and the pressure it poses on the shared resources (Section V-A).

Let SD_{ai} and SD_{bi} represent the predicted slowdowns of *App-a* and *App-b* in the i-th phase T_i. In the third step, Equation 5 calculates solo-run execution time of *App-a* and *App-b*. The slowdowns of *App-a* and *App-b* can be calculated to be T_{colo}/T_{solo_a} and T_{colo}/T_{solo_b}.

$$T_{solo\_a} = \sum_{i=1}^{n} \frac{T_i}{SD_{ai}}; \quad T_{solo\_b} = \sum_{i=1}^{n} \frac{T_i}{SD_{bi}} \tag{5}$$

Whenever a kernel is launched or completes, Themis updates the current slowdowns of all the co-located applications following the above three steps.

It is worth nothing that the kernel slowdown mdoel generalizes well for unseen applications since we use representative applications with various characteristics to train the model. If a new class of application is identified, we can enhance the model using the incremental update.

Because the slowdown model in Themis takes hardware event statistics as input, Themis requires to enhance current GPU to support per-SM event collection. We show that the modification is minor in Section V-B.

We further present an SM allocation engine based on Themis that can precisely rein in application slowdown at co-location in Section VII.

V. Modeling Kernel Slowdown

In this section, we present the details of kernel slowdown model (KSM), which is used to predict the slowdown of a kernel's stable phase in Themis. According to the previous key factors analysis (Section III-C), building a precise kernel slowdown model is equivalent of finding the correlation function R in Equation 6. In the equation, N_{sm} is the number of SMs allocated to a kernel, $inher\_characteristics$ is its sensitivity to the contention, $interference$ reflects the level of co-located kernel caused contention on the shared resources.

$$SD = R(N_{sm}, inher\_characteristics, interference) \tag{6}$$

At runtime, N_{SM} is already known, but $interference$ and $inher\_characteristics$ are not available. To solve this problem, we adopt the following steps as shown in Fig. 4.

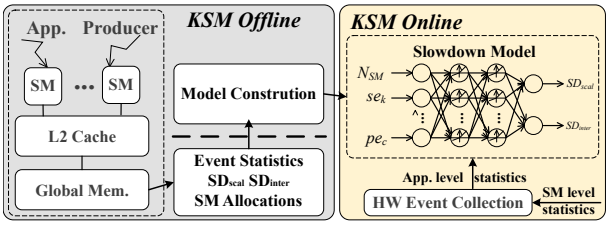


Fig. 4: Design overview of KSM.

(1) First, we identify the hardware events that are strongly correlated with the inherent characteristics and the interference level (Section V-A). These statistics are then used as a good approximation to the corresponding factor in function R.

(2) Then, we collect training samples (Section V-C). We design dedicated pressure producers that can stress various pressures on the shared resource, and co-locate them with representative benchmarks to collect relevant statistics.

(3) Lastly, the SM allocation and the statistics are then used to be inputs and the slowdowns are used as the targets to train a neural network to fit the function R (Section V-D).

At runtime, when multiple applications are co-located, Themis periodically collects the required statistics, feeds the data into the KSM model, and predicts the slowdown of an application in the current time period at co-location.

A. Identifying Correlated Events

A brute force method is incorporating all the available hardware events in the slowdown model. However, **irrelevant events can easily result in overfitting that conveys poor accuracy**. Moreover, the large input dimension also result in long prediction time. Thus, we only use hardware events that are highly correlated with *inher_characteristics* or *interference*.

To measure the pressures (P_{cache} and P_{bw}) a kernel poses on the shared resources, we design dedicated kernels to be pressure meters. Let r represent either the shared L2 cache resource or the global memory bandwidth resource; k_m represent the *pressure meter* for r. We run kernel k and k_m together, and use k_m's slowdown to estimate k's pressure on the resource r. We design k_m such that its performance is only sensitive to the shared resource r, and therefore the slowdown of kernel k_m should reflect amount of resource *r* stressed by its colocated kernel. The pressure meter design will be detailed in Section V-C.

We use *spearman correlation coefficient* [10] that is widely-used in machine learning area, to measure the correlation between a hardware event with a kernel's scalability, sensitivity to shared resource contention, and its pressure on shared resources. The larger the coefficient is, the higher the event is correlated with the scalability, sensitivity, or pressure.

Let ρ_{scal}, ρ_{inter}, ρ_{cache} and ρ_{bw} denote the spearman correlation coefficients between an event and SD_{scal} , SD_{inter}, P_{cache}, P_{bw} respectively. When calculating ρ_{scal} and ρ_{inter}, we co-locate representative applications (Section VI) with the pressure meters under an SM allocation, collect the slowdowns and corresponding event statistics of these applications. The slowdowns and the statistics are then used to calculate ρ_{scal} and ρ_{inter}. We get the coefficients under five SM allocation settings and report the median in Table I. We adopt the same strategy to calculate ρ_{cache} and ρ_{bw}.

TABLE I: Events related to a kernel's inherent characters

Event	Description	$\lvert\rho_{scal}\rvert$, $\lvert\rho_{inter}\rvert$	$\lvert\rho_{bw}\rvert$,$\lvert\rho_{cac}\rvert$
IPC	# of inst./cycle	**0.81**,0.71	0.43,0.36
L1dAcc	# of data access	**0.56**,0.44	0.24,0.21
L1dMiss	# of data miss	**0.63**,0.47	0.48,0.52
L1cAcc	# of const access	0.66,**0.67**	0.39,0.19
L1cMiss	# of const miss	**0.76**,0.70	0.41,0.33
L2Acc*	# of l2 access	**0.63**,0.51	0.42,**0.66**
L2Miss*	# of l2 miss	**0.69**,0.58	**0.78**,0.36
MemBW*	% of bw. usage	**0.76**,0.67	**0.73**,0.31

Let se and pe denote all the events and the starred events in Table I respectively. As shown in the table, se is highly correlated with SD_{scal} and SD_{inter}, and pe is highly correlated with P_{cache} and P_{bw}. Thus, for a kernel *k*, we use its se as an approximation of its inherent characteristics and its pe as an approximation of its pressure. The interference from *k*'s colocated kernels (ϕ_k) can be calculated as the sum of their pressures. We re-formalize the correlation function R in Equation 6, where N_{sm} is the number of SMs allocated to *k*, se_k is *k*'s inherent characteristic related event set, and pe_c is the interference related event set of *k*'s co-located kernels. The input size and output size of R are 12 (1+8+3) and 2 (SD_{scal} and S_{inter}) respectively. The intermediate results (SD_{scal} and SD_{inter}) are useful when reining in application-level slowdown (Section VII).

$$< SD_{scal}, SD_{inter} >= R(N_{sm}, se_k, \sum\nolimits_{c \in \phi_k} pe_c) \qquad (7)$$

B. Hardware Modification

In Equation 7, the input parameters (se and pe) are hardware event statistics caused by each kernel that can be collected online with minor hardware modification. In our design, we enhance GPU to support SM-level event collection, and accumulate the statistics of the SMs allocated to a kernel to be the kernel's event statistics. To enable SM-level event collection, a GPU needs to track the source SM of each memory request. Themis does not need to add extra hardware to track the requests, because modern GPUs use Network-on-Chip (NoC) as the communication infrastructure [17], where the source of each request has already been incorporated into the request to route the response back efficiently.

Themis only need to add at most 8 (the number of events in Table I) additional registers for each SM as the event counters. The storage overhead is $4 \times 8 \times 60$=1920 bytes for a GPU having 60 SMs. The hardware modification overhead is minor.

C. Collecting Training Samples

We implement the enhanced GPU using GPGPU-Sim [4]. Based on the enhanced GPU, we collect hardware event statistics in Table I under different interference levels as training data to train the slowdown model. The interference level is

determined by the contention on shared resources. Simply co-locating real world applications to collect training data results in exponential increase of the required training set size when more than two applications are co-located. For instance, KSM needs to collect training samples in M^N possible colocations if we use M applications to collect training data for a N-workload colocation scenario.

Listing 1: Cache and bandwidth pressure producers

```
/********** Cache pressure producer *********/                    1
// ws: working set size; warp_size: gpu warp size                 2
// Arr: shuffle {0,1,...,ws-1} by warp size                       3
__global__ void cache(int* Arr, int C, ...) {                     4
  int idx = blockDim.x * blockIdx.x + threadIdx.x;                 5
  while(...) {                                                     6
    idx = Arr[idx];rep_pure_comp(C);//rep. C times                 7
 }}                                                                8
/********* Bandwidth pressure producer ********/                  9
//n: number of vectors; dim: dimension of a vector                10
__global__ void bandwidth(int* A, int* B, ...) {                  11
  int idx = blockDim.x * blockIdx.x + threadIdx.x;                 12
  while(...) { for(int i = 0; i < dim; ++i) {                      13
     A[i*n+idx] = B[i*n+idx] + val;                                14
     val = rep_pure_comp(C);// repeat C times                      15
}}}                                                               16
```

To reduce the number of required training samples, we instead co-locate representative applications with the pressure producers that can stress various pressures on the shared resources. In this way, we do not need to collect more training data when the number of concurrent kernels increases. Suppose that we use two types of pressure producers (for shared L2 cache and global memory bandwidth respectively), each of L levels, then we only need to collect training data in $M \times L \times 2$ colocations.

As shown in List 1, we use dedicated kernel as the cache (bandwidth) producer for the corresponding shared resource. By changing the ratio of memory access over computation, the pressure on the shared resources varies. More specifically, if we increase the variable C in Line 7 and Line 15 of Listing 1, the two kernels become more computationally intensive and thus produce smaller pressure on the shared cache and global memory bandwidth, respectively. We use the producer with the smallest C as the pressure meter in Section V-A.

For every kernel k in the training set, we first profile it to collect its solo-run processing time T_n with n SMs ($n = 1, 2, ..., N$), if the target spatial multitasking GPU has N SMs. After that, we co-locate k with a cache (bandwidth) pressure producer k_p, allocate the N SMs to them, record k's execution time T_{colo} at co-location, and collect event statistics (Table I) caused by k and k_p respectively. If n SMs are allocated to k in this co-location, we can get a valid training sample, where the slowdowns ($SD_{scal} = T_n/T_N$, $SD_{inter} = T_{colo}/T_n$) are output labels, the number of SMs allocated to k and the event statistics are input features. By changing the training kernel k, the number of SMs allocated to k, and the pressure of cache (bandwidth) pressure producer, we can collect representative training samples. In current implementation, we use kernels in 15 GPU benchmarks, 12 pressure levels for the cache (bandwidth) pressure producer and 5 SM allocations (i.e., 10:50, 20:40, 30:30, 40:20, 50:10; here 10:50 means 10 SMs are allocated to k and 50 SMs are allocated to the pressure producer).

TABLE II: The GPGPU-Sim setup

Parameter	Configuration
SMs	60 SMs, 32 CTA, 64K Regs, 96K SharedMem
L1 Cache	24K Data, 2K Inst, 6K Texture, 8K Const
L2 Cache	2M, 256K/Memory sub-partition, 8-way
Memory	4MCs, 2 partitions/MC, FR-FCFS, 3500MHz

D. Building Kernel Slowdown Model

Adopting machine learning technique, we train a neural network using the training samples to fit the correlation function R. As shown in Fig. 4, the model consists of an input layer, two hidden layers, and an output layer. For the sake of computation efficiency, we use *leakyRelu* as the activation function. The number of nodes in each hidden layer equals to the number of input dimensions. To train the model, we use Equation 8 as the loss function L.

$$L = L_{scal} + L_{inter} + \alpha L_{colo} + \lambda \|\mathbf{W}\|_2 ,$$
$$where\ L_x = \left\| \frac{SD_x^* - SD_x}{SD_x^*} \right\|_2 , SD_x^* \text{ is ground truth.} \quad (8)$$

The loss function is comprised of four parts: 1) L_{scal}, loss from the scalability slowdown estimation, 2) L_{inter}, loss from the interference scalability estimation, 3) L_{colo}, loss from real slowdown SD estimation, and 4) the regularization term which can reduce the risk of overfitting. When training our slowdown model, we set $\alpha = 2$ and $\lambda = 0.001$.

The slowdown model trained for a GPU cannot be used to predict slowdown at co-location on another type of GPU, because they have different features. When a new generation of GPU is adopted, we can collect samples on it and train it a new model. As we will show in Section VI-B, benefit from the design of pressure producers, the overhead of collecting samples and training slowdown model is moderate.

VI. Evaluation of Themis Accuracy

We evaluate the accuracy of Themis on a 60-SM enhanced GPU. We modified GPGPU-Sim [4] to support spatial multitasking and preemption, and follow the same assumptions and implementation reported in prior work [1], [2], [25], [28]. The reason we do **NOT** evaluate a real-system GPU is that Themis (as well as ASM) requires per-SM performance counters and Bubble-Flux relies on SM preemption. Current GPUs (even Nvidia V100) do not support these features yet. We also add 8 registers for each SM as hardware event counters. Table II summarizes GPGPU-Sim settings.

Table III lists the benchmarks we used in the experiment, where 7 workloads are from Parboil [23] (marked with *) and 15 workloads are from Rodinia [5]. Note that, we have removed duplicated benchmarks in Rodinia and Parboil. We use 15 out of the 22 benchmarks (68%) to train the slowdown model and use the rest 7 benchmarks as the test set. **To evaluate Themis, the training set and the test set are**

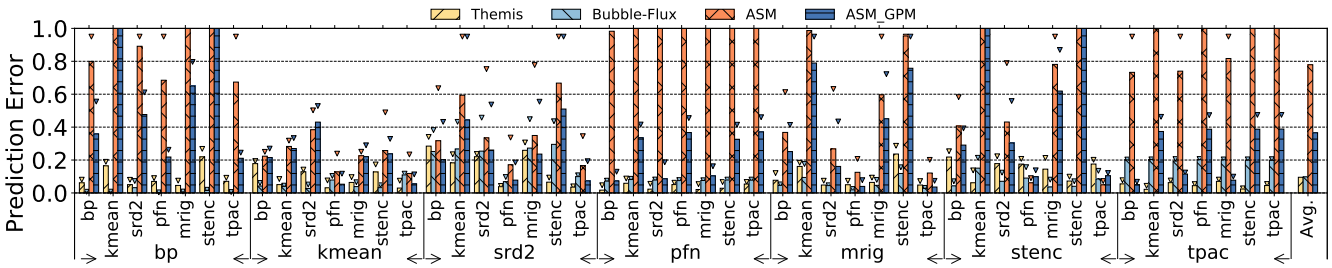


Fig. 5: The slowdown prediction error of the 49 pairwise co-locations under three SM allocation settings.

TABLE III: Benchmarks in train set and test set[1]

	Training set	Test set
Compute intensive	mriq (mri_q), ctcp* (cutcp) wall(heartwall)	pfn (particlefiltern), tpac* (tpacf)
Balanced	bfs, spot (hotspot), srd1 (srdv1), path (pathfinder) leuk (leukocyte)	bp (backprop), srd2 (srdv2)
Memory intensive	eul (euler3d), nn, pff (particlefilterf), lava (lavaMD), lbbm* hist* (histo), gau (gaussian)	mrig* (mri_gridding), kmeans stenc* (stencil)

selected using stratified sampling method [9] in machine learning. Specifically, we divide the 22 benchmarks into three categories: compute-intensive, memory-intensive, and balanced; we randomly select several benchmarks from each category to form the test set. In this way, the test set is representative of benchmarks from all categories.

We compare Themis with Bubble-Flux [33], ASM [24] and ASM-GPM. ASM-GPM captures both scalability slowdown and interference slowdown by combining ASM and GPM [31]. GPM predicts scalability slowdown of an application through clustering-classification method in solo-run scenario. When implementing Bubble-Flux, we alternate phase-in/phase-out stages every 300k cycles to balance accuracy and preemption overhead. For the best accuracy of ASM, we disable cache sampling since it harms the accuracy. For the best accuracy of GPM, we cluster the training set into 8 groups and use the same hardware events in Table I to train the classifier.

A. Accuracy of Themis

We evaluate the accuracy of Themis, Bubble-Flux, ASM, ASM-GPM in all the $7 \times 7 = 49$ pairwise combinations of the 7 test benchmarks under 3 SM allocation settings (10:50, 20:40 and 30:30). Fig. 5 shows their prediction errors. In the figure, each bar shows the average prediction error for a combination of 3 different SM allocation settings while the "$\triangledown$" above the bar indicates the maximum error. For the ease of presentation, we clip the errors to 0 and 1. Observed from this figure, the average prediction errors of Themis for all the co-locations of the three different SM allocation settings is 9.5%. ASM and ASM-GPM have much higher prediction errors compared with Themis. The reason ASM is accurate on CPU but suffers from poor accuracy on GPU is that it ignores scalability slowdown. In addition, although GPM can predict scalability slowdown, it relies on hardware events collected in solo-run mode. Therefore, it cannot precisely predict scalability slowdown at co-location because hardware events are polluted by concurrent applications.

Although Bubble-Flux achieves similar prediction accuracy compared with Themis, our experiment in Section III has shown that Bubble-Flux incurs severe performance loss. The average aggregated performance across all the co-locations drops to 0.95, which is lower than the aggregated performance in sequential execution scenario. Therefore, **Bubble-Flux is not practical for spatial multitasking GPU.** On the contrary, Themis does not harm the aggregated performance.

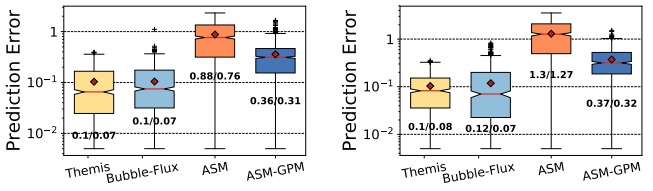
B. Overhead Analysis

The runtime overhead of Themis is low. As shown in Table IV, the kernel slowdown model in Themis has overall 312 parameters, each of 4 bytes. The storage overhead of the model is 1248 bytes. The model is small enough to be fitted into L1 cache. Moreover, single prediction only needs $\sim 0.3K$ float point multiplications. It is lightweight enough to be used online. At runtime, slowdown prediction does not interfere with the applications executionthe and the prediction can be performed on microcontroller [14] in GPU (within 1 us). The overhead of collecting event statistics on GPU using hardware counters is also small [20].

Meanwhile, the overhead of building our kernel slowdown model is moderate, because we only co-locate training applications with two pressure producers to collect training samples. Moreover, Themis only needs to collect training samples in several SM allocation settings (5 settings are used in our model) and relies on the neural network to interpolate the performance under other settings. Section VI-D reports Themis's sensitivity to the number of SM allocation settings used in training the model. Once the training samples are collected, we can train the slowdown model in 10 minutes.

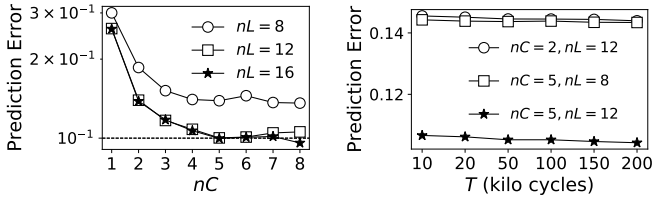
TABLE IV: The size of each layer in the slowdown model and the amount of computation per prediction.

	In	**Hidden1**	**Hidden2**	**Out**	**All**
input size	12	12	12	2	-
weight size	-	12×12	12×12	12×2	312
multiplication	-	144+12	144+12	24+2	338



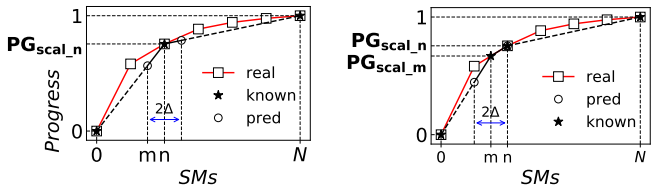
(a) 3-app co-locations (b) 4-app co-locations

Fig. 6: Prediction errors of Themis, Bubble-Flux, ASM, and ASM-GPM for 3-application and 4-application co-locations.



(a) nC and nL (b) Event collecting period

Fig. 7: Themis's sensitivity to nC, nL, and the length of predicting period (T) respectively.



(a) Two-segment regression (b) Multi-segment regression

Fig. 8: Approximating scalability progress curve of a kernel using piecewise linear regression [19].

C. Beyond Pair-wise Co-location

It is possible that more than two applications are co-located on the same spatial multitasking GPU. As such, we also evaluate the case of using Themis to predict application slowdown for all 3-application co-locations ($7^3 = 343$) and 150 randomly picked 4-application co-locations (the SMs are evenly allocated to the co-located applications).

Fig. 6 shows prediction errors of Themis, Bubble-Flux, ASM and ASM-GPM in these co-locations. Observed from this figure, the average/median prediction errors of Themis are 10%/7% and 10%/8% for 3-application co-locations and 4-application co-locations. Bubble-Flux achieves only slightly worse accuracy than Themis, but it reduces aggregated performance to 0.95 in both scenarios.

D. Sensitivity to Hyper Parameters

We also perform sensitivity to the hyper parameters for collecting training samples in Themis: the number of pressure levels (nL) and the number of SM allocation settings (nC). Fig. 7(a) shows their impacts on the accuracy of Themis, where the prediction error of Themis drops when nC and/or nL increases. This is because the model is trained with more valid samples when nC and/or nL increases. However, the predication accuracy plateaus with 12 levels in the cache (bandwidth) pressure producers, and 5 SM allocation settings. Our current model is trained with $nC = 5$ and $nL = 12$.

When predicting slowdown, we can also collect event statistics periodically in the middle of kernel execution. Fig. 7(b) shows the sensitivity of Themis to the predicting period (denoted by T). Observed from the figure, T does not affect Themis's accuracy significantly, because the characteristics of a kernel is relative stable [32]. According to this observation, we can further divide a stable kernel overlap phase into multiple periods. In this case, our SM allocation engine (to be introduced in Section VII) can identify unfair sharing or QoS violation earlier and re-allocate SMs accordingly.

To conclude, Themis is more accurate and more efficient than state-of-the-art techniques for predicting application slowdown at co-location on spatial multitasking GPUs.

VII. Reining in Application Slowdown

In this section, we demonstrate the proactive slowdown management based on our derived prediction model. Specifically, we design an SM allocation engine based on Themis, which can rein in application slowdown at co-location for the purposes of enforcing fair sharing or QoS target. The engine periodically uses Themis to predict each application's slowdown. It compares the predicted slowdown with user determined threshold (e.g., fair sharing or QoS) and dynamically adjusts the SM allocation. *The key challenge is how to predict the slowdown of an application with an un-tried SM allocation because event statistics under an un-tried SM allocation are not available.* As such, we augment Themis with the interpolation ability to handle the un-tried SM allocation case.

A. Proactive Slowdown Prediction

We augment Themis with a *Proactive Slowdown Predictor* (**PSP**) to address the above challenge. PSP is based on our finding that the slowdown of a kernel is compose of scalability slowdown and interference slowdown in Section III-C. The scalability slowdown is a kernel's inherent characteristics, for which we use piecewise linear regression [19] for proactive prediction. Meanwhile, a kernel's interference slowdown is also affected by the co-located applications, for which we use the current interference slowdown to approximate it under a similar SM allocation.

Suppose a kernel k is currently allocated n SMs. Let SD_{inter_n} (SD_{scal_n}) denote its current interference (scalability) slowdown predicted with Themis. Equation 9 calculates the slowdown of k if it is allocated m SMs (denoted by SD_m). In this equation, SD_{scal_m} (SD_{inter_n}) is k's scalability (interference) slowdown with m SMs. Equation 9 is valid when $|m - n| \leq \Delta$: when Δ is small (e.g., 3.), SD_{inter_m} and SD_{inter_n} are close because of similar the pressure on

the shared resources.

$$SD_m = SD_{scal\_m} \times SD_{inter\_m} \approx SD_{scal\_m} \times SD_{inter\_n} \quad (9)$$

In Equation 9, SD_{inter_n} can be predicted with Themis directly. PSP adopts piecewise linear regression to accurately predict SD_{scal_m}, and therefore SD_m. The red curve in Fig. 8 shows the scalability of kernel k in term of scalability progress, PG_{scal} (reciprocal of scalability slowdown). N is the overall number of SMs in the GPU, and PG_{scal_n} is scalability progress of k when it is allocated n SMs (obtained with Themis). Because $0 \leq PG_{scal} \leq 1$, we approximate the curve using two-segment piecewise linear regression as shown in Fig. 8(a). The two segments are from point [0,0] to [n, PG_{scal_n}], and from [n, PG_{scal_n}] to [N,1] respectively. Based on this approximation, the scalability progress of k with m SMs, denoted by PG_{scal_m}, can be calculated in Equation 10. Then, $SD_{scal\_m} = 1/PG_{scal\_m}$.

$$PG_{scal\_m} = \begin{cases} \frac{PG_{scal\_n} \times (m-n)}{n} + PG_{scal\_n}, & if\ m < n \\ \frac{(1-PG_{scal\_n}) \times (m-n)}{N-n} + PG_{scal\_n}, & if\ m > n \end{cases} \quad (10)$$

At runtime, we record scalability progress information and further improve the prediction accuracy. This is achieved by extending the segments in the piecewise linear regression. For instance, after we allocate m SMs to kernel k, its real PG_{scal_m} is known. In this case, as shown in Fig. 8(b), we can further break the segment from point [0, 0] to [n, PG_{scal_n}] into two segments: one from [0, 0] to [m, PG_{scal_m}], and one from [m, PG_{scal_m}] to [n, PG_{scal_n}]. The more segments we use, the more accurate piecewise linear regression approximates scalability progress curve.

A naive method is to approximate the actual progress (reciprocal of the slowdown at colocation) curve instead of the scalability progress. However, there is no such a convex progress curve to be approximated, because the actual progress of a kernel depends on both the number of its SMs and the co-located kernel. For instance, k may have smaller progress when it is allocated more SMs but is co-located with L2 cache (memory) intensive kernels.

B. SM Allocation Engine

Based on Themis and PSP, our SM allocation engine reins in application slowdown to fulfill user requirement. Algorithm 1 shows the common algorithm used in the engine.

Suppose m applications p_1, ..., p_m are co-located on a spatial multitasking GPU. Let π=<n_1, n_2,...,n_m> denote the current SM allocation, where n_i ($i = 1, 2, ...m$) is the number of SMs allocated to p_i. Algorithm 1 tries to find a better allocation π'=<n'_1, n'_2,...,n'_m> that increases *allocation score* (line 14 in Algorithm 1). The score is calculated according to user-defined policy (e.g. fair sharing or QoS guarantee).

In more details, SM allocation engine goes through all pairs of p_i and p_j ($1 \leq i, j \leq m$), and examines whether reallocating one SM from p_i to p_j would increase the score. If so, the engine updates the score and the identified SM allocation π'. Note that the above search may run multiple rounds (line 5). The algorithm stops when the criterion is satisfied, the score fails to increase, or the distance between π' and π is larger than $maxDist$, which equals to Δ in PSP. The distance between π and π' is defined to be $\max_{1 \leq \forall i \leq m}\{|n_i - n'_i|\}$. If the distance is too large, the proactive slowdown prediction of PSP is not accurate as discussed in Section VII-A, thus may result in inappropriate SM reallocation. We investigate the sensitivity to $maxDist$ in Section VII-C.

Algo. 1 Algorithm of identifying a better SM allocation

```
1: Input π : {original allocation}
2: Input sd: {original slowdown list}
3: m ← # active kernels ; π' ← π {current allocation}
4: sd' ← sd {current slowdown list}
5: while ‖π', π‖ < maxDist do
6:     if Criterion(π', sd') then
7:         return π'
8:     no_improvement ← True
9:     for i ← 1, ..., m do
10:        for j ← 1, ..., m do
11:            π'' ← π' {temporal allocation list}
12:            π''[i] ← π''[i] − 1; π''[j] ← π''[j] + 1
13:            sd'' ← PSP(π'')
14:            if Score(π'', sd'') > Score(π', sd') then
15:                π' ← π''; sd' ← sd''
16:                no_improvement ← False
17:    if no_improvement then
18:        return π'
19: return π'
```

C. Case Study 1: Enforcing Fair Sharing

In the first case study, we enforce weighted fair sharing by implementing a fairness-aware SM allocation policy, **Themis-Fair**, in SM allocation engine. A GPU is considered to be fairly shared by multiple applications, if their slowdowns are proportional to the reciprocals of their weights. Assume m applications p_1, ..., p_m are co-located, and their weights are w_1, ..., w_m ($\sum_{i=1}^{m} w_i = 1$). Let s_1, ..., s_m represent their slowdowns at co-location. Equation 11 calculates *unfairness* of the sharing. If *unfairness*=0, the GPU is fairly shared.

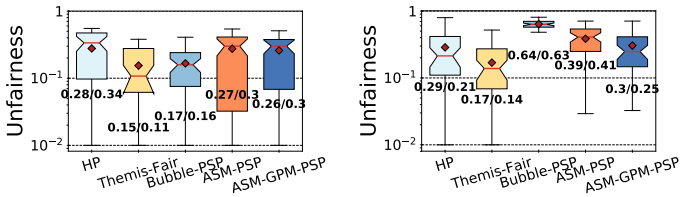
$$unfairness = \frac{\max\limits_{1 \leq \forall i \leq m}\{s_i \times w_i\} - \min\limits_{1 \leq \forall i \leq m}\{s_i \times w_i\}}{\max\limits_{1 \leq \forall i \leq m}\{s_i \times w_i\}} \quad (11)$$

Algorithm 2 shows the criterion and score functions defined in Themis-Fair. We compare Themis-Fair with Bubble-PSP, ASM-PSP, ASM-GPM-PSP that combines Bubble-Flux, ASM, ASM-GPM with our SM allocation engine. We also compare Themis-Fair with the static weighted even allocation policy HP [1]. By default, we use $maxDist$=5 and $thres$=0.1.

Algo. 2 Criterion and Score Functions in Themis-Fair

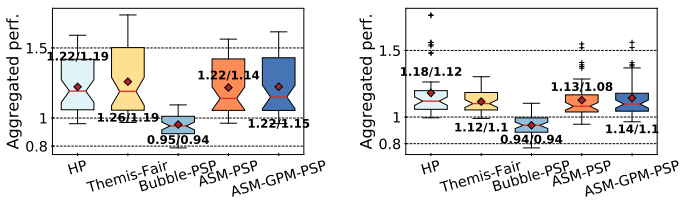
```
1: Function Criterion(π, sd):
2:     return unfairness(sd) < thres
3: Function Score(π, sd):
4:     return −unfairness(sd)
```

Unfairness. For all the 49 pair-wise co-locations in two weight configurations (<0.5, 0.5>, <0.2, 0.8>), Fig. 9 shows the resulted unfairness when we allocate SMs using HP, Themis-Fair, Bubble-PSP, ASM-PSP and ASM-GPM-PSP. Observed from the figure, Themis-Fair achieves the smallest unfairness in all the configurations. For configuration



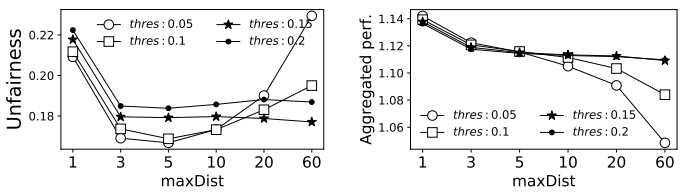
(a) <0.5,0.5> (b) <0.2,0.8>

Fig. 9: Distribution of unfairness in all the 49 pair-wise co-locations with different weight configurations.



(a) <0.5,0.5> (b) <0.2,0.8>

Fig. 10: Aggregated performance of all the 49 pair-wise co-locations with different weight configurations.



(a) Unfairness (b) Aggregated performance

Fig. 11: Impact of *maxDist* and *thres* in Themis-Fair on unfairness and aggregated performance.

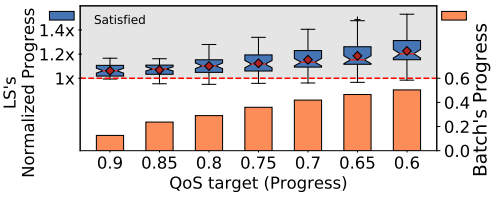


Fig. 12: QoS of LS applications and aggregated progress of batch applications with Themis-QoS.

<0.5, 05>, the average/median unfairness with Themis-Fair is 0.15/0.11 (very close to the *thres* 0.1), and it is much smaller than the unfairness with HP, Bubble-PSP, ASM-PSP and ASM-GPM-PSP. Themis-Fair reduces the average unfairness by more than 47% compared with HP. For configuration <0.2, 0.8>, we observe that the unfairness with Bubble-PSP increases significantly to 0.64. This is because the unfairness in solo-run profiling stage of Bubble-PSP increases when the difference between the weights is large.

Aggregated Performance. Fig. 10 shows the aggregated application performance when we use HP, Themis-Fair, Bubble-PSP, ASM-PSP and ASL-GPM-PSP to allocate SMs. Observed from the figure, only Bubble-PSP reduces the aggregated performance compared with no co-location scenario. The inefficiency of Bubble-PSP results from the preemption-based profiling as analyzed in Section VI.

Sensitivity. Fig. 11 shows the impact of *maxDist* and *thres* in Themis-Fair. Observed from the figure, *maxDist=5* results in the smallest unfairness in all the cases. If *maxDist* is smaller (e.g., =1), Themis-Fair may need many adjust rounds before achieving fair sharing. On the contrary, if *maxDist* is larger (e.g., =20), the proactive slowdown prediction is not accurate, resulting inappropriate SM allocation. Meanwhile, when *maxDist* is small (< 5), *thres* has little impact on the aggregated performance. If *maxDist* is large, a large *thres* may improve the aggregated performance. This sensitivity study suggests us to set *maxDist*=5 and *thres*=0.1.

D. Case Study 2: Guaranteeing QoS

We also study how to use Themis to guarantee QoS by implementing a QoS-aware SM allocation policy, **Themis-QoS**, whose goal is to ensure that the progress of latency-sensitive (LS) application is larger than a given QoS target PG_{target} while maximizing the progress of batch applications with no QoS requirement at co-location [26], [33]. Note that we assume the solo-run time of a QoS kernel is known such that we can convert its QoS target to the progress target, similar in [29]. For example, assume that the latency QoS target is 200 ms and the solo-run execution is 100 ms, then the progress target is 0.5. Similar to the fair sharing case, Themis-QoS monitors its slowdown online and adjusts the SM allocation accordingly if the QoS target is violated.

We evaluate Themis-QoS with all the 49 pair-wise co-locations under seven QoS targets ranging from 0.6 to 0.9. Fig. 12 shows the progress of LS applications normalized to QoS targets (upper part), and the aggregated progress of batch applications (lower part) at co-location. If the normalized progress of an LS application is larger than 1, its QoS is satisfied. Observed from this figure, the QoS of LS applications in almost all the co-locations is satisfied with Themis-QoS. In the worst case, LS applications in 8.2% of the co-locations suffer from slightly QoS violation (violated less than 5%). Meanwhile, Themis-QoS also improves aggregated performance of batch applications when the QoS target decreases.

In summary, SM allocation engine based on Themis and proactive slowdown prediction can precisely rein in application slowdown to fulfill user requirements.

VIII. Conclusions

We present Themis, a precise and efficient application-level slowdown predictor for application co-location on spatial multitasking GPUs. Based on Themis, we further propose PSP to predict the slowdown of an application with a new SM allocation proactively, and implement an SM allocation engine

to rein in application slowdown through SM reallocation. Our evaluation shows that Themis can predict application slowdown with the error smaller than 9.5%. Meanwhile, the SM allocation engine can precisely rein in application slowdown.

IX. Acknowledgement

This work is partially sponsored by the National Basic Research 973 Program of China (No. 2015CB352403), the National Natural Science Foundation of China (NSFC) (61602301, 61632017, 61702328, 61702329, 61872240).

References

[1] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for gpgpu spatial multitasking," in *International Symposium on High Performance Computer Architecture*. IEEE, 2012, pp. 1–12.

[2] P. Aguilera, K. Morrow, and N. S. Kim, "Qos-aware dynamic resource allocation for spatial-multitasking gpus," in *Asia and South Pacific Design Automation Conference*. IEEE, 2014, pp. 726–731.

[3] Amazon, "Amazon ec2 elastic gpus," 2017, https://aws.amazon.com/ec2/Elastic-GPUs/.

[4] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009, pp. 163–174.

[5] 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 *International Symposium on Workload Characterization*. IEEE, 2009, pp. 44–54.

[6] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 17–32.

[7] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2016, pp. 681–696.

[8] Q. Chen, L. Zheng, M. Guo, and Z. Huang, "Eewa: Energy-efficient workload-aware task scheduling in multi-core architectures," in *International Parallel and Distributed Processing Symposium Workshops*. IEEE, 2014, pp. 642–651.

[9] W. G. Cochran, *Sampling techniques*. John Wiley & Sons, 2007.

[10] G. W. Corder and D. I. Foreman, *Nonparametric statistics: A step-by-step approach*. John Wiley & Sons'14.

[11] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[12] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2010, pp. 335–346.

[13] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE micro*, vol. 28, no. 3, 2008.

[14] Y. Fujii, T. Azumi, N. Nishio, and S. Kato, "Exploring microcontrollers in gpus," in *Asia-Pacific Workshop on Systems*. ACM, 2013, p. 2.

[15] C. Iancu, S. Hofmeyr, F. Blagojević, and Y. Zheng, "Oversubscription on multicore processors," in *International Symposium on Parallel & Distributed Processing*. IEEE, 2010, pp. 1–11.

[16] M. Jahre and L. Eeckhout, "Gdp: Using dataflow properties to accurately estimate interference-free performance at runtime," in *International Symposium on High Performance Computer Architecture*. IEEE, 2018, pp. 296–309.

[17] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, "A detailed and flexible cycle-accurate network-on-chip simulator," in *International Symposium on Performance Analysis of Systems and Software*. IEEE, 2013, pp. 86–96.

[18] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *International Symposium on Microarchitecture*. ACM, 2011, pp. 248–259.

[19] V. E. McZgee and W. T. Carleton, "Piecewise regression," *Journal of the American Statistical Association*, vol. 65, no. 331, pp. 1109–1124, 1970.

[20] Nvidia, "Nvidia profiler user's guide," 2018, https://docs.nvidia.com/cuda/profiler-users-guide/.

[21] J. J. K. Park, Y. Park, and S. Mahlke, "Dynamic resource management for efficient utilization of multitasking gpus," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 527–540.

[22] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *International Symposium on Computer Architecture*. IEEE, 2016, pp. 267–278.

[23] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[24] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *International Symposium on Microarchitecture*. IEEE, 2015, pp. 62–75.

[25] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *International Symposium on Computer Architecture*. IEEE, 2014, pp. 193–204.

[26] Y. Ukidave, X. Li, and D. Kaeli, "Mystic: Predictive scheduling for gpu based cloud servers using machine learning," in *International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 353–362.

[27] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog, "Efficient and fair multi-programming in gpus via effective bandwidth management," in *International Symposium on High Performance Computer Architecture*, 2018, pp. 247–258.

[28] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing," in *International Symposium on High Performance Computer Architecture*. IEEE, 2016, pp. 358–369.

[29] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Quality of service support for fine-grained sharing on gpus," in *International Symposium on Computer Architecture*. ACM, 2017, pp. 269–281.

[30] Z. Wang, L. Zheng, Q. Chen, and M. Guo, "Cap: co-scheduling based on asymptotic profiling in cpu+ gpu hybrid systems," in *International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 2013, pp. 107–114.

[31] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *International Symposium on High Performance Computer Architecture*. IEEE, 2015, pp. 564–576.

[32] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming," in *International Symposium on Computer Architecture*. IEEE, 2016, pp. 230–242.

[33] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 607–618.

[34] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 382–393.

[35] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers," in *International Symposium on Microarchitecture*. IEEE, 2014, pp. 406–418.

[36] X. Zhao, Z. Wang, and L. Eeckhout, "Classification-driven search for effective sm partitioning in multitasking gpus," in *International Conference on Supercomputing*. ACM, 2018, pp. 65–75.

[37] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2014.

[38] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2010, pp. 129–142.