Run-Time Performance Estimation and Fairness-Oriented Scheduling Policy for Concurrent GPGPU Applications

Qingda Hu, Jiwu Shu*, Jie Fan and Youyou Lu
Department of Computer Science and Technology, Tsinghua University
Tsinghua National Laboratory for Information Science and Technology
*Correponding Author: shujw@tsinghua.edu.cn
{hqd13, fanj11}@mails.tsinghua.edu.cn, luyouyou@tsinghua.edu.cn

***Abstract*—In order to satisfy the competition of multiple GPU-accelerated applications and make full use of GPU resources, a lot of previous works propose spatial-multitasking to execute multiple GPGPU applications simultaneously on a single GPU device. However, when adopting the spatial-multitasking framework, the inter-application interference may slow down different applications differently, leading to the unreasonable allocation of shared resources among concurrent GPGPU applications, degrading system fairness severely and resulting in sub-optimal performance. Thus, it is imperative to develop mechanisms to control negative inter-application interactions and utilize shared resources fairly and efficiently.**

Quantitatively estimating application slowdowns can enable us to accurately minimize system unfairness. Although several previous works pay attention on showdown estimation for CPUs, we find that they may be inaccurate for GPUs. Therefore, we propose a novel Dynamical Application Slowdown Estimation (DASE) model to estimate application slowdowns accurately. Our evaluations show that DASE has significantly lower estimation error (only 8.8%) than the state-of-the-art estimation models (36.3% and 32.8%) across all two-application workloads. Furthermore, to verify the effectiveness of our DASE model, we leverage our model to develop an efficient fairness-oriented Streaming Multiprocessors (SM) allocation policy DASE-Fair to minimize the overall system unfairness. Compared to the even SM partition policy, DASE-Fair improves fairness dramatically by more than 16.1% on average.

***Index Terms*—GPGPUs; Fairness; Performance Estimation Model; Memory System**

I. Introduction

With the development of General Purpose Graphics Processing Units (GPGPUs), a huge number of GPU-accelerated applications have appeared ranging from hand-held devices to supercomputers. However, this trend results in the fact that multiple applications may contend with each other for a shared GPU device [13]. In addition, because computing ability and hardware resources of GPUs are growing rapidly, many irregular GPGPU applications with limited parallel processing capability cannot fully utilize resources [1], [11], [18]. To solve these problems, several previous works propose a new spatial-multitasking computing paradigm to execute multiple kernels or applications[1] simultaneously on a single GPU device [1], [2], [3], [4], [11], [12], [13], [18], [19], [24].

As shown in Figure 1, the baseline GPU architecture consists of a set of Streaming Multiprocessors (SM). Traditionally, only one application is allowed to execute at a time so that it monopolizes all SMs [6]. Oppositely, spatial multitasking distributes multiple applications to different sets of SMs to execute them concurrently, which differs from the conventional temporal multitasking on GPUs which subdivides time to access GPU resources sequentially. Spatial multitasking has been shown to not only improve resources utilization and the overall system performance, but also provide higher system responsiveness [1].

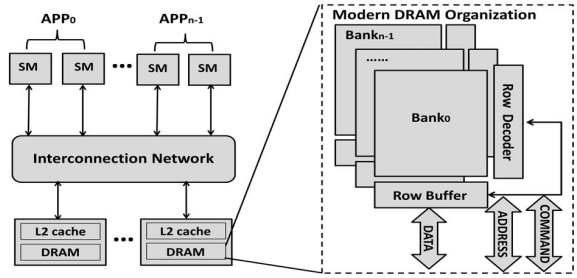


Fig. 1: The baseline GPU architecture

When multiple applications run simultaneously, they share hardware resources, such as the interconnection network, the memory system and so on. However, because conventional network and memory system designs of GPUs are application-agnostic, our experiments in Section 3 show that sharing the computing and memory resources among concurrent applications often results in the unreasonable allocation of available shared resources, degrading the overall system fairness severely. For instance, if an application consumes higher memory bandwidth, other concurrent applications will be penalized to experience a serious slowdown or starvation. Note that evenly allocating the same number of SMs among applications does not guarantee equal slowdown to each application, because it does not consider different application characteristics. Thus, it is important to control the negative inter-application interactions and utilize shared resources fairly and efficiently.

Because the slowdown of each application heavily depends on other simultaneously executing applications and available

[1]For simplicity and without loss of generality, the terms application and kernel are used interchangeably as we assume that concurrent kernels derive from separate applications in this paper.

2332-5690/16 $31.00 © 2016 IEEE
DOI 10.1109/ICPP.2016.14

resources, the inter-application interference has different effects on the performance of different applications [11], [12]. Thus, the ability to estimate accurate slowdowns of concurrent applications is gaining more attention in today's era of simultaneous execution of multiple applications. Prior works [22], [23] propose accurate interference estimation models, and use estimated slowdowns to manage resource allocation, to enable Quality of Service (QoS) and fairness guarantees on CPUs. But these models have focused on estimating application slowdowns on CPUs, we observe that they are inaccurate on GPUs, and there is little work on predicting the slowdowns of individual kernels on GPUs.

Our goal in this paper is to design an accurate performance estimation model for concurrent GPGPU applications to dynamically detect system unfairness, called the Dynamical Application Slowdown Estimation (DASE) model. Quantitatively estimating application slowdowns can enable several mechanisms to achieve QoS guarantees and minimize unfairness more accurately. In order to demonstrate the effectiveness of the DASE model, we propose a simple fairness-oriented SM partition policy based on the predicted slowdowns, called DASE-Fair, which dynamically and efficiently reassigns the SMs among concurrent GPGPU applications, to improve system fairness and performance. To the best of our knowledge, DASE is the first work to provide a run-time performance estimation model for concurrent GPGPU applications. Major contributions of this work are summarized as follows.

- We quantify the effect of the inter-application interference on shared resources, and put forward a new model to estimate the slowdowns of concurrent GPGPU applications.
- We qualitatively and quantitatively compare the accuracy of our DASE model with two latest application-slowdown estimation models on CPUs, including MISE [23] and ASM [22]. Our experiments show that estimation results of our DASE model deviate from the actual slowdown by only 8.8% while MISE's is 36.3% and ASM's is 32.8% across all two-application workloads.
- We demonstrate the effectiveness of our model by designing and evaluating a simple SM management policy based on DASE. Compared with the even SM allocation policy, it improves fairness by more than 16.1% on average and performance by more than 3.7% on average.

II. BACKGROUND

A. Baseline GPU Architecture

As shown in Figure 1, a typical GPU architecture consists of three main components: a collection of SMs, an interconnection network and a set of memory partitions.

Streaming Multiprocessors. The Streaming Multiprocessors (SM) perform parallel data processing and generate a stream of memory requests to memory partitions. GPU kernels organize a huge number of threads into a set of independent thread blocks. Multiple thread blocks of the same kernel can execute concurrently on each SM, and they require the same hardware resources. All threads within the same thread block can execute simultaneously, and cooperate with each other through shared memory and barrier synchronization. The number of thread blocks and the number of threads in each thread block are specified by the programmer.

At launch time, thread blocks are allocated to each SM until available resources of each SM are fully utilized. When a thread block is allocated to a SM, it remains resident in this SM until all its execution work is finished. When all warps of a thread block are finished, the SM driver assigns a new thread block to the SM to occupy the remaining available resources.

Memory Partitions. Each memory partition has a L2 cache and a DRAM memory subsystem. The modern DRAM memory organization consists of a set of banks, which can be regarded as arrays of DRAM cells, organized as rows and columns. To serve a memory request, the corresponding row containing the data is sensed and latched into the row buffer. If subsequent requests access the same row, the latched row can be served faster without accessing the DRAM array. Moreover, multiple banks can process outstanding memory requests in parallel (e.g., a bank is transferring data while another is activating a row). Bank-Level Parallelism (BLP) of an application is the average number of banks which execute memory requests simultaneously when it has at least one outstanding request. Promoting BLP can hide memory latency and increase memory bandwidth utilization.

In addition, a set of address, command and data buses are shared by all banks, and each bus is occupied by only one bank at a time. The memory controller holds outstanding requests in the request buffer, and serves the next request when it satisfies the DRAM timing constraints and does not cause the bank and the address/command/data buses conflicts.

B. Spatial Multitasking

Concurrent execution of multiple GPGPU kernels. Traditional GPUs only support temporal multitasking, which subdivides time among multiple applications to execute them on the full set of GPU resources sequentially. However, previous works [1], [11], [18] observe that many irregular GPGPU applications with limited parallel processing capability cannot fully utilize hardware resources. In addition, multitasking applications in multi-core CPUs may contend for a single GPU device, especially in today's mobile devices, which may need high system responsiveness.

Current modern GPUs (Fermi [15] and Kepler [16]) provide a basic spatial multitasking framework to enable concurrent kernels from different applications to run simultaneously on a single GPU. However, the details of allocating hardware resources among multiple kernels are not publicly documented. Previous works [3], [4], [18] suggest that the LEFTOVER policy is most likely to be used by current GPUs, which launches a next kernel only when there are enough remaining resources after the previous kernel was issued to the GPU. However, it is a poor scheme because it cannot ensure that different applications will always run simultaneously [18]. Thus, several works [1], [2], [3], [4], [11], [12], [13], [18], [19], [24] propose a flexible spatial multitasking to assign different parts of SMs to multiple applications, which enable them to run concurrently on the GPU.

Adriaens et al. [1] observe that some GPGPU applications fail to fully utilize GPU resources, and propose spatial

multitasking to improve hardware resources utilization and system performance. Liang et al. [13] propose a software-hardware solution to improve the efficiency of spatial-temporal multitasking. Pai et al. [18] convert kernels into elastic kernels to control over resource usage at find-grained. Tanasic et al. [24] design a set of hardware modifications to enable concurrent execution of multiprogrammed GPU workloads. Park et al. [19] propose a collaborative preemption approach to reduce substantial overhead induced by the context switch of preemptive multitasking. However, none of these works focus on solving the inter-application interference.

Resource management on GPUs. Many prior works propose several resource management policies to reduce the inter-application interference. Jog et al. [11] propose an application-aware memory scheduler, which serves memory requests of different applications in a round robin fashion, to avoid severe starvation induced by high row-buffer locality and memory-intensity applications. Jog et al. [12] develop two memory scheduling policies to improve instruction throughput and weighted speedup, respectively. However, although they reduce the negative interference among applications in the memory level, they do not fully address the fairness problem [4], nor do they handle the slowdown estimation model as we propose in this paper.

Aguilera et al. [3] propose a dynamic SM allocation mechanism to achieve the QoS goal. Aguilera et al. [4] propose a runtime technique to allocate SMs among concurrent applications to improve fairness. However, all these works need to run each application in advance, and use isolated kernel profiling information to compute application slowdowns. However, data-dependent applications may receive different input data, so their performance cannot be profiled in advance [2].

III. MOTIVATION

In this section, we show that the inter-application interference may lead to severe unfairness, and we point out the reason for it. In addition, we briefly introduce previous works on slowdown estimation.

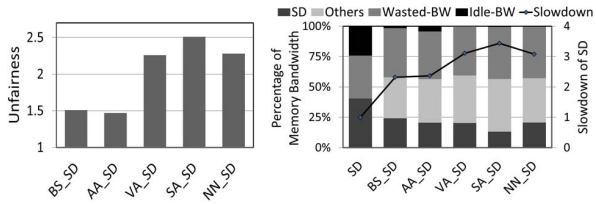
A. Observation

The slowdown of each application and the overall system unfairness are defined as the following equations [5], [11], [14], [22], [23]. IPC_i^{shared} is the average number of executed instructions when i_{th} application is running with other applications. IPC_i^{alone} is the average number of executed instructions when i_{th} application is running alone on the entire GPU without the interference from other applications. Note that an application will use **all SMs** when running alone.

$$Slowdown_i = \frac{IPC_i^{alone}}{IPC_i^{shared}} \quad (1)$$

$$Unfairness = \frac{MAX\{slowdown_i\}}{MIN\{slowdown_i\}} \quad (2)$$

For a better understanding of severe unfairness among applications, we execute five two-applications combinations concurrently (Section 5 describes our experimental configurations). When the overall system is completely fair, all application slowdowns are the same so that the ideal value of unfairness is 1. However, as the fourth bar in Figure 2(a) shows, unfairness is 2.51 when adopting the baseline GPU architecture. That is, the slowdown of SD is 3.44 and the slowdown of SA is only 1.37 (not visible in Figure). Hence, we conclude that SD is seriously affected by others.



(a) Unfairness of two-applications combinations (b) DRAM bandwidth utilization distribution

Fig. 2: Unfairness of workloads

In order to find out the reason for severe unfairness, we decompose the DRAM bandwidth into these parts: (A) SD (Others): the corresponding portion of occupied bandwidth for SD (other applications); (B) Wasted-BW: the corresponding portion of occupied bandwidth for satisfying DRAM timing constraints while no data is transferred; (C) Idle-BW: the corresponding portion of occupied bandwidth when DRAM is idle. Besides, we plot SD when it is running alone. As the fifth bar in Figure 2(b) shows, SD obtains a unfair portion of bandwidth (13%) compared with that (40.5%) when SD is running alone, accounting for only 32.1% (40.5%/13.3%), close to the slowdown of SD 3.44 (1/32.1%=3.12). This imbalanced allocation leads to SD's severe performance degradation. We observe that the slowdowns of a wide range of applications also have this feature. Thus, we believe that the major cause for unfairness may be the unfair allocation of the DRAM bandwidth. A.Jog et al [11] also manifest this phenomenon.

As shown in Figure 2, the sensitivity to shared resources of individual applications is different, and the slowdown of each application is different heavily upon the co-executing applications. Therefore, we design a memory-interference induced slowdown estimation model in Section 4.

B. Prior Work on Slowdown Estimation

Accurately predicting application slowdowns can enable us to design a more reasonable and intelligent shared resources allocation policy to provide fairness and QoS guarantees. For instance, the SM driver can use accurate estimated slowdowns to design a dynamical fairness-oriented SM allocation policy, as we will demonstrate in Section 7.

Slowdown Estimation Models on CPUs. Recently, both the Memory-interference Induced Slowdown Estimation (MISE) and the Application Slowdown Model (ASM) are proposed to estimate application slowdowns accurately on CPUs, based on two observations: 1) the performance of a memory-intensive application is roughly proportional to its memory request service rate, and 2) assigning memory requests of an application the highest priority over all other applications' requests in accessing main memory, can mitigate most interference from other applications. Using the above observations, they periodically allocate requests of each application the highest priority to reduce most of the inter-application

interference, and use the ratio of request service rates as a proxy for slowdown estimation. Different from MISE, ASM further considers the shared cache interference.

However, we observe that these models are inaccurate on GPUs. The reason is as follows. First, because a GPGPU application would use all SMs when it is running alone, these models can only estimate slowdowns on the constant number of SMs rather than all SMs. Second, because they assume that giving memory requests of one CPU application the highest priority can eliminate most of interference, they roughly estimate the inter-application interference on shared resources. But we observe that the interference remains severe even if requests of one GPGPU application are allocated the highest priority, because the number of memory requests on GPUs is always far more than that on CPUs [5]. We make a detailed qualitative and quantitative comparison of them with the accuracy of DASE in Section 6.

Performance Estimation Models on GPUs. To the best of our knowledge, there is no work to provide a dynamical slowdown estimation model for concurrent GPGPU applications, so we describe prior works on performance estimation models for a single GPGPU application. Hong et al. [9] develop an analytical performance model to estimate the execution time of a GPGPU application. In addition, Hong et al. [10] propose an analytical performance and power model to compute the number of SMs which provides the best performance per Watt. However, it is a heavy burden for users because these models need user to make source code analysis, such as computing the total number of coalesced/un-coalesced memory type instructions, the total number of computation/memory instructions, etc. What is worse, data-dependent applications receive different input data, may cannot be analyzed in advance [2]. In addition, these works are designed for the execution of only one application. In this paper, we aim to dynamically estimate slowdowns accurately without the help of users.

IV. The DASE Model

A. Overview

In this section, we propose the Dynamical Application Slowdown Estimation (DASE) model, which is based on the observation that the performance of a memory-intensive application is roughly proportional to its memory request service rate [22], [23]. A memory-intensive application is one that issues a overwhelmingly large number of requests into memory subsystems, which can keep running when its requests are served, so the rate at which its accesses are served has substantial impact on its performance.

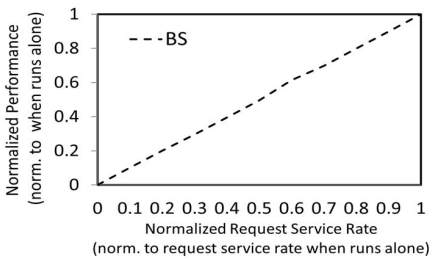


Fig. 3: Performance vs. Request service rate

For ease of understanding, we conduct a memory-intensive application from CUDA SDK [17] to validate our observation. The application is executed multiple times with varying memory intensity, and we measure its performance. As shown in Figure 3, its performance is directly proportional to its memory request served rate, verifying our observation. We find that a wide range of memory-intensive applications also have the feature [22], [23]. In Section 4.2, we further extend this model to support non-memory-intensive applications. Thus, we can estimate the slowdown of an application A_i as the ratio of the memory request service rate when it is running alone (alone-request-service-rate, $ARSR_i$) vs. the memory request service rate when it is running with other applications on the same system (shared-request-service-rate, $SRSR_i$) as Eq.(3).

$$Slowdown_i = \frac{ARSR_i}{SRSR_i} \tag{3}$$

First, we first provide a detailed description about how to compute $SRSR_i$ as Eq.(4). $Time_i^{shared}$ is the number of cycles spent when A_i is running with other applications concurrently. $Request_i^{shared}$ represents the number of requests served while A_i is running with other applications during $Time_i^{shared}$ cycles. It is simple to estimate $Time_i^{shared}$ because we can use a cycle counter to count up it directly. However, the key challenge is to estimate $Request_i^{shared}$, because it may include many redundant memory accesses due to contention cache misses. Contention cache misses means those would be hits when A_i is running alone, which is caused by the inter-application interference in the shared cache.

$$SRSR_i = \frac{Request_i^{shared}}{Time_i^{shared}} \tag{4}$$

Because we observe that applications' behavior is relatively stable over a period of time which is equal or greater than the duration of a thread block, $ARSR_i$ can be computed as Eq.(5) or Eq.(6). $Time_i^{alone}$ is the number of cycles required to serve $Request_i^{shared}$ requests when A_i is running alone. $Request_i^{alone}$ represents the number of requests served when A_i is running alone during $Time_i^{shared}$ cycles. However, it is a challenge to estimate $ARSR_i$ because we cannot directly compute neither $Request_i^{alone}$ nor $Time_i^{alone}$. A straightforward approach of estimating $ARSR_i$ is to avoid other concurrent applications accessing shared resources for a short periods of time and measure it. However, this method cannot eliminate shared cache interference, because cache needs a long time to be warmed up. Moreover, this method will greatly reduce system responsiveness and slow down other applications [23]. In addition, note that an application will use all SMs when it is running alone, it is also impossible to estimate slowdowns accurately by this method.

$$ARSR_i = \frac{Request_i^{shared}}{Time_i^{alone}} \tag{5}$$

$$= \frac{Request_i^{alone}}{Time_i^{shared}} \tag{6}$$

In Section 4.2, we quantify the effect of the inter-application interference on application slowdowns. Moreover, because the above model only supports estimating the slowdown of each

application when using allocated SMs, we extend this model to support estimating the slowdown on all SMs in Section 4.3.

B. Estimation Model On Assigned SMs

Because banks of each DRAM memory system can process outstanding memory requests in parallel, our estimation model regards each DRAM memory system as BLP_i queues, where BLP_i is the average number of banks which are executing memory requests of A_i or will be occupied by memory requests of A_i waiting in the memory queue [14]. Concurrent applications contend with each other for available memory resources (BLP_i queues). Next, we estimate the slowdowns for the Non-Memory-Bandwidth-Bound (NMBB) applications and the Memory-Bandwidth-Bound (MBB) applications from different perspectives.

1) Non-Memory-Bandwidth-Bound Applications: According to Eq.(4) and Eq.(5), we can estimate slowdowns as Eq.(7),

$$Slowdown_i = \frac{(5)}{(4)} = \frac{Time_i^{shared}}{Time_i^{alone}} \tag{7}$$

As described previously, it is simple to count up $Time_i^{shared}$ directly. However, it is difficult to estimate $Time_i^{alone}$ because it is required to get how much memory stall-time an application has experienced when it is running concurrently with other applications. Therefore, we approximate $Time_i^{alone}$ by subtracting $Time_i^{interference}$ from $Time_i^{shared}$ as Eq.(8), and estimate $Time_i^{interference}$ instead. $Time_i^{interference}$ is the number of extra interference cycles caused by other applications in the shared memory system.

$$Time_i^{alone} = Time_i^{shared} - Time_i^{interference} \tag{8}$$

To simplify the following discussion, assuming that one application A_i is running with another application A_j. Our experiments show that the main interference $Time_i^{interference}$ from A_j, which takes up the DRAM execution time originally for serving A_i's memory requests, mainly comes from the following aspects: (a) bank interference, (b) row buffer interference and (c) shared cache interference.

DRAM Bank Interference. Because a request cannot be preempted once be scheduled into the bank, if memory requests of A_j occupy some banks, memory requests of A_i which are issued to the same banks, cannot be serviced until requests of A_j are finished. In addition, memory requests of A_i also cannot be issued to the corresponding banks when the memory controller is busy to issue memory requests of A_j.

In order to simplify hardware implementation cost, the memory controller uses a counter $BLPAccess_i$ to keep track of the average number of banks which are executing requests of A_i currently, and uses (BLP_i - $BLPAccess_i$) to approximate all such bank interference, because the main reason of blocking requests of A_i is that their banks are occupied by A_j or the memory controller is busy to issue requests of A_j.

$$Time_i^{BK} = Time_i^{shared} \times (BLP_i - BLPAccess_i) \tag{9}$$

DRAM Row Buffer Interference. When A_i is running with A_j, its row buffer hit ratio may be reduced, when A_j's requests close a row (activated by A_i), which will be visited by subsequent requests of A_i. If it happens, the corresponding bank must spend more time $(tRP + tRCD)$ [14] for A_i to open a row. To consider such interference, DASE maintains the last access row address for A_i in each bank. If A_i wants to open a row identical to the last access row, such buffer interference is detected. We use a counter $ERBMiss_i$ to record the number of extra row buffer misses.

$$Time_i^{RB} = ERBMiss_i \times (tRP + tRCD) \tag{10}$$

Cache Interference. Like row buffer interference, last-level cache (LLC) hit ratio of A_i may also be decreased. The extra memory accesses induced by contention cache miss also consume available DRAM resource for A_i. In order to detect such misses, we maintain auxiliary tag directory (ATD) [20], which has the same associativity and adopts the same LRU replacement policy, to keep track of the state of the cache when A_i is running alone. When a LLC miss occurs for A_i, if ATD has the same miss entry, we predict that the cache line was replaced by A_j previously. We use a counter $ELLCMiss_i$ to record the number of extra LLC miss.

$$Time_i^{LLC} = ELLCMiss_i \times Time_i^{average} \tag{11}$$

$Time_i^{average}$ represents the average time spent during which requests of A_i are scheduled into one bank and sent to the data bus. We use a counter $Request_i$ to count up the total number of served memory requests for A_i, a counter $Time_i^{request}$ to keep track of the total time spent in banks for all memory requests of A_i.

$$Time_i^{average} = \frac{Time_i^{request}}{Request_i} \tag{12}$$

In addition, we employ set sampling [20] to reduce the overhead of ATD, which approximates the number of misses by maintaining a few sampled sets. $SampleMiss_i$ represents the number of extra LLC miss detected by ATD. $SampleFraction_i$ is the fraction of sample sets vs. all sets.

$$ELLCMiss_i = \frac{SampleMiss_i}{SampleFraction_i} \tag{13}$$

Bank Level Parallelism. As described previously, increasing all interference cycles is not accurate, because multiple banks can execute multiple requests simultaneously [14]. For this reason, we regard each DRAM memory system as BLP_i queues and approximate $Time_i^{interference}$ as

$$Time_i^{interference} = \frac{Time_i^{BK} + Time_i^{RB} + Time_i^{LLC}}{BLP_i} \tag{14}$$

Thread Level Parallelism. Because GPU applications always rely on strong Thread-Level Parallelism (TLP) to hide memory latency, especially non-memory-intensive applications, which spend a large fraction of time in the compute phase, the extra stall-time caused by other applications may be hidden by executing multiple thread concurrently and not impact their performance. Thus, we take into account it to extend our model to support all non-memory-intensive applications.

Since the SMs overlap memory access with computation, exactly determining how much the stall-time has impacted the performance is much more difficult to implement. Let α_i be the fraction of time when the SM pipeline is stalled waiting for memory requests of A_i, so it is not stalled by the interference from other applications during the remaining (1 - α_i) [23]. The SMs stall for memory requests only when its TLP is unable to hide memory latency. Thus, the slowdowns can be estimated approximatively as Eq.(15). In addition, similar to previous work [23], we observe that setting α_i to 1 makes DASE more accurate when α_i is large.

$$Slowdown_i = 1 - \alpha_i + \alpha_i \times \frac{Time_i^{shared}}{Time_i^{alone}} \quad (15)$$

2) Memory-Bandwidth-Bound Applications: A memory-bandwidth-bound application is one that its performance is limited by memory bandwidth, whose α_i cannot be ignored even if it is running without the interference from other applications. In addition, when a MBB application is running with others, the memory request buffer is full of outstanding requests at all times, $Time_i^{BK}$ may be underestimated when some requests of A_i are still waiting in the cache. Thus, the accuracy of estimated slowdowns may be affected when adopting Eq.(15). We can reduce such interference by assigning highest priority for memory accesses of A_i in accessing the network and cache, but this method needs complex control logic and large hardware cost in both network and cache layers. So we estimate the slowdown of the MBB applications from different perspectives. According to Eq.(4) and Eq.(6), we can estimate the slowdown of A_i as Eq.(16).

$$Slowdown_i = \frac{(6)}{(4)} = \frac{Request_i^{alone}}{Request_i^{shared}} \quad (16)$$

First, $Request_i^{shared}$ is computed by Eq.(17). $Request_i$ represents the number of served memory requests from A_i when it is running concurrently with other applications, and we use a counter to count up it directly. However, as explained previously, $Request_i$ may include many extra requests due to contention cache misses, so we compute $Request_i^{shared}$ by subtracting the extra LLC misses from $Request_i$.

$$\text{Re}quest_i^{shared} = Request_i - ELLCMiss_i \quad (17)$$

Second, it is a challenge to estimate $Request_i^{alone}$. Fortunately, we find that the number of served memory requests from a MBB application when it is running alone is close to the number of served memory requests from all concurrent applications. Figure 4 shows examples when one MBB application (SB) is running with other applications, the number of average served memory requests when SB is running alone (420 served requests per 1000 GPU cycles) is close to the average sum of served memory requests of SB and 2_{st} application (439).

The reason is as follows. When the memory-bandwidth-bound application SB is running alone, it nearly transfers data from the DRAM all the time, the memory system is busy to serve its memory requests at all times so that the BLP memory banks is always full of its outstanding requests.

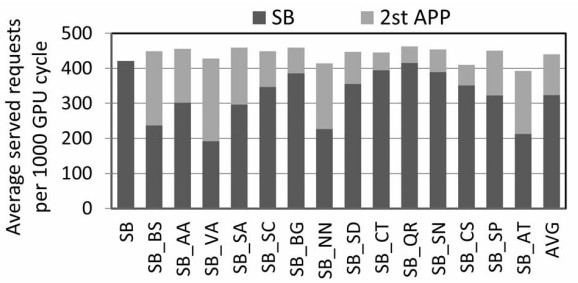


Fig. 4: Average served memory requests of SB and 2st application

However, when multiple applications are running concurrently, according to the DRAM timing constraints, the bank and the memory controller may be busy to serve requests of other applications, unable to execute the corresponding requests of SB. In addition, the data bus is also busy when transferring data of other applications, forbidding SB to transfer data even if its requests are ready. Hence, memory system can serve more memory requests for SB if without requests of other applications. The increased number of requests for SB is quite close to the number of served requests for others. We find other MBB applications also have the feature. So we use the sum of served memory requests from all concurrent applications to approximate $Request_i^{alone}$ as Eq.(18):

$$Request_i^{alone} = \sum_{i=0}^{N-1} Request_i \quad (18)$$

Note that because the number of requests from a MBB application is large, we observe that the decrease of the row buffer hit ratio is negligible and does not affect the accuracy, so the above model does not consider it.

3) MBB or NMBB?: We adopt a simple way to dynamically determine an application as a memory-bandwidth-bound (MBB) application if it satisfies Eq.(19), Eq.(21) and Eq.(22), otherwise as a NMBB application.

First, when a MBB application is running with other applications, the number of served memory requests from all concurrent applications is equal or greater than $Request_{max}$, because MBB applications always transfer data when the DRAM is idle. Where $Request_{max}$ represents the maximum memory requests issued by the DRAM memory.

$$\sum_{i=0}^{N-1} Request_i \geq Request_{max} \quad (19)$$

$Request_{max}$ is computed via dividing $Time_{shared}$ by $Time_{perReq}$ as Eq.(20). $Time_{perReq}$ represents the number of cycles required to serve each request, which is constant depend on the last level cache line size and DRAM burst length. In addition, we find that the memory bandwidth cannot be fully used due to the influence induced by satisfying DRAM timing constraints, such as ACT and PRE operations, etc. Certainly, $Request_{max}$ is heavily upon the characteristics of running applications, and users can adjust it when running different kernels. However, in this paper, we multiple $Request_{max}$ by

an empirical default parameter 0.6. The strategy of dynamically calculating $Request_{max}$ based on kernel characteristics can be further explored, but is beyond the scope of this paper.

$$Request_{\max} = \frac{Time_{shared}}{Time_{perReq}} * 0.6 \quad (20)$$

Second, we find that MBB applications always have a high bandwidth utilization, regardless of which applications their are running concurrently with. $Count_{App}$ represents the number of concurrently executing applications.

$$\frac{Request_i^{shared}}{Request_{\max}} \geq \frac{1}{Count_{App}} \quad (21)$$

Third, a MBB application is one that its performance is limited by memory bandwidth, even if they are running without the interference from other applications. Eq.(22) means that an application is MBB if the SM pipeline is still stalled waiting for requests even if using all available memory bandwidth.

$$\frac{1}{1-\alpha_i} * Request_i^{shared} \geq Request_{\max} \quad (22)$$

C. Estimation Model On All SMs

We previously have described how to estimate slowdowns when A_i is running only on the assigned SMs, but it can use all SMs when it is running alone. For MBB kernels, our experiments show that their performance cannot increase even if assigning more SMs to them. However, for NMBB kernels, their performance may be improved when using all SMs. Application behavior of all thread blocks of the same kernel is very similar, because they execute the same code. So we provide an estimation model on all SMs for NMBB kernels as Eq.(23), where SM_{all} and SM_i represent the number of all SMs and assigned SMs for A_i, $Slowdown_i^{all}$ and $Slowdown_i^{shared}$ represent the slowdown of A_i on all SMs and assigned SMs.

$$Slowdown_i^{all} = Slowdown_i^{shared} \times \frac{SM_{all}}{SM_i} \quad (23)$$

However, we find that the performance improvement of NMBB applications may be limited due to thread-level parallelism ability and memory bandwidth demand.

Thread Level Parallelism Ability. For example, if A_i has only x thread blocks and it has used up all x blocks when it is running on the assigned SMs, its IPC cannot be improved even if increasing its SMs. We use TB_i^{sum} to represent the total number of thread blocks which are not finished, and TB_i^{shared} to represent the number of thread blocks which are running concurrently with other applications on the assigned SMs.

$$Slowdown_i^{all} = \min\{\frac{Slowdown_i^{shared} \times TB_i^{sum}}{TB_i^{shared}}, Slowdown_i^{all}\} \quad (24)$$

Memory Bandwidth Demand. When we allocate more SMs to an application, it will consume much more memory bandwidth, and may be limited to memory bandwidth.

$$Slowdown_i^{all} = \min\{\frac{Request_{max}}{Request_i^{shared}}, Slowdown_i^{all}\} \quad (25)$$

In addition, SM_{all} and SM_i can be obtained at compile time, TB_i^{sum} and TB_i^{shared} can be obtained at runtime.

D. Implementation and Hardware Cost

The DASE model is invoked when a new kernel is scheduled to a set of SMs or thread blocks of a kernel are used up. The slowdown of each application is sampled by averaging it over a period of time which is equal or greater than the duration of a thread block. However, because long-running thread blocks of some kernels may reduce responsiveness, we choose a fixed length of interval 50K cycles, and we observe that it is enough effective to capture application characteristics. Note that the estimation process is not on the critical path.

As shown in Table 1, DASE has many hardware counters to keep track of the related information. At the beginning of each estimation interval, the controller resets all counters. The slowdown of each application is estimated one by one to reduce hardware cost, so DASE incurs a small hardware cost. The major cost of DASE is memory hardware counters. When N = 4, the total hardware cost for them is less than 0.4KB in each memory partition (as fraction of the baseline 64 KB L2 cache in each memory partition: 0.4KB/64KB<0.625%).

TABLE I: The major hardware cost for DASE

Memory hardware counters (Per memory partition)	
$ERBMiss/ELLCMiss$ counters	32bits
Last access row address registers	N_{bank} * 16bits
Sample ATD	8set*8way*32bit
Served memory requests counters	32bits per application
$Time^{Request}$ counters	32bits
$BLP/BLPAccess$ counters	32bits
Other hardware counters	
The fraction of stalled time α	32bits per SM
Interval cycle counters	32bits
$SM_{sum}/SM_{used}/TB_{sum}/TB_{used}$	32bits

V. Evaluation Methodology

Simulation Setup. We use a detailed cycle-level simulator GPGPU-sim v3.2.2 [6] to simulate GPU. The baseline configuration in Table 2 is exactly the same to prior work [4], which is approximate to the NVIDIA GeForce GTX 480 GPU. We distribute commands of different kernels to concurrently running streams (a set of serially executing instructions) to simulate concurrent execution of multiple GPGPU applications [11]. Similar to previous work [11], unless otherwise noted, we adopt a straightforward SM-partition scheme to assign the SMs evenly among applications. For example, the first application gets the first half of all SMs and the second gets the rest when two applications are running concurrently. In Section 6.2, we evaluate the sensitiveness of DASE to varing SM allocation.

TABLE II: The baseline GPU configuration

SM	1400MHz, 16 SMs, Max 48 warps(1536 threads) 48KB shared memory, 32684 registers
Caches	16KB 4-way L1 data cache, 768KB L2 cache 128B cache block size
Interconnect	1 crossbar/direction (16 SMs, 6 Memory Controllers), 1400MHz, Local-RR
Memory	FR-FCFS, 16 DRAM-banks/MC 924 MHz, tRP = 12, tRCD = 12

Workloads. Because our DASE model pays attention on application slowdowns induced by the inter-application interference on shared resources, in order to demonstrate the effectiveness of our model, Table 3 shows that we choose representative kernels from 15 GPGPU applications, as they

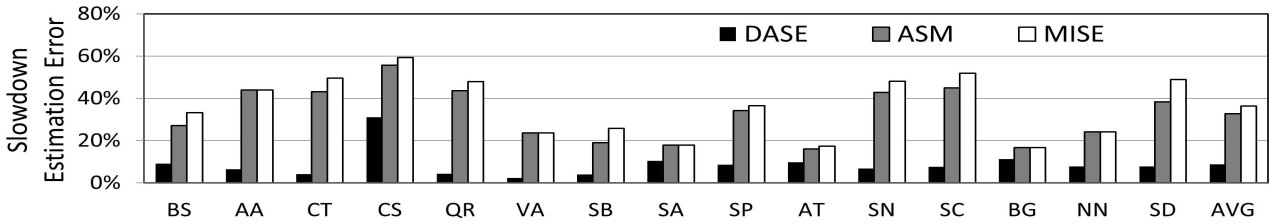


Fig. 5: Slowdown estimation accuracy when adopting different estimation models on two-application workloads

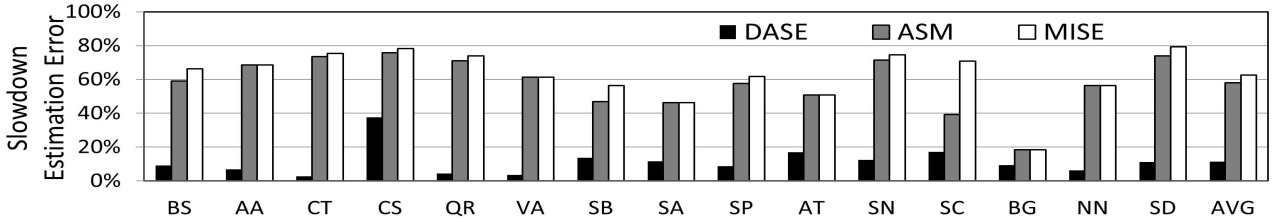


Fig. 6: Slowdown estimation accuracy when adopting different estimation models on four-application workloads

show a variety of memory bandwidth behaviors and cause a lot of interference to other concurrently executing applications on shared resources. These applications are chosen from CUDA SDK [17], Parboil [21], Polybench [8] and Rodinia [7]. We form all possible two-applications, and randomly select 30 four-applications workloads from 15 GPGPU applications.

TABLE III: All evaluated applications, along with their attained DRAM bandwidth utilization when they are executing alone on the entire GPU device

Application	Abbr.	From	BW Utilization
blackScholes	BS	SDK	65%
asyncAPI	AA	SDK	61%
convolutionTexture	CT	SDK	16%
convolutionSeparable	CS	SDK	32%
quasirandom	QR	SDK	14%
vectorAdd	VA	SDK	60%
sobol	SB	SDK	68%
scan	SA	SDK	58%
scalarProd	SP	SDK	55%
alignedTypes	AT	SDK	47%
sortingNetworks	SN	SDK	20%
stencil	SC	Parboil	53%
BICG	BG	PolyBench	21%
Nn	NN	Rodinia	56%
srad	SD	Rodinia	40%

Because a large number of threads are executing concurrently, cycle-accurate GPU simulation is more time consuming than CPU. Thus, it is difficult to simulate the entire execution of each workload in a reasonable amount of time. Instead, we use the methodology in [1], [2] to compare: Each heterogeneous workload is executed for 5M GPU cycles, an application will be restarted if finishes before 5M cycle. According to the number of instructions for each application completed in 5M cycles, the same number of instructions are simulated alone to ensure comparing the same amount of work. Note that 5M cycles are sufficient to get a representative of the steady-state application behaviors [1], [2].

Metrics. We use average estimation error [14], [22] to compare estimation accuracy of DASE, MISE and ASM. In addition, we use Harmonic speedup [14], [22], [23] as our performance measure metric because it provides a balanced measure between fairness and performance.

$$Error = 100\% * \frac{EstimatedSlowdown - ActualSlowdown}{ActualSlowdown} \quad (26)$$

$$H.Speedup = \frac{N}{\sum_{i=0}^{n-1} IPC_i^{alone}/IPC_i^{shared}} \quad (27)$$

VI. Predict Accuracy

As discussed previously, MISE [23] and ASM [22] are the latest works to estimate the slowdowns of concurrent applications on CPUs. To show that our DASE model is more accurate than MISE and ASM, we compare the average estimation error of our DASE model with MISE and ASM. Both our implementations of DASE and ASM keep track of shared cache interference with a sampled auxiliary tag directory (8 cache sets). Figure 5 shows that estimation results of our DASE model deviate from the actual slowdown by only 8.8% while MISE's is 36.3% and ASM's is 32.8% across all two-application workloads. Figure 6 shows that estimation results of our DASE model deviate from the actual slowdown by only 11.4% while MISE's is 62.6% and ASM's is 58% across all 30 four-application workloads.

The reasons are as follows. First, both ASM and MISE are designed for slowdown estimation on CPUs, can only predict application slowdowns on assigned SMs. However, because a GPGPU application would use all SMs when it is running alone, neither ASM nor MISE support slowdown estimation on all SMs. Thus, slowdown estimation accuracy when adopting MISE or ASM across four-application workloads is lower than that across two-application workloads.

Second, because both MISE and ASM assume that giving requests of one CPU application the highest priority on the memory controller can eliminating most of interference, they do not deeply study the details of the interference among GPGPU applications. However, we observe that the interference remains severe even if requests of one GPGPU application are assigned the highest priority, because the number of requests on GPUs is always far more than that on CPUs, and requests

of other applications will be scheduled when there is no request to compete. Thus, the interference cycles are computed unreasonably so that the error of estimated results may be great when adopting MISE and ASM on GPUs. In summary, we conclude that DASE is significantly more accurate than MISE and ASM on GPUs.

A. Distribution of Slowdown Estimation Error

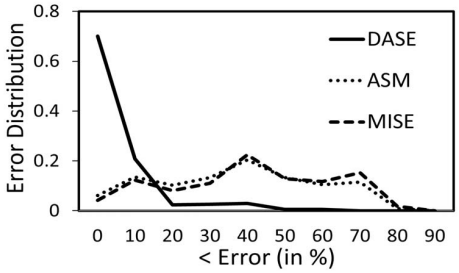
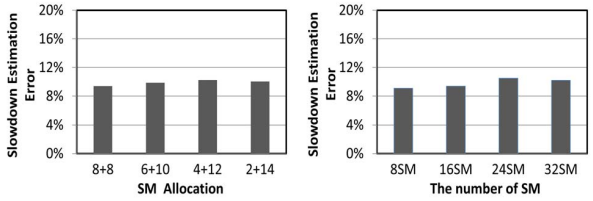


Fig. 7: Error distribution

Figure 7 shows that the distribution of slowdown estimation error for DASE, MISE and ASM, across all workloads. The x-axis shows estimation error ranges and the y-axis shows the distribution of each error range. First, 70.2% of DASE's estimate slowdowns have an error less than 10%, while only 6.2% and 4.2% of ASM and MISE's estimate slowdowns have an error within 10%. In addition, 90.9% of DASE's estimates have an error within 20%, whereas the error distribution of 20% of ASM and MISE is only 19.8% and 16.5% respectively. Therefore, we conclude that our DASE model has significantly lower variance than ASM and MISE.

B. Sensitive to the SM Allocation



(a) Sensitive to the SM allocation (b) Sensitve to the number of SMs

Fig. 8: Slowdown estimation accuracy when adopting (a) varying SM allocation, (b) varying number of SMs

Figure 8(a) shows that sensitivity of estimation accuracy to different SM allocation schemes across randomly selected 30 two-application workloads. We allocate different number of SMs to concurrent applications at launch time, for instance, 6+10 means that the first application is allocated 6 SMs and the second one is assigned 10 SMs. Then, we use the DASE model to estimate application slowdowns. The average estimation error shown in Figure 8(a) shows that DASE is significantly robust to varying SM allocation schemes.

C. Sensitive to the Number of SMs

Next, we evaluate the sensitivity of estimation accuracy to SM count. We allocate the same number of SMs to concurrent applications at launch time and use the DASE model to estimate slowdowns. As shown in Figure 8(b), the error bars show that our DASE's slowdown estimates are significantly accurate across varying number of SMs. This is because we consider the inter-application interference on the shared resources deeply and carefully in Section 4.

VII. Leveraging DASE

In this section, we employ the DASE model to develop a fairness-oriented SM allocation policy DASE-Fair. Certainly, the DASE can also be leveraged to design other slowdown-aware mechanisms to provide QoS guarantees and improve performance, we leave other scenarios as future work.

In the real system, it is unreasonable to try all possible SM allocation schemes before selecting the best one that minimizes unfairness, because the system may execute different combinations of a wide range of applications. Thus, our DASE-Fair policy allocates the same number of SMs to concurrent applications, and selects a best SM partition with the maximum improvement in fairness at runtime, dynamically adjusts the SM allocation if our estimation model proves that the initial SM allocation is unfair.

Before dynamically reassigning SMs among applications, we need to know the slowdown of each application when it is allocated i SMs and other applications are allocated the remaining SMs. Estimating it requires two pieces of information: 1) the slowdown of the application when it is running alone on i SMs, and 2) the interference from other concurrently executing applications on the remaining SMs. However, it is difficult to exactly achieve both points. Therefore, we design a simple estimation scheme.

First, we use the DASE model to achieve all slowdowns of concurrent applications, and compute the reciprocals of their slowdown values as Eq.(28). For example, if the slowdown of one application is 4, its reciprocal is 0.25. We use reciprocal because it linearly reflects the performance of concurrent applications compared to the case when running alone, and its value is between 0 and 1.

$$Reciprocal_i = \frac{1}{Slowdown_i} \tag{28}$$

Second, we try to estimate the slowdown of each application when it is allocated i SMs, using two different linear functions of Reciprocal vs. SM count. For the case when this application is allocated more SMs, we use the linear function between the current estimation slowdown with assigned SMs to the application's slowdown with all SMs as Eq.(29). Note that if one application is allocated all SMs, the reciprocal of its slowdown is 1. $Reciprocal_i^{estimated}$ represents the reciprocal of A_i's slowdown which is estimated by DASE, and $Reciprocal_i^x$ represents the reciprocal of A_i's slowdown on x SMs. SM_{all} and SM_i mean the number of all SMs and assigned SMs for A_i.

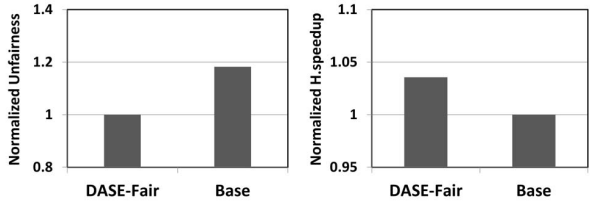
$$Reciprocal_i^x = Reciprocal_i^{estimated} + \frac{x - SM_i}{SM_{all} - SM_i} \times (1 - Reciprocal_i^{estimated}) \tag{29}$$

For example, estimated slowdown of one application is 2 when running on 8 SMs (a total of 16 SMs, other applications occupy the remaining 8 SMs). So $Reciprocal_i^{estimated}$ of its

slowdown = 1/2 = 0.5. When the application is allocated 12 SMs (others occupy 4 SMs), we estimate the reciprocal $Reciprocal_i^{12}$ of its slowdown is 0.5 + ((12 - 8) / (16 - 8)) * (1 - 0.5) = 0.75. For the case when it is allocated less SMs, we use the similar linear function as Eq.(30). Note that if each application is allocated 0 SMs, its slowdown reciprocal is 0.

$$Reciprocal_i^x = Reciprocal_i^{estimated} - \frac{SM_i - x}{SM_i - 0} \times (Reciprocal_i^{estimated} - 0) \quad (30)$$

Using the above equations, we can estimate the slowdowns of all applications no matter how much SMs are allocated. Because the number of SMs is small, according to Eq.(2), we search all possible SM allocation schemes to find a best scheme to minimize unfairness. Then we migrate a set of SMs from one kernel to another kernel by adopting SM Draining [3], [4], [19], which waits for the SMs to finish executing the remaining work allocated from the previous kernel. No additional thread blocks can be allocated to the reallocated SMs until finishing reallocation. Then, new kernels are assigned to the reallocated SMs.



(a) Unfairness (b) H.Speedup

Fig. 9: Unfairness and H.Speedup when employing different SM allocation policies

However, this method is unsuitable for some kernels, which have too less thread blocks or are too short. Thus, besides these unfit kernels, we evaluate unfairness and H.speedup when employing different SM allocation policies across all workloads. Figure 9 only compares DASE-Fair with the default policy which assigns the SMs evenly among application, because previous QoS-aware SM allocation policies [3], [4] are required isolated kernel profiling information to compute the slowdowns of concurrent applications. As we see, our DASE-Fair policy provides significantly better fairness (16.1%) and higher performance (3.7%) on average.

VIII. Conclusions

In this paper, we propose a new slowdown estimation model DASE to detect the unfairness in GPGPUs. Experiments show that our proposed DASE model has a significantly lower estimation error across all two-application workloads (i.e., 8.8%), compared to the state-of-the-art models (e.g., MISEs at 36.3% and ASMs at 32.8%). Based on the DASE model, we further design a fairness-oriented streaming multiprocessor partition policy, which dynamically reassign shared resources to mitigate the fairness and performance degradation. In future work, we plan to design more slowdown-aware scheduling policies to provide better QoS guarantees.

Acknowledgment

We would like to thank the anonymous ICPP reviewers for their comments that helped improve this paper. This work is supported by the National Natural Science Foundation of China (Grant No.61232003, 61433008), the Beijing Municipal Science and Technology Commission of China (Grant No.D151100000815003), the National High Technology Research and Development Program of China (Grant No.2013AA013201).

References

[1] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for gpgpu spatial multitasking. In *HPCA*.
[2] P. Aguilera, J. Lee, A. Farmahini-Farahani, K. Morrow, M. Schulte, and N. S. Kim. Process variation-aware workload partitioning algorithms for gpus supporting spatial-multitasking. In *DATE*.
[3] P. Aguilera, K. Morrow, and N. S. Kim. Qos-aware dynamic resource allocation for spatial-multitasking gpus. In *ASP-DAC*.
[4] P. Aguilera, K. Morrow, and N. S. Kim. Fair share: Allocation of gpu resources for both performance and fairness. In *ICCD*, 2014.
[5] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged memory scheduling: achieving high performance and scalability in heterogeneous systems. *ISCA*, 2012.
[6] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS*, 2009.
[7] 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 *IISWC 2009*, 2009.
[8] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, 2012.
[9] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ISCA*, 2009.
[10] S. Hong and H. Kim. An integrated gpu power and performance model. In *ISCA*, 2010.
[11] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das. Application-aware memory system for fair and efficient execution of concurrent gpgpu applications. In *GPGPU*, 2014.
[12] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das. Anatomy of gpu memory system for multi-application execution. In *MEMSYS*, 2015.
[13] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen. Efficient gpu spatial-temporal multitasking. *TPDS*, 2015.
[14] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*, 2007.
[15] Nvidia. Nvidias next generation cuda compute architecture: Fermi. 2009.
[16] Nvidia. Nvidias next generation cuda compute architecture: Kepler gk110. Technical report, 2012.
[17] C. Nvidia. Nvidia cuda c programming guide. *NVIDIA Corporation*.
[18] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *ASPLOS*, 2013.
[19] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *ASPLOS*, 2015.
[20] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.
[21] 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*, 2012.
[22] 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 *MICRO*, 2015.
[23] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *HPCA*, 2013.
[24] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on gpus. In *ISCA*, 2014.