HSM: A Hybrid Slowdown Model for Multitasking GPUs

Xia Zhao
Ghent University

Magnus Jahre
Norwegian University of Science and Technology

Lieven Eeckhout
Ghent University

Abstract

Graphics Processing Units (GPUs) are increasingly widely used in the cloud to accelerate compute-heavy tasks. However, GPU-compute applications stress the GPU architecture in different ways — leading to suboptimal resource utilization when a single GPU is used to run a single application. One solution is to use the GPU in a multitasking fashion to improve utilization. Unfortunately, multitasking leads to destructive interference between co-running applications which causes fairness issues and Quality-of-Service (QoS) violations.

We propose the Hybrid Slowdown Model (HSM) to dynamically and accurately predict application slowdown due to interference. HSM overcomes the low accuracy of prior white-box models, and training and implementation overheads of pure black-box models, with a hybrid approach. More specifically, the white-box component of HSM builds upon the fundamental insight that effective bandwidth utilization is proportional to DRAM row buffer hit rate, and the black-box component of HSM uses linear regression to relate row buffer hit rate to performance. HSM accurately predicts application slowdown with an average error of 6.8%, a significant improvement over the current state-of-the-art. In addition, we use HSM to guide various resource management schemes in multitasking GPUs: HSM-Fair significantly improves fairness (by 1.59× on average) compared to even partitioning, whereas HSM-QoS improves system throughput (by 18.9% on average) compared to proportional SM partitioning while maintaining the QoS target for the high-priority application in challenging mixed memory/compute-bound multi-program workloads.

Keywords GPU; Multitasking; Slowdown Prediction; Performance Modeling

1 Introduction

GPUs have become the preferred compute platform for a wide variety of compute-heavy tasks including machine learning, scientific simulations, and data analytics. Commonly, users only intermittently need GPU capacity — making it more economical to rent GPU capacity on demand than buying GPUs. Hence, leading cloud service suppliers such as Google and Amazon offer GPU-equipped virtual machine instances [1, 2]. The simplest way to provide cloud-based GPU compute capacity is to give each user exclusive access to physical GPUs. This strategy is wasteful since most GPU-compute workloads do not fully utilize all GPU resources. For instance, a compute-bound application will utilize the Streaming Multiprocessors (SMs) well, but use relatively little memory bandwidth. Conversely, a memory-bound application will have high bandwidth utilization, but use the SMs inefficiently.

This utilization problem can be overcome by enabling GPUs to execute multiple applications concurrently — thereby better utilizing compute and memory resources [3–6]. Unfortunately, this creates another problem: The co-running applications can interfere unpredictably with each other in the shared memory system [7–9]. In a GPU-enabled cloud, the main undesirable effects of interference are that it can result in (1) users being billed for resources they were unable to use, and (2) unpredictable Quality-of-Service (QoS) and Service-Level Agreement (SLA) violations.

Enforcing fairness/QoS requires understanding how interference affects the performance of co-running applications. More specifically, we need to predict the performance reduction (*slowdown*) during multitasking (*shared mode*) compared to an ideal configuration (*private mode*) where the application runs alone with exclusive access to all compute and memory system resources [10]. Using shared mode quantities (e.g., shared mode bandwidth utilization) as proxies for private mode quantities (e.g., private mode bandwidth utilization) is typically inaccurate since interference can change application resource consumption significantly. Thus, slowdown prediction schemes use performance models or heuristics that take shared mode measurements as input to predict private mode performance. While slowdown prediction has been studied extensively for CPUs (e.g., [11–13]), the problem is less explored for multitasking GPUs [14–16]. Since slowdown prediction is performed online, architects need to appropriately trade off the overheads of the scheme (i.e., the number of counters and logic complexity) against the required accuracy to use chip resources as efficiently as possible.

Broadly speaking, slowdown prediction models can be classified as white-box [10, 11, 14] versus black-box [15, 16]. White-box models are derived from fundamental architectural insights which enables them to, in theory, precisely capture key performance-related behavior. This means that they incur limited implementation overhead when used online (i.e., few counters and simple logic). Unfortunately, it is

difficult to formulate accurate white-box models for GPUs because of the high degrees of concurrent threads and memory accesses which leads to non-trivial overlap effects, i.e., it is extremely difficult to tease apart the performance impact from shared resource interference. DASE [14], the state-of-the art white-box GPU slowdown model, estimates interference in all shared resources individually to then predict the overall performance impact. While successful for CPUs [10–13], this approach is both complex and inaccurate for GPUs because of the high degree of overlap effects — we observe an average prediction error of 17.9% and up to 75.3%.

Black-box models circumvent this problem by automatically learning the performance impact of the non-trivial overlap effects. Unfortunately, this ability comes at the cost of non-negligible training and implementation overheads. For instance, Deep Neural Networks (DNNs) are typical candidates for complex machine learning problems, but they can take a long time to train, usually have hundreds of input parameters, and consist of tens of layers; resulting in the weights used for online inference requiring megabytes of storage even after compression [17]. Themis [16], the state-of-the-art black-box model for GPU slowdown prediction, uses a four-layer neural network and limited number of input parameters to reduce overheads, but still requires kilobytes of storage and performs hundreds of floating-point multiplications to issue a single prediction. For our diverse set of workloads, we find that Themis' neural network is too simple to provide robust slowdown predictions (average error of 33.8%, and up to 99.8%).

Our key insight is that we can design accurate performance models with low complexity by combining the strengths of white-box and black-box approaches. More specifically, we first use white-box modeling to determine the key performance-related behavior resulting from interference in multitasking GPUs. Second, we use black-box modeling to learn the performance impact of the non-trivial overlap effects. In this way, we synergistically combine white-box and black-box modeling in a *hybrid* approach. More specifically, the white-box approach captures key performance-related behavior — to reduce training and implementation overhead — while the black-box model accounts for non-trivial overlap effects — to achieve high accuracy.

The result is our Hybrid Slowdown Model (HSM) which builds upon five white-box insights regarding interference in GPUs. (1) The streaming execution model of GPUs results in kernels being either memory or compute-bound, and (2) the performance of compute-bound kernels scale linearly with the number of allocated SMs. (3) The performance of memory-bound kernels is strongly correlated with memory bandwidth, and (4) DRAM Row Buffer Hit Rate (RBH) determines a memory-bound kernel's DRAM bandwidth utilization potential. The reasons are that RBH dictates the fraction of the theoretical DRAM bandwidth that remains idle due to row management operations and that modern address mapping policies achieve nearly perfect bank and channel balance [18]. (5) Memory-bound kernels have similar RBH in the shared and private mode. The reason is that GPUs have hundreds or thousands of requests in flight — meaning that they are typically able to keep their row hit requests in the memory controller queues even during multitasking. Further, memory controllers protect RBH by prioritizing row hits to maximize bandwidth [19].

The black-box component of HSM addresses the problem of relating RBH to performance for memory-bound applications, which is non-trivial due to the highly parallel execution model of GPUs. We observe that performance is proportional to memory bandwidth utilization, and we use linear regression to learn the precise relationship for each GPU architecture. The constants of the linear regression model are insensitive to the exact kernels used for training as long as low-RBH and high-RBH kernels are included. Overall, HSM has an average slowdown prediction error of 6.8% (30.3% max error); a significant improvement compared to the state-of-the-art.

To demonstrate the utility of HSM, we use its accurate slowdown predictions to guide SM-allocation policies designed to optimize for system-level performance metrics such as QoS and fairness. More specifically, we find that HSM-based SM allocation is an effective mechanism to manage both the SM resources *and* shared memory bandwidth in a coordinated way to improve system-level performance including fairness (HSM-Fair) as well as system throughput (STP) while maintaining QoS for the high-priority application (HSM-QoS). Our experiments show that HSM-based SM allocation leads to a significant improvement over both even partitioning and DASE-based SM allocation. HSM-Fair improves fairness by 1.59×, 1.36×, and 1.29× compared to even partitioning, Themis and DASE, respectively. When HSM-QoS is used to optimize STP while respecting the QoS target of a high-priority application, it improves STP by 18.9%, 13.0%, and 15.2% in mixed compute/memory-bound workloads compared to proportional SM partitioning, Themis-based SM allocation, and DASE-based SM allocation, respectively.

In summary, we make the following major contributions:

- We introduce hybrid GPU slowdown modeling which combines the understanding of key performance-related behavior provided by white-box models — to reduce training and implementation overhead — with the ability of black-box models to account for non-trivial overlap effects — to achieve high accuracy.
- We propose HSM, the first hybrid slowdown model for GPUs, which accurately predicts application slowdown based on Row Buffer Hit Rate (RBH) measured during multitasking. HSM predicts slowdown with an average error of 6.8% whereas white-box DASE [14]

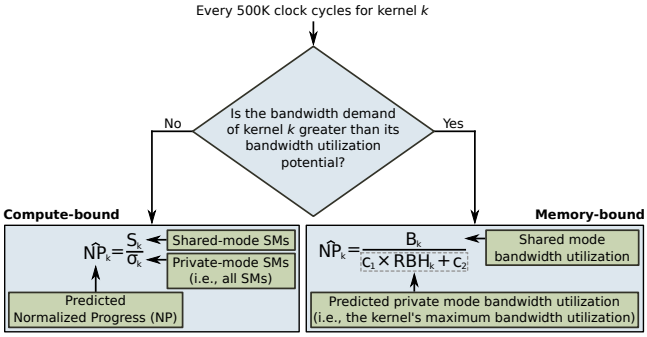
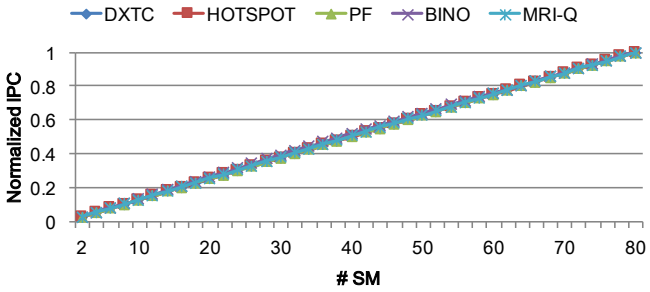


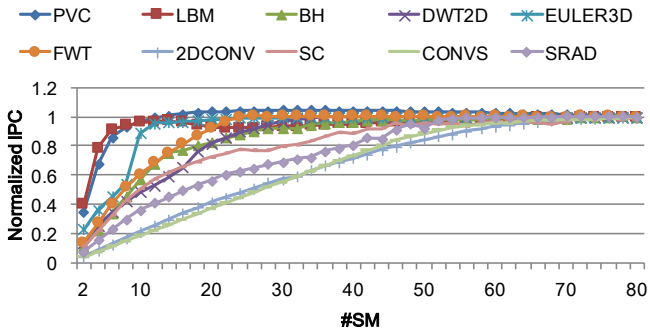
Figure 1. Flow-chart outlining the main operation of HSM.

and black-box Themis [16] yield average prediction errors of 17.9% and 33.8%, respectively.

- We use HSM to allocate SMs to applications to achieve fairness- and QoS-aware multitasking GPUs. HSM-Fair improves fairness by 1.59×, 1.36× and 1.29× compared to even partitioning, Themis-based SM allocation, and DASE-based SM allocation, respectively. HSM-QoS improves system throughput by 18.9%, 13.0%, and 15.2% on average compared to proportional SM partitioning, Themis, and DASE, while respecting the QoS target for the high-priority application in challenging mixed compute/memory-bound multi-program workloads. Proportional SM partitioning, Themis, and DASE do not respect the QoS target.



(a) Compute-bound kernels



(b) Memory-bound kernels

Figure 2. Slowdown of GPU-compute kernels as a function of the number of allocated SMs.

2 The Hybrid Slowdown Model (HSM)

HSM predicts application slowdown due to interference, and slowdown is captured by the performance metric *Normalized Progress (NP)*:

$$NP_k = IPC_k^{Shared} / IPC_k^{Private} \quad (1)$$

Equation 1 states that the normalized progress of a kernel k (NP_k) is the ratio of k's performance in shared mode (IPC_k^{Shared}) — where it is allocated a subset of the SMs and competes with other applications for shared memory system resources — to its performance in the private mode ($IPC_k^{Private}$). The private mode is an off-line configuration where the application executes in isolation with exclusive access to all SMs and memory system resources. Shared-mode performance can be measured straightforwardly with hardware counters. Private-mode performance is much more challenging: It requires predicting isolated performance based on shared-mode measurements gathered dynamically while applications are interfering with each other.

2.1 HSM Overview

Figure 1 outlines the main operation of HSM's NP prediction procedure. The main components of HSM are the classifier and the NP predictors. The classifier determines whether an application is memory or compute-bound. For compute-bound applications, HSM leverages the white-box insight that performance is proportional to SM allocation to predict NP (the left-hand path of Figure 1). If the application is classified as memory-bound, HSM predicts the NP of the application using a linear regression model (the right-hand path of Figure 1). HSM is extremely lean and only tracks the number of executed instructions, allocated SMs, bandwidth utilization, and row buffer hits and accesses for each co-runner in the shared mode. In the following sections, we describe HSM's classifier and NP predictors as well as the five white-box insights that underpin its design.

2.2 HSM Classifier

Insight #1: The streaming execution model of GPUs results in kernels being either memory-bound or compute-bound.

The GPU compute model is highly parallel and streaming. This causes the GPU to behave similar to a decoupled architecture [20], i.e., the memory accesses fill the L1 cache, shared memory, or constant cache available within the SM mostly in parallel to the SM executing compute instructions. If the arithmetic intensity and locality of the kernel is sufficiently high, the SM's memory resources will be able to hide memory latencies and the kernel will be compute-bound. Conversely,

if the SM-local memory resources are insufficient, the memory access component becomes the performance bottleneck and the kernel is memory-bound.

Figure 2 sheds more light on this issue by plotting performance as a function of SMs relative to the 80 SM configuration (see Section 5 for details regarding our experimental setup). We conduct the experiment in private mode with IPC as the performance metric. Figure 2a illustrates that the normalized performance of a compute-bound kernel scales linearly with the number of allocated SMs, while Figure 2b shows that the performance of memory-bound kernels initially increases before saturating[1]. The gradient is determined by the bottleneck resource. If the kernel misses frequently in the Last-Level Cache (LLC) and has low RBH, few SMs are enough to saturate the memory system (high gradient). Conversely, a kernel with less frequent LLC misses (or better LLC hit rate) will need many SMs to saturate DRAM bandwidth (low gradient).

Classifier: To classify a kernel, we compute the application's memory bandwidth demand and its potential to utilize the theoretical DRAM bandwidth and compare these. The lower value will be the bottleneck resource. Overall, we are able to correctly classify all kernels within our applications. Equation 2 calculates the maximum memory bandwidth demand B_k^d for kernel k:

$$B_k^d = (I_{\text{Max}}/I_k) \times \text{MemAccesses} \times \text{ReqSize} \times (f/E). \quad (2)$$

We first compute the ratio of the maximum number of instructions I_{Max} that a kernel can execute during E clock cycles and divide this by the number of instructions it actually executed I_k. Then, we compute the total amount of data fetched within the window by multiplying the instruction ratio with the number of row buffer accesses and the request size. Finally, we multiply by the ratio of the clock frequency f and the epoch size E to compute the bandwidth demand in bytes per second. I_{Max}, ReqSize, f, and E are architectural inputs.

Equation 3 represents the application's potential for utilizing the theoretical DRAM bandwidth:

$$B_k^p = B_{\text{Max}} \times (c_1 \times \text{RBH}_k + c_2). \quad (3)$$

We determine the bandwidth utilization potential of a kernel k by measuring RBH_k and applying HSM's linear regression model (see Section 2.3.2). Then, we multiply this utilization with the theoretical memory bandwidth (i.e., 900 GB/s in our setup) to compute the effective bandwidth supply B_k^p. The kernel is memory-bound if its bandwidth demand exceeds the effective bandwidth supply (i.e., $B_k^d > B_d^p$).

2.3 Predicting Normalized Progress (NP)

We now describe how HSM predicts NP for compute and memory-bound applications. When formulating HSM, we adopt the notation proposed in prior work [10]. The objective of the HSM model is to predict the normalized progress of a kernel k (i.e., $\hat{\text{NP}}_k$). We use a hat to show that a value is a prediction based on values obtained in shared mode. Further, we use Latin letters for shared-mode quantities and the corresponding Greek letter to describe the corresponding private-mode quantity. A prediction $\hat{\alpha}$ may differ from the actual value α, and we use the Absolute Relative Error (ARE) to quantify this difference (i.e., ARE $=\mid \hat{\alpha} - \alpha \mid / \alpha$).

2.3.1 Compute-Bound Kernels

Compute-bound kernels have sufficiently high arithmetic intensity and locality for the SM's L1 cache, shared memory, or constant cache to hide the latency of accessing the LLC or DRAM:

Insight #2: For compute-bound kernels, performance improves linearly with the number of allocated SMs.

Predictor: To predict NP for compute-bound kernels, we simply model $\hat{\text{NP}}_k$ as the ratio of the shared-mode SMs S_k over the private-mode SMs σ_k:

$$\hat{\text{NP}}_k = \frac{S_k}{\sigma_k}. \quad (4)$$

By definition, an application uses all available SMs in the private mode (i.e., σ_k is an architecture-specific constant). Some kernels might have too few TBs to use all SMs, but none of the kernels used in our evaluation have this characteristic. In this case, our model would need to use the number of TBs in the application instead of the number of SMs and abstain from allocating more SMs than TBs in shared mode.

2.3.2 Memory-Bound Kernels

The challenging part for NP prediction is to appropriately account for the performance impact of shared memory system interference [10]. To overcome this challenge while minimizing overheads, HSM's NP-predictor for memory-bound kernels combines three white-box insights with an architecture-specific linear regression model.

Insight #3: The performance of memory-bound kernels is correlated with memory bandwidth, and eventually saturates when increasing the number of allocated SMs.

This insight follows from Figure 2b. Here, performance first increases with more SMs before it saturates. The reason is that GPU memory systems are highly parallel, and the effective bandwidth increases when more requests — and in particular row buffer hits — are available within each memory channel. Thus, increasing the number of SMs increases Memory-Level Parallelism (MLP) and results in better memory bus utilization[2]. Eventually, all of the potential row buffer

[1] We define that a kernel is memory-bound on a particular GPU if there exists an SM allocation where performance saturates. Thus, a memory-bound kernel cannot be made compute-bound by increasing its SM allocation.

[2] PVC has a slight (4.5%) performance degradation with higher SM counts. This is due to self-interference in the LLC. Variation in LLC utilization also causes knees in the performance curves of other benchmarks (e.g., 2DCONV), but it does not majorly affect the overall performance trend.

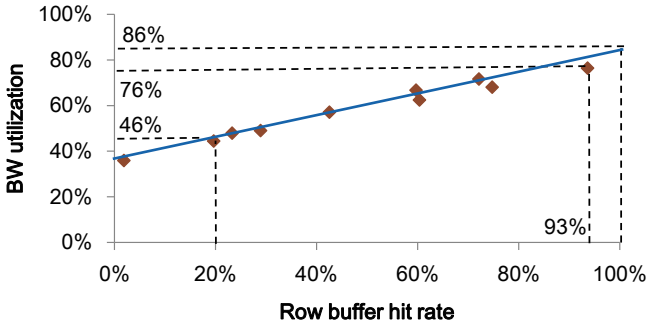


Figure 3. HSM uses linear regression to learn the relationship between RBH and bandwidth utilization.

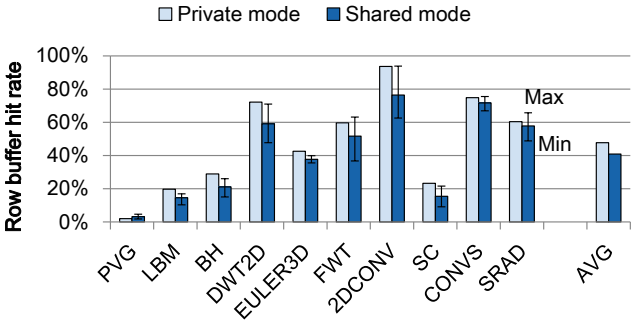


Figure 4. RBH is similar in the shared and private modes.

hits are available to the memory controller, maximizing the effective memory bandwidth. Increasing the SM allocation beyond this point is useless; it simply increases the number of memory requests without improving bandwidth utilization (and hence performance)[3].

The above analysis indicates that performance is maximized for memory-bound kernels when they are able to saturate DRAM bandwidth. Recall that the kernel is allocated all SMs and has exclusive access to memory system resources in private mode. In other words, a memory-bound kernel will saturate DRAM bandwidth in private mode. Thus, we need to identify an architectural characteristic that is measurable in shared mode and has a clear relationship with saturation performance.

Insight #4: RBH determines effective DRAM bandwidth utilization and the relation is approximately linear.

Figure 3 supports Insight #4 by plotting private-mode RBH against bandwidth utilization for our 10 memory-intensive benchmarks as well as the relation we identify with linear regression. RBH is strongly correlated with effective bandwidth utilization because: (1) It determines the amount of DRAM bandwidth that will remain idle due to row management operations within banks (see Section 3 for a detailed architectural explanation); and (2) Load imbalance is exceedingly rare because modern address mapping policies evenly distribute requests across banks and channels [18]. The effective bandwidth range in Figure 3 reaches from around 320 GB/s to 690 GB/s — a dramatic difference for memory-bound kernels. The information required to compute RBH is available within the memory controllers since they identify and prioritize requests that hit in the row buffers to maximize utilization (e.g., the First-Ready, First-Come-First-Served (FR-FCFS) policy [19]).

We have now determined that RBH is a key parameter for determining a kernel's effective bandwidth utilization in private mode. However, NP prediction is performed using shared-mode counters. Thus, we need establish the relationship between shared-mode and private-mode RBH.

Insight #5: RBH is sufficiently similar in the shared and private mode to enable accurate NP prediction.

Figure 4 explains Insight #5 by showing RBH for our memory-intensive benchmarks in private mode compared to the average, maximum, and minimum RBH when co-run with all other applications (i.e., both memory and compute-bound). Overall, RBH is similar, and the reason is that the highly parallel execution model of GPUs means that applications are generally able to keep their row-hit requests in the queues of the memory controller (see Section 3). By favoring row-hit requests, the memory controllers therefore protect the RBH of the co-running kernels — enabling HSM to use shared-mode RBH as a proxy for private-mode RBH.

Predictor: HSM's NP-predictor for memory-bound kernels captures Insight #3, #4, and #5 mathematically:

$$\hat{\mathrm{NP}}_k = \frac{B_k}{\hat{\beta}_k} = \frac{B_k}{c_1 \times \mathrm{RBH}_k + c_2} \tag{5}$$

Equation 5 states that the predicted normalized progress $\hat{\mathrm{NP}}_k$ can be expressed as the ratio of kernel k's shared-mode bandwidth utilization B_k and predicted private-mode bandwidth utilization $\hat{\beta}_k$. HSM uses black-box linear regression to learn the relationship between private-mode bandwidth utilization $\hat{\beta}_k$ and shared-mode RBH (i.e., $c_1 \times \mathrm{RBH}_k + c_2$).

The architectural meaning of the learned constants c_1 and c_2 is that c_1 expresses the expected increase in bandwidth utilization for a corresponding increase in RBH, while c_2 is the expected bandwidth utilization when all memory requests are row buffer misses. We train the black-box model once for each architecture by running a number of benchmarks in private mode, recording their RBH and bandwidth utilization, and then use least-mean square linear regression to determine c_1 and c_2. We find that the values of the constants is insensitive to the benchmarks used for training (see Section 6.5).

[3]We verified that memory bandwidth utilization as a function of the number of SMs is similar to Figure 2b.

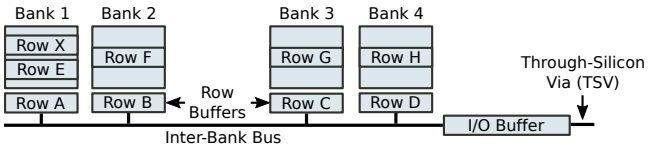


Figure 5. Internal organization of an HBM channel with the initial state for the examples in Figures 6 and 7.

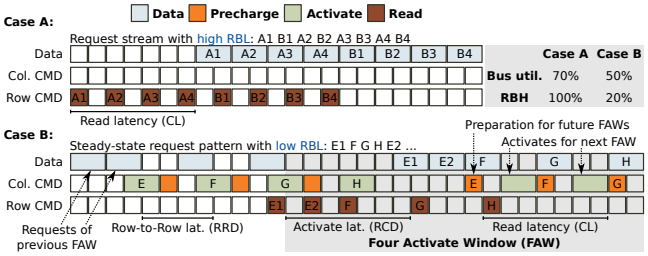


Figure 6. Example explaining why RBH determines effective bandwidth utilization (Insight #4).

3 Explaining HSM's Memory System Insights

In this section, we explain the architectural behavior that leads to the performance trends captured by Insights #4 and #5 in the context of a modern GPU with 3D-stacked High Bandwidth Memory (HBM) [21, 22]. In HBM, DRAM chips are placed on top of each other and memory accesses go through Through Silicon Vias (TSVs). This organization enables more channels and higher bandwidth per channel than non-stacked organizations. To improve command bandwidth, HBM memories have one bus for column commands (e.g., activates and precharges) and one for row commands (e.g., reads and writes). However, the internal organization of the DRAM chips does not change: Each channel still consists of rows, columns, and banks, and it is very efficient to access rows that are already in the row buffers. Therefore, the insights in this section are also relevant for GDDR memory interfaces, and HSM works equally well for GDDR5 (see Section 6.5).

Explaining Insight #4: Figure 6 explains why RBH determines bandwidth utilization potential by considering two cases where different memory requests are queued in the memory controller (see Figure 5 for the initial state). In Case A, all requests access open rows (i.e., RBH is 100%). Thus, the memory controller can directly issue read commands (using the row command bus) to fully utilize memory bandwidth once the start-up latency has been incurred. Bandwidth utilization equals 70% which is lower than the expected value of 86% in Figure 3 because of the start-up latency. A realistic GPU application will have sufficient in-flight memory requests to sustain the steady-state memory bus utilization. However, switching the memory bus from read to write mode

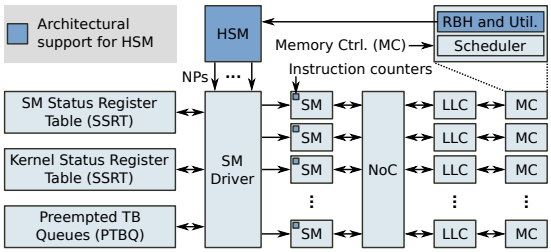


Figure 8. A block diagram showing the architectural support for HSM-based SM allocation policies.

incurs a small latency which explains why the maximum observed utilization in Figure 3 is 76%.

In Case B, we look at the steady-state bandwidth utilization of a kernel with an RBH of 20% (i.e, one in five requests is a row buffer hit). GPU memory controllers keep rows open once activated to improve row buffer hit rate and only precharges (closes) a row when a request needs another row in that bank. The application in Case B has an unfavorable access pattern where the requests cycle through rows in different banks. In this case, the memory controller uses the column command bus (i.e., Col. CMD) intensively to activate and precharge the banks' row buffers — making HBM's capability to quickly activate rows the performance bottleneck. To limit the current draw of the HBM module, the bank activate rate is limited [23], and only four banks can be activated within a Four Activate Window (FAW) (20 bus cycles in our model). Figure 6 shows that the memory controller issues activate commands at the maximum rate which results in a data bus utilization of 50% within the FAW. A bus utilization of 50% fits well with the results in Figure 3 where HSM predicts an utilization of 46% when RBH is 20%.

Explaining Insight #5: Figure 7 explains why shared and private-mode RBH are similar. We consider two applications (BM1 and BM2); see Figure 5 for the initial channel state. Both BM1 and BM2 have an RBH of 75% in private mode and retain the exact same RBH in shared mode. The reason is that the memory controller prioritizes row buffer hit requests over row misses which means that BM1's access to $B2$ and BM2's access to $C2$ remain hits in shared mode even if they are interleaved behind other row miss requests in the memory controller queue. BM1 and BM2 experience a row conflict in Bank 1 since BM1 needs row E and BM2 row X. Since the access for row E is the oldest, it is issued first. This does not affect RBH since the controller issues both requests for row E before closing the row and activating row X. However, the row conflict significantly delays BM2's accesses to row X and affects shared-mode memory bus utilization. This illustrates that interference can significantly affect performance, and indirectly motivates for HSM-based multitasking schemes.

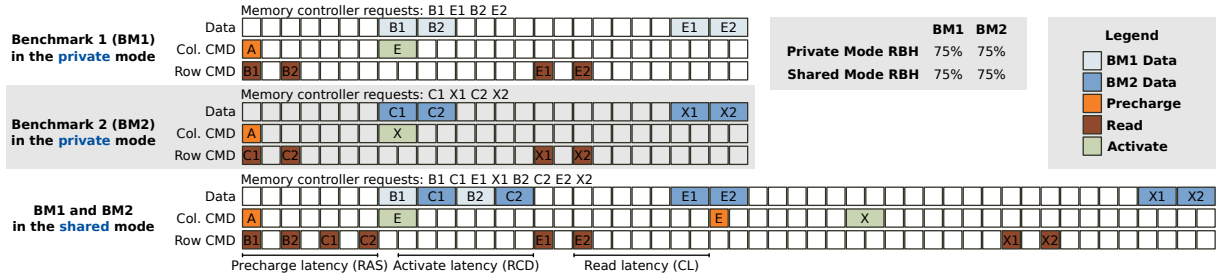


Figure 7. Example explaining why RBH is similar in shared and private mode (Insight #5).

4 Fairness/QoS-Aware Multitasking

We now leverage HSM for fairness- and QoS-aware multitasking. HSM enables this because NP is a key component of widely-used system performance metrics including System Throughput (STP), Average Normalized Turn-around Time (ANTT), and fairness. More specifically, STP is the sum of NPs, while ANTT is the harmonic mean of NPs [24]. Fairness is the ratio of the minimum and maximum NPs [7]. QoS targets can be defined as an NP lower-bound for a high-priority application.

We assume a GPU that supports spatial multitasking with a similar architecture to [5, 6] (see Figure 8). Supporting multitasking requires adding two tables — the SM Status Register Table (SSRT) and the Kernel Status Register Table (KSRT) — and queues for storing the handlers of preempted TBs for each kernel. The SM driver schedules TBs onto SMs and uses the SSRT and KSRT to keep track of kernel and TB execution. We implement HSM-driven policies within the SM driver to optimize for fairness or QoS.

4.1 Architectural Support for HSM

HSM measures RBH by adding a memory request counter and a row buffer hit counter for each co-runner in all memory controllers (see Figure 8). We find that 16-bit counters are sufficient to capture the maximum number of memory requests issued by a kernel within an epoch. Thus, the storage overhead of the row counters amounts to $2 \times 16 \times k \times c$ where k is the maximum number of co-running kernels and c is the number of memory channels. Similarly, shared-mode bandwidth utilization can be captured with a 19-bit counter for each kernel and co-runner. For our 32-channel HBM-based configuration, we support up to 4 co-running kernels which leads to a storage overhead of 0.8 KB. To count the executed instructions, we reuse the performance counter infrastructure already available within the SMs. A single counter per SM is sufficient since only the TBs of a single kernel can run on a particular SM within one epoch.

HSM assumes that the number of shared-mode LLC misses sufficiently accurately predicts private-mode LLC misses. We also investigated if accuracy can be improved by using Auxiliary Tag Directories (ATDs) [25]. HSM with ATDs accurately predicts private-mode miss rates (average error 1.8%), but only improves performance prediction accuracy minorly compared to HSM without ATDs (from 6.8% to 6.4% on average). Since even our aggressively sampled ATDs require roughly 1 KB of storage, we conclude that the accuracy benefit does not outweigh the cost.

4.2 Implementing HSM-based SM Allocation

All our SM allocation policies start by distributing the SMs evenly across kernels. At the end of an epoch (e.g., 500K clock cycles), HSM retrieves the executed instructions, bandwidth utilization, and the number of row hits and accesses for each co-runner, and uses these values to predict the current NP of all co-running kernels (see Figure 1). The HSM-based allocation policies use these NP predictions to (possibly) repartition the SMs into a configuration that is more favorable for the system performance objective. This epoch-based allocation strategy works because TBs within a kernel typically exhibit similar behavior, i.e., the last epoch is generally representative of the next epoch. However, if a kernel completes its execution, we end the current epoch and start a new one with an even SM allocation because behavior can be very different across kernels. To reduce the preemption overhead, we adaptively choose between a draining versus context switching policy [5, 26]. If, during the epoch, the number of TBs finished on one SM is larger than the number of TBs that can be concurrently executed on one SM, we follow a draining policy; if not, we adopt the context switching policy.

Prediction Latency: Predicting NP involves some arithmetic operations, and HSM has a fixed-function hardware unit for this purpose (see Figure 8). It contains a single ALU to perform additions and comparisons in 1 cycle, multiplications in 3 cycles, and divisions in 25 cycles. For memory-bound kernels (worst case), we first need to perform 32 additions on the hit and access counters (64 in total) before dividing the values to obtain RBH. Then, we perform an add and a multiply to predict private-mode bandwidth utilization. Computing the bandwidth demand and utilization potential requires four multiplications and two divisions, and classifying the kernel requires a compare operation. Finally, predicting NP requires a single division (since utilization has already been computed). In total, this procedure takes 181 cycles for a

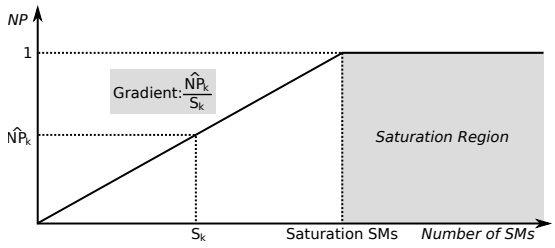


Figure 9. Normalized progress as a linear function of the number of SMs allocated to a kernel.

single kernel. The performance impact of prediction latency is negligible, and we account for it in our evaluation.

Predicting the NP-impact of SM Allocations: The precise SM allocation procedure depends on the system performance target. However, all allocation schemes need to predict how changing the SM allocation will affect the NP of each kernel. Our key insight is that performance is linear in SM count for compute-bound applications and linear in effective memory bandwidth for memory-bound applications. Since allocating more SMs to a memory-bound kernel increases its MLP, it will also have a proportional effect on the bandwidth share of the kernel until performance saturates (see Figure 2b). Thus, NP improves proportionally to the number of SMs allocated to a kernel outside of the saturation region (see Figure 9).

Although NP is linear with SM allocation outside of the saturation region, the gradient is highly application dependent. To solve this problem, we leverage that we know the predicted $\hat{NP}_k$ of kernel k and the number of SMs allocated to k in the last epoch (i.e., S_k). We also know that if we allocate zero SMs to k, the kernel will not be able to execute any TBs and its NP will be zero. In other words, the linear function must pass through the origin. Thus, the gradient $\hat{g}_k$ is simply $\hat{NP}_k/S_k$. Architecturally, $\hat{g}_k$ expresses the benefit in terms of improvement in NP of allocating an additional SM to k.

4.3 HSM-based SM Allocation Policies

HSM-driven SM allocation directly manages compute resources — by allocating SMs to applications — and indirectly manages NoC and memory bandwidth — since the effective bandwidth share of each application is roughly proportional to its memory access rate outside of the saturation region. The key insight is that providing more SMs to an application causes it to issue a larger number of concurrent memory requests and proportionally increase the application's memory bandwidth share. The reason is that requests are serviced in order in the NoC and mostly in order in the memory controllers, resulting in queue occupancy being proportional to offered bandwidth.

HSM-Fair: The objective of HSM-Fair is to improve fairness by ensuring that the NPs of co-running kernels are approximately equal. To accomplish this, we use HSM to predict the current fairness. If fairness is below a user-specified threshold (e.g., 0.9), the HSM unit instructs the SM driver to repartition the SMs to improve it (see Figure 8). First, HSM-Fair finds the kernel with the maximum NP and the kernel with the minimum NP. Then, it uses the linear NP model (see Figure 9) to compute how many SMs should be taken from the high-NP kernel and given to the low-NP kernel to result in similar NP for both kernels. If the current fairness value is higher than the threshold, we keep the current SM allocation. The reason is that we do not want to incur the preemption overhead unless fairness needs to improve.

Figure 2b shows that many memory-bound kernels exhibit a pattern of diminishing returns as the number of SMs approach the saturation region. This means our linear model will predict NP inaccurately if the current number of SMs allocated to the kernel is close to the saturation region. HSM is robust to this situation since it will decide to decrease the number of SMs allocated to the memory-bound kernel if this is favorable in terms of fairness. In the next epoch, the SM allocation will be further away from the saturation region, and the linear model will become more accurate. Thus, HSM-Fair quickly converges to a favorable SM allocation.

HSM-QoS: This policy maintains the NP of the kernels of a high-priority application at a certain level (e.g., 0.8), and then uses the remaining resources to maximize STP. Implementing a QoS policy on top of HSM is straightforward. First, we check if the high-priority kernel meets its NP target. If not, we use the NP model (see Figure 9) to predict the number of SMs necessary to reach the target and take these SMs from the low-priority kernels. If the current NP of the high-priority application is above a user-defined threshold (e.g., 0.9), we use the model to predict the number of SMs it needs to marginally outperform its performance target. Then, we give the freed SMs to the low-priority kernels such that the sum of their NP values, and thus system throughput, is maximized. Note that HSM-QoS does not guarantee a that particular QoS level will be met, as the model (even if it is accurate) has some residual error. However, if a specific QoS level (e.g., 0.8) is required, a target can be set that is slightly above the requirement (e.g., 0.9) to account for model inaccuracies.

5 Methodology

Simulated System: We modified GPGPU-sim v3.2.2 [34] by adding support for spatial multitasking of co-executing programs on different SMs. GPGPU-sim is further extended with Ramulator [27], a detailed memory simulator, to model an HBM-based memory system. We model a GPU with 80 SMs that are connected through a crossbar to 32 memory channels (8 channels per stack) and two 96 KB LLC slices per channel. We use the state-of-the-art PAE randomized address mapping scheme to uniformly distribute memory

Table 1. Simulated GPU architecture.

Baseline HBM-based Configuration	
No. SMs	80 SMs
SM resources	1.4 GHz, 32 SIMT width, 96 KB shared memory Max. 2048 threads (64 warps/SM, 32 threads/warp)
Scheduler	2 warp schedulers per SM, GTO policy
L1 data cache	48 KB per SM (6-way, 64 sets), 128 B block, 128 MSHR entries
LLC	6 MB in total (64 slices, 16-way, 48 sets), 120 clock cycles latency
NoC	80 × 64 crossbar, 32-byte channel width
Memory stack configuration	440 MHz, 4 memory stacks, 8 channels /stack, open page, FR-FCFS, 64 entries/queue, 16 banks/chan., 900 GB/s
HBM Timing [21, 27]	tRC=24, tRCD=7, tRP=7, tCL=7, tWL=2, tRAS=17,tRRDl=5, tRRDs=4, tFAW=20 tRTP=7, tCCDl=1, tCCDs=1, tWTRl=4, tWTRs=2
GDDR5-based Configuration	
No. SMs	40 SMs
LLC	2.75 MB in total (22 slices, 16-way, 64 sets), 120 clock cycles latency
NoC	40 × 22 crossbar, 32-byte channel width
DRAM Timing	Hynix GDDR5 [28]
DRAM configuration	2750 MHz, 11 Memory Controllers (MC), 16 banks/MC, FR-FCFS [19], open page mode 484 GB/s, 12-12-12 (CL-tRCD-tRP) timing

Table 2. GPU-compute benchmarks considered in this study.

Benchmark	Abbr.	MPKI	#Knls	#Insns
Page View Count [29]	PVC	4.79	1	1.35 B
Lattice-Boltzmann Method [30]	LBM	6.09	3	1.24 B
BlackScholes [31]	BH	1.54	14	5.41 B
DWT2D [32]	DWT2D	2.72	1	3.47 B
EULER3D [32]	EULER3D	4.39	7	2.24 B
FastWalshTransform [31]	FWT	2.23	4	3.63 B
2D-convolution [33]	2DCONV	1.21	1	11.03 B
Streamcluster [32]	SC	3.42	2	3.17 B
Convolution Separable [31]	CONVS	1.14	4	7.54 B
Srad_v2 [32]	SRAD	1.09	1	5.27 B
DXTC [31]	DXTC	0.0004	2	18.68 B
HOTSPOT [32]	HOTSPOT	0.08	1	18.28 B
PATHFINDER [32]	PF	0.06	5	10.83 B
BinomialOptions[31]	BINO	0.02	1	23.6 B
MRI-Q [30]	MRI-Q	0.01	3	9.95 B

accesses across LLC slices, memory channels, and banks [18]. In the FR-FCFS memory controller, we cap the number of row buffer hits at 5 to avoid starvation [35], unless mentioned otherwise. Details regarding the simulated GPU architecture are provided in Table 1.

Workloads: We use a wide range of GPU benchmarks covering different domains. These benchmarks are selected from Rodinia [32], Parboil [30], CUDA SDK [31], PolyBench [33], and Mars [29], and are listed in Table 2. These benchmarks are classified as memory-bound versus compute-bound using the procedure described in Section 2. The memory-bound applications exhibit diverse memory behavior in terms of MPKI, see Table 2, and sensitivity to allocated SM count, see Figure 2b. We construct multi-program workloads by pairing these applications to obtain a total of 105 multi-program workload mixes. These 105 workloads consist of 50 heterogeneous mixes, pairing a memory-bound application with a compute-bound application; plus 45 homogeneous memory-bound mixes and 10 homogeneous compute-bound mixes.

We simulate five million cycles for each workload mix — this is in line with prior GPU multi-tasking research [3, 14], and we confirm that this is representative. If a benchmark finishes before others, it is re-launched and re-executed from the beginning. The reported performance results are gathered only for the first run for each of the benchmarks. Private-mode performance for each benchmark in the workload mix is determined by simulating the benchmark in isolation for the same number of instructions as in shared mode.

Metrics: We use three multi-program metrics: fairness, system throughput (STP) and average normalized turnaround time (ANTT) [24]. As in previous work [7], we define a system to be fair if the NPs of equal-priority applications are the same. STP quantifies overall system performance (higher-is-better). ANTT focuses on per-application performance and quantifies average per-application execution time (lower-is-better).

6 Evaluation

In this section, we first compare the prediction accuracy of the following private-mode GPU performance models:

- **HSM:** The HSM model as described in Section 2.
- **DASE [14]:** The state-of-the-art white-box approach.
- **Themis [16]:** The state-of-the-art black-box approach.

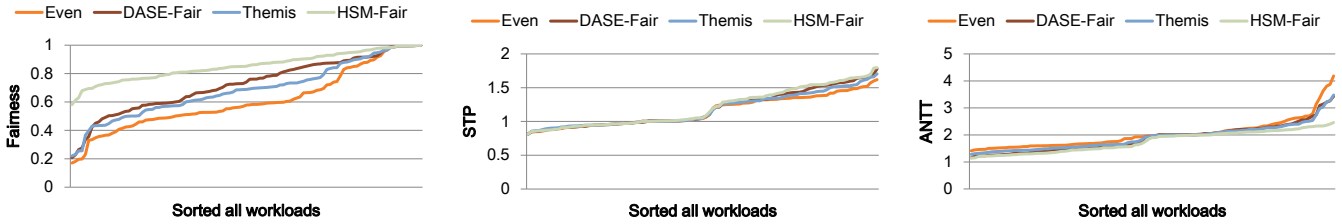
DASE and Themis sometimes grossly overestimate slowdowns, hence we cap their predictions at S_{Max}/S_k to improve accuracy. S_{Max} is the number of SMs in the GPU, and S_k is the number of shared-mode SMs allocated to kernel k. Applying this optimization reduces DASE's (Themis') average prediction error from 119.9% to 17.9% (69.9% to 33.8%).

We next evaluate HSM's applicability for steering SM allocation, for which we consider the following schemes:

- **Baseline:** The **Even** scheme allocates SMs evenly among the co-executing applications.
- **Fairness: HSM-Fair** uses HSM to allocate SMs to applications to improve fairness. Similarly, **Themis-Fair** and **DASE-Fair** implement fairness schemes using the Themis [16] and DASE [14] models, respectively.
- **QoS: HSM-QoS** uses HSM to allocate a sufficient number of SMs to a high-priority application so that its NP is approximately equal to a user-supplied target (e.g., 0.8). **Themis-QoS** and **DASE-QoS** implement QoS policies with Themis [16] and DASE [14], respectively.
- **Bandwidth partitioning:** Application-aware memory bandwidth partitioning [36] (**BW-App**) prioritizes the memory requests of the application with the lowest bandwidth utilization within the memory controller.

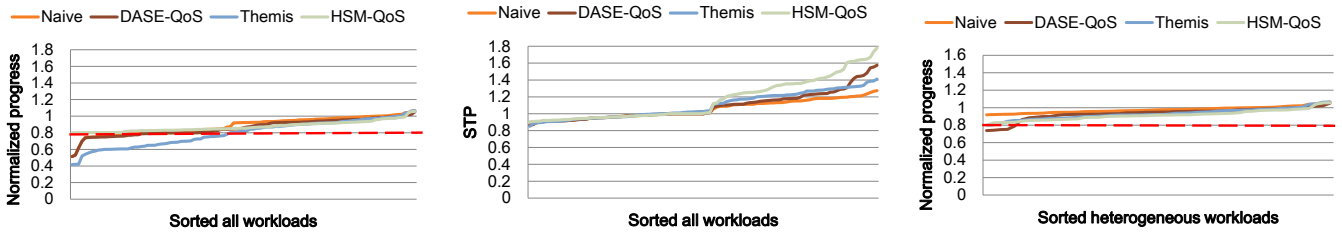
6.1 HSM Accuracy

Figure 10 quantifies HSM's NP prediction accuracy compared to the state-of-the-art DASE and Themis. The average prediction error equals 6.8% for HSM (max error of 30.3%),



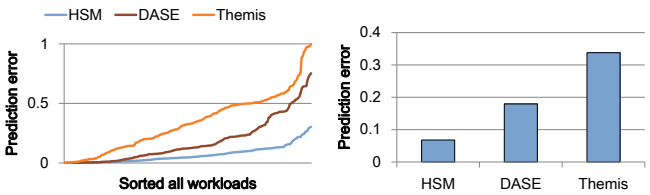
(a) Fairness **(b)** STP **(c)** ANTT

Figure 11. Fairness-aware SM allocation for all workload categories. *HSM-Fair significantly improves fairness compared to even partitioning and DASE for all workload mixes, while improving STP and ANTT for many.*



(a) QoS (all workloads) **(b)** STP **(c)** QoS (heterogeneous workload mixes)

Figure 12. QoS-aware SM allocation for a QoS target of 0.8. *HSM-QoS meets the 0.8 NP target for all workloads while improving STP; even partitioning and DASE do not always meet the QoS target, and yield suboptimal STP.*



(a) S-curves **(b)** Average error

Figure 10. HSM accuracy across all workload mixes. *HSM achieves significantly higher accuracy compared to state-of-the-art DASE (white-box) and Themis (black-box).*

whereas the average prediction errors of DASE and Themis are 17.9% (75.3% max error) and 33.8% (99.8% max error), respectively. We sort the prediction errors of each scheme in Figure 10a to illustrate that HSM has a significantly better error distribution than DASE and Themis. In fact, HSM is more accurate than DASE and Themis in nearly all workloads; the exception is a few easily predictable workloads where all schemes have less than 5% error.

DASE estimates application slowdown by modeling inter-application interference within each individual shared resource, including the shared LLC, DRAM channels, banks and row buffers — as previously done for CPUs [35, 37, 38]. Unfortunately, predicting shared resource interference effects at a per-request granularity leads to highly inaccurate predictions because of the high degrees of parallelism observed in GPU memory systems, i.e., memory requests heavily overlap with each other and hence per-request latency predictions do not accurately capture the effect of interference on overall application performance. Themis' neural network model is too simple to fully learn the relationship between online performance counters and slowdown. Therefore, it has decent prediction accuracy for the relationships captured by the model; otherwise, the errors are (much) larger.

6.2 Fairness-Aware SM Allocation

We now leverage HSM to steer SM allocation. As aforementioned, accurately predicting the normalized progress of an application is at the foundation for model-based SM allocation. In this section, we focus on fairness-aware SM allocation, which means that we allocate SMs to applications to fairly balance the applications' normalized progress.

Figure 11 reports fairness, STP and ANTT when optimizing for fairness. HSM-Fair improves fairness for all workload mixes. On average, we find that HSM-Fair achieves a fairness of 84.1% compared to 52.7%, 61.9%, and 65.2% for even partitioning, Themis, and DASE, respectively — an improvement by 1.59×, 1.36×, and 1.29×, respectively. HSM-Fair, while optimizing for fairness, also improves STP and ANTT for heterogeneous workload mixes by allocating more SMs to the compute-bound applications: we note an average improvement in STP by 7.2% and up to 12.2%; ANTT improves by 15.8% on average and up to 20.4%.

6.3 QoS-Aware SM Allocation

QoS-aware SM allocation provides a user-specified QoS target for a high-priority application. The SMs not allocated to the high-priority application are allocated to the other application to improve STP. We randomly denote one of the applications in the workload mix as the high-priority application in the homogeneous mixes; we denote the memory-bound application as the high-priority application in the heterogeneous mixes. Figure 12 quantifies QoS and STP across all workloads for a QoS target at 0.8 NP for the high-priority application, and compares against a baseline that grants 80% of the SMs (64 out of 80 SMs) to the high-priority application. HSM-QoS meets the QoS target for all workloads whereas the baseline (proportional partitioning), Themis, and DASE do not. Moreover, HSM-QoS improves STP significantly by providing the SMs that are not needed by the high-priority application to meet its QoS target, to the low-priority application. HSM-QoS improves STP by 7.7%, 4.4%, and 5.5% on average across all workloads compared to the baseline, Themis, and DASE, respectively, while meeting the user-specified QoS target; for the more challenging heterogeneous workloads, HSM-QoS improves STP by 18.9%, 13.0%, and 15.2% compared to the baseline, Themis, and DASE, respectively. Themis aggressively removes SMs from memory-bound applications, resulting in higher average STP than DASE for the heterogeneous workloads at the cost of significant QoS violations for mixes with two memory-bound applications (see Figure 12a).

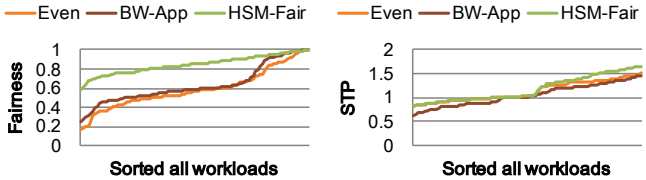
6.4 Memory Bandwidth Partitioning

We now compare HSM-Fair to application-aware memory bandwidth partitioning [36] (i.e., BW-App) and even partitioning across all workloads (see Figure 13). We conclude that HSM-Fair outperforms BW-App in terms of fairness and STP. We make a distinction between homogeneous memory-bound workload mixes and heterogeneous mixes.

For memory-bound workloads, BW-App improves fairness by 18.1% on average compared to even partitioning. This improvement is not as high as for HSM-Fair, which improves fairness by 75% on average. Memory bandwidth partitioning loses effect when memory requests from a highly memory-bound application with a high memory access rate dominate (or even worse, fully occupy) the memory controller queue, leaving the controller little to no freedom to reorder requests from other applications. In contrast, HSM-Fair moderates an application's memory access rate by repartitioning the SMs, thereby also indirectly partitioning memory bandwidth.

For heterogeneous workloads, BW-App is not effective because the compute-bound application needs more SMs to improve fairness, not memory bandwidth. BW-App improves fairness by 5.3% on average, whereas HSM-Fair improves fairness by 55.5%[4].

[4]TLP management techniques [39], which manage shared cache and memory bandwidth by moderating the number of warps allocated per application within an SM, suffer from the same limitation. Because of even SM allocation, the attainable fairness is equally limited because a compute-bound application needs more SMs, not memory bandwidth, to improve fairness.



(a) Fairness **(b)** STP

Figure 13. Fairness and STP compared to memory bandwidth partitioning. *HSM-Fair significantly improves fairness compared to memory bandwidth partitioning.*

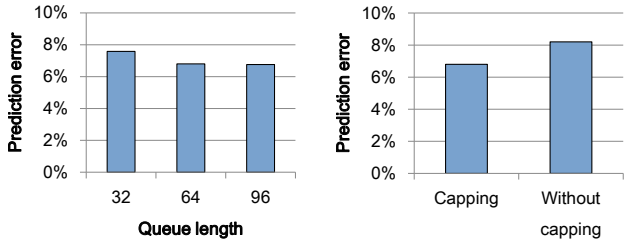
6.5 Sensitivity Analyses

Fairness threshold: Section 4.3 described how HSM-Fair uses a fairness threshold to guide fairness-aware SM allocation. We evaluate HSM-Fair's sensitivity to that threshold (ranging from 0.7 to 0.95) and observe that as long as the fairness threshold is set to at least 0.9, as done for all the results in the paper, HSM-Fair significantly improves fairness, meanwhile improving ANTT and yielding stable STP.

Training the black-box model: HSM uses linear regression to predict memory bandwidth utilization (dependent variable) as a function of row buffer hit rate (independent variable). Constructing this model requires a number of training workloads. To evaluate the model's robustness, we randomly select couples of benchmarks (one benchmark with an RBH below 0.3 and one benchmark with an RBH above 0.7) to determine the model's constants c_1 and c_2. We conclude that the variation for both constants is less than 0.2%.

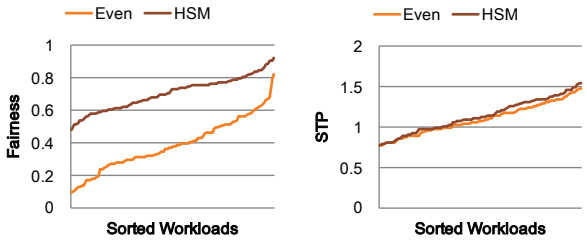
Memory controller parameters: Figure 14a explores the accuracy impact of DRAM controller queue size, and Figure 14b analyzes the prediction accuracy impact of forcing the FR-FCFS scheduler to service the oldest request after 5 consecutive row hits (i.e., capping [35]). Overall, HSM's prediction accuracy is robust across memory controller configurations (average prediction error variability within 2%). The reason is that memory-bound kernels issue a large number of requests. Thus, the memory controller can generally find enough row hits to protect RBH. Further, we confirm that the throughput-impact of capping is minor; average STP with capping is within 2.5% of no capping.

GDDR5: The results presented so far assumed an HBM memory system attached to an aggressive 80-SM GPU. HSM is equally accurate and effective for a more moderate GPU system with 40 SMs and a GDDR5 memory system. The results are similar: HSM achieves an average prediction error of 6.5%, and yields a fairness of 88.7% when used for fairness-aware SM allocation.



(a) Memory controller queue length **(b)** FR-FCFS row hit capping

Figure 14. HSM accuracy sensitivity to memory controller parameters. *Overall, HSM's accuracy robust to different memory controller configurations.*



(a) Fairness **(b)** STP

Figure 15. Fairness and STP for 4-program workloads. *Again, HSM-Fair significantly improves fairness.*

Four-program workload mixes: HSM is trivially extended to multi-application workloads by iteratively focusing on the two applications with the highest and lowest NP in each epoch and balancing fairness among those. HSM-Fair improves fairness by 2.2× compared to even partitioning, while delivering similar STP, see Figure 15. The STP improvement is not as high as for the two-program workloads because compute-bound applications gets relatively fewer SMs allocated.

7 Related Work

Slowdown prediction: Accurately predicting an application's slowdown is critical to fairness- and QoS-aware multitasking GPUs. HSM is the first accurate and low-complexity hybrid slowdown model. DASE [14] is a complex white-box model which we have shown provides much lower prediction accuracy than HSM. Similarly, we have compared to black-box Themis [15, 16] and shown that its neural network model is too simple to fully learn the relationship between online performance counters and application slowdown.

Predicting the private-mode performance of co-runners has been extensively studied in the CPU domain. MISE [12] and ASM [13] predict private-mode performance by periodically giving each co-runner highest priority in the memory controller. This approach is not suitable for GPUs since the requests of a memory-intensive low-priority kernel will fill the queues of the memory controller and thereby slow down the high-priority kernel. GDP [10], PTCA [11], and ITCA [40] are white-box approaches that use online counters and architectural insights to predict private-mode latencies of individual memory requests. Unfortunately, these CPU-oriented proposals cannot be straightforwardly adapted to GPUs — because of the high degree of overlap effects in GPUs and their sensitivity to memory bandwidth rather than latency.

GPU resource management in multitasking GPUs: A number of prior proposals rely on heuristics to manage memory bandwidth in multitasking GPUs. Jog et al. [36, 41] propose application-aware memory schedulers, while Wang et al. [39] scale resources within an SM to manage memory bandwidth. We find that (i) memory bandwidth partitioning does not provide fairness for heterogeneous workloads, and (ii) SM partitioning is a more effective solution managing both compute and memory bandwidth resources.

Another class of related work uses offline profiling to determine private-mode performance and dynamically allocate resources. Aguilera et al. [4, 42] adjust the number of SMs allocated to applications to improve fairness and QoS. Wang et al. [43] use fine-grained sharing of SM-internal resources to improve QoS. HSM would greatly benefit these approaches as users would no longer be required to retrieve representative private-mode performance numbers for all applications.

Managing SMs among concurrent applications in multitasking GPUs received significant attention recently [3, 4, 14, 15, 26, 42]. These approaches indirectly infer the performance impact of a particular SM allocation; HSM, in contrast, predicts the performance impact of a particular SM allocation.

Memory bandwidth management: A number of works target memory controller scheduling policies in single-tasking GPUs. Chatterjee et al. [44] introduce the warp-aware memory scheduler to reduce memory latency divergence within a warp. MeDiC [45] focuses on inter-warp heterogeneity and concurrently considers the memory controller scheduling policy, LLC bypassing, and LLC insertion. Lakshminarayana et al. [46] propose a scheduling policy to dynamically choose between the shortest job first policy and FR-FCFS, while Jog et al. [47] introduce a criticality-aware memory scheduler. Li et al. [48] propose an inter-core locality-aware memory scheduler which prioritizes memory requests with high inter-core locality. This body of prior work considered single-tasking GPUs, in contrast to this work. Moreover, we demonstrate that HSM-enabled SM allocation holistically manages both the SM and memory bandwidth resources.

A number of schemes have been proposed for managing memory bandwidth in CPUs (e.g., [35, 49, 50]), but these are not readily applicable to GPUs since they focus on latency while GPUs are inherently latency-tolerant.

8 Conclusion

This paper presented the Hybrid Slowdown Model (HSM) to accurately predict the slowdown of co-running applications (average error of 6.8%) which provides a foundation for interference-aware SM resource allocation policies in multitasking GPUs. HSM combines a white-box model derived from fundamental architectural insights — to reduce training and implementation overheads — with a black-box model that accounts for the highly concurrent GPU execution model — to achieve high accuracy. To showcase the capabilities of HSM, we proposed two HSM-based fairness- and QoS-aware SM-allocation policies: HSM-Fair and HSM-QoS. HSM-Fair improves fairness by 1.59× on average compared to even SM partitioning. HSM-QoS maintains the NP target of the high-priority application while improving system throughput by 18.9% on average compared to even SM partitioning for challenging heterogeneous workload mixes.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work was supported in part by European Research Council (ERC) Advanced Grant agreement no. 741097, FWO projects G.0434.16N and G.0144.17N, NSFC under Grants no. 61572508, 61672526 and 61802427. Xia Zhao was supported through a CSC scholarship and UGent-BOF co-funding. Magnus Jahre is supported by the Research Council of Norway (Grant no. 286596).

A Artifact Appendix

A.1 Abstract

This appendix explains how we submitted our modified version of GPGPU-sim v3.2.2 [34] to the ASPLOS'20 artifact evaluation process. We provided our simulator infrastructure to the artifact evaluators as a virtual machine image (Virtual Box). The image contained simulator sources, precompiled simulator and benchmark binaries, input files, and the complete result set.

A.2 Artifact check-list (meta-information)

- Program: CUDA programs on modified GPGPU-sim v3.2.2
- Compilation: gcc and nvcc
- Run-time environment: Ubuntu
- Hardware: x86-64 CPU with at least 16 GB
- Metrics: Instruction per cycle (IPC)
- Output: Simulation results
- Experiments: Single application and multi-application simulation
- How much disk space required (approximately)?: >10 GB
- How much time is needed to prepare workflow (approximately)?: Less than one hour
- How much time is needed to complete experiments (approximately)?: Most simulations finish within two days
- Publicly available?: No

References

[1] Amazon, "Amazon web services." https://aws.amazon.com/cn/ec2/.
[2] Google, "Graphics Processing Unit (GPU) | Google Cloud." https://cloud.google.com/gpu/.
[3] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The Case for GPGPU Spatial Multitasking," in *Proceedings of the International Symposium on High-Performance Comp Architecture (HPCA)*, pp. 1–12, February 2012.
[4] P. Aguilera, K. Morrow, and N. S. Kim, "Fair Share: Allocation of GPU Resources for Both Performance and Fairness ," in *Proceedings of the International Conference on Computer Design (ICCD)*, pp. 440–447, October 2014.
[5] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 593–606, March 2015.
[6] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling Preemptive Multiprogramming on GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 193–204, June 2014.
[7] R. Gabor, S. Weiss, and A. Mendelson, "Fairness and Throughput in Switch on Event Multithreading," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 149–160, December 2006.
[8] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," in *Proceedings of the International Conference on Supercomputing (ICS)*, pp. 257–266, June 2004.
[9] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Thoughput and Fairness in SMT Processors," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 164–171, November 2001.
[10] 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 (HPCA)*, pp. 296–309, 2018.
[11] K. Du Bois, S. Eyerman, and L. Eeckhout, "Per-Thread Cycle Accounting in Multicore Processors," *ACM Transactions on Architecture and Code Optimization*, vol. 9, pp. 1–22, March 2013.
[12] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 639–650, February 2013.
[13] 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 *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 62–75, December 2015.
[14] Q. Hu, J. Shu, J. Fan, and Y. Lu, "Run-Time Performance Estimation and Fairness-Oriented Scheduling Policy for Concurrent GPGPU Applications," in *International Conference on Parallel Processing (ICPP)*, pp. 57–66, August 2016.
[15] W. Zhao, Q. Chen, and M. Guo, "KSM: Online Application-Level Performance Slowdown Prediction for Spatial Multitasking GPGPU," *IEEE Computer Architecture Letters*, vol. 17, pp. 187–191, July 2018.
[16] W. Zhao, Q. Chen, H. Lin, J. Zhang, J. Leng, C. Li, W. Zheng, L. Li, and M. Guo, "Themis: Predicting and reining in application-level slowdown on spatial multitasking GPUs," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.
[17] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.
[18] Y. Liu, X. Zhao, M. Jahre, Z. Wang, X. Wang, Y. Luo, and L. Eeckhout, "Get Out of the Valley: Power-Efficient Address Mapping for GPUs," in

Proceedings of the International Symposium on Computer Architecture (ISCA), pp. 166–179, June 2018.

[19] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 128–138, June 2000.

[20] J. E. Smith, "Decoupled access/execute computer architectures," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 289–308, 1984.

[21] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, "Architecting an Energy-Efficient DRAM System for GPUs," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 73–84, February 2017.

[22] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 41–54, December 2017.

[23] JEDEC, *High Bandwidth Memory (HBM) DRAM.* 2015.

[24] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, vol. 28, pp. 42–53, May 2008.

[25] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 423–432, December 2006.

[26] X. Zhao, Z. Wang, and L. Eeckhout, "Classification-Driven Search for Effective SM Partitioning in Multitasking GPUs," in *Proceedings of the International Symposium on Supercomputing (ICS)*, pp. 65–75, June 2018.

[27] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 15, pp. 45–49, January 2016.

[28] "Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0." http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf, 2009. Hynix.

[29] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 260–269, October 2008.

[30] 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," tech. rep., March 2012.

[31] "NVIDIA CUDA SDK Code Samples." https://developer.nvidia.com/cuda-downloads.

[32] 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 *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pp. 44–54, October 2009.

[33] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a High-Level Language Targeted to GPU Codes," in *Proceedings of Innovative Parallel Computing (InPar)*, pp. 1–10, May 2012.

[34] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proceeding of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 163–174, April 2009.

[35] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 146–160, December 2007.

[36] 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 *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS)*, pp. 223–234, October 2015.

[37] 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 *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 335–346, March 2010.

[38] M. Jahre and L. Natvig, "A Light-Weight Fairness Mechanism for Chip Multiprocessor Memory Systems," in *Proceedings of the Conference on Computing Frontiers (CF)*, 2009.

[39] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog, "Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pp. 247–258, February 2018.

[40] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "ITCA: Inter-Task Conflict-Aware CPU Accounting for CMPs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 203–213, September 2009.

[41] 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 *Proceedings of Workshop on General Purpose Processing Using GPUs (GPGPU)*, pp. 1:1–1:8, March 2014.

[42] P. Aguilera, K. Morrow, and N. S. Kim, "QoS-Aware Dynamic Resource Allocation for Spatial-Multitasking GPUs," in *Proceedings of the International Conference on Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 726–731, January 2014.

[43] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Quality of Service Support for Fine-Grained Sharing on GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 269–281, June 2017.

[44] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing DRAM Latency Divergence in Irregular GPGPU Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 128–139, November 2014.

[45] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, "Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance," in *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, pp. 25–38, October 2015.

[46] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin, "DRAM Scheduling Policy for GPGPU Architectures Based on a Potential Function," *IEEE Computer Architecture Letters*, vol. 11, pp. 33–36, July 2012.

[47] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Exploiting Core Criticality for Enhanced GPU Performance," in *Proceedings of the International Conference on Measurement and Modeling of Computer Science (SIGMETRICS)*, pp. 351–363, June 2016.

[48] D. Li and T. M. Aamodt, "Inter-Core Locality Aware Memory Scheduling," *IEEE Computer Architecture Letters*, vol. 15, pp. 25–28, January 2016.

[49] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 374–385, December 2011.

[50] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pp. 63–74, June 2008.