Quality of Service Support for Fine-Grained Sharing on GPUs

Zhenning Wang
Department of Computer Science
Shanghai Jiao Tong University
znwang@sjtu.edu.cn

Jun Yang
Electrical and Computer Engineering Department
University of Pittsburgh
juy9@pitt.edu

Rami Melhem
Department of Computer Science
University of Pittsburgh
melhem@cs.pitt.edu

Bruce Childers
Department of Computer Science
University of Pittsburgh
childers@cs.pitt.edu

Youtao Zhang
Department of Computer Science
University of Pittsburgh
zhangyt@cs.pitt.edu

Minyi Guo
Department of Computer Science
Shanghai Jiao Tong University
guo-my@cs.sjtu.edu.cn

ABSTRACT

GPUs have been widely adopted in data centers to provide acceleration services to many applications. Sharing a GPU is increasingly important for better processing throughput and energy efficiency. However, quality of service (QoS) among concurrent applications is minimally supported. Previous efforts are too coarse-grained and not scalable with increasing QoS requirements. We propose QoS mechanisms for a fine-grained form of GPU sharing. Our QoS support can provide control over the progress of kernels on a per cycle basis and the amount of thread-level parallelism of each kernel. Due to accurate resource management, our QoS support has significantly better scalability compared with previous best efforts. Evaluations show that, when the GPU is shared by three kernels, two of which have QoS goals, the proposed techniques achieve QoS goals 43.8% more often than previous techniques and have 20.5% higher throughput.

CCS CONCEPTS

• **Computer systems organization** → **Multiple instruction, multiple data**; • **Applied computing** → *Data centers*;

KEYWORDS

GPU, Quality of Service

ACM Reference format:
Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2017. Quality of Service Support for Fine-Grained Sharing on GPUs. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080203

1 INTRODUCTION

GPUs have been widely adopted in many systems to accelerate compute-intensive applications, such as MapReduce [15], Graph Processing [47] and Deep Learning [14, 40]. A GPU has a massive number of simple compute cores grouped into streaming multiprocessors (SM) [27]. GPUs exploit thread-level parallelism (TLP) with these cores, hiding memory latency through heavy multi-threading.

In today's GPU-equipped systems, multiple applications may need to share a single GPU at the same time. In desktop computing, for instance, HD video players may co-execute with live video calling and graphic processing required by the OS. While sharing a GPU by multiple applications can be implemented with various mechanisms [25, 37, 39], there is little support in current GPU hardware to manage quality of service (QoS) among applications that share the GPU. Tasks such as graphic processing need to be executed in a responsive manner to guarantee a good user experience. Failing to meet QoS may lead to an unsatisfactory user experience, such as game lags and frame drops. Non-graphic tasks may also have performance requirements. For example, data centers often provide services to users who require applications to progress at certain rates.

Initial effort to address QoS for GPUs used modified device drivers, invoked system call traps/API to schedule GPU commands (device initialization, data transfers, kernel launch, etc.) or controlled the order of multiple kernels [18, 25, 32]. Applications share the GPU in a time-multiplexed manner at the granularity of kernel execution[1]. Once a kernel is launched onto the GPU, it cannot be easily interrupted because current GPUs do not support preemption. Other applications have to wait for the completion of the running kernel. Thus, these previous techniques work best for short kernels, as long-running kernels would block waiting kernels for a long time.

Recently, there has been momentum to support preemption in GPUs. The Heterogeneous System Architecture (HSA) specification [11] defines three kinds of preemption: 1) soft preemption, where hardware can delay the preemption request; 2) hard preemption, where hardware has to save the context to memory; and 3) context reset, where hardware drops the context of the executing kernel.

Along with industrial advances, there had been academic efforts to investigate hard preemption [37, 41, 42] and context reset [31]. Further, support for preemption has been studied to improve sharing. Rather than time-multiplex the GPU, kernels can now run concurrently on the same GPU by hot swapping between executing and pending kernels [37, 41, 42]. In [37], a context switch is performed at the granularity of one SM, i.e., the context of kernels can be

[1] An application using the GPU has multiple kernels, each capable of spawning many threads that are grouped into thread blocks (TB)

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080203

swapped in integer number of SMs. Hence, different application kernels can execute concurrently on disjoint sets of SMs, i.e., a spatially partitioned multitasking of a GPU. Fairness or QoS can be managed by adjusting the number of SMs in each partition [2, 3, 37]. However, previous studies [30, 37, 42] show that there is a great resource under-utilization problem in applications across various domains, and such under-utilization is mainly within each SM. Hence, partitioning SMs among kernels cannot address this problem because each SM still executes one kernel at a time. Therefore, managing fairness or QoS by adjusting the number of SMs is too coarse-grained and suboptimal.

An improved form of sharing was proposed in which sharing is performed in each SM [23, 30, 41, 42, 44]. These schemes allow multiple kernels to co-run at the level of an SM, rather than being spatially partitioned between SMs. This fine-grained sharing is enabled at run-time with Partial Context Switch [41, 42]. This technique swaps the context of kernels in an integer number of TBs (a.k.a. cooperative thread array). This finer unit of context has less swap cost than swapping at the granularity of an SM. Fine-grained sharing achieves better resource utilization and GPU throughput than coarse-grained sharing [41, 42]. Fairness can be maintained by adjusting within-SM resources among sharer kernels, which is more effective than coarse-grained sharing due to more precise resource management [42].

In this paper, we develop QoS mechanisms in multitasking GPUs with fine-grained sharing for datacenter-scale workloads. The difference between fairness and QoS is that the former tries to equalize a specific metric, such as performance, among all kernels, while the latter differentiates the metric and guarantees it for only some kernels. Hence, their resource allocation algorithms are vastly different. We demonstrate that QoS management with fine-grained sharing is more effective and more scalable than coarse-grained sharing. Kernels with QoS goals can receive "just enough" resources to reach their goals. Kernels without QoS goals can use any remaining resources to maximize their throughput. The main advantage of our scheme over coarse-grained sharing is that it can, through warp scheduling, manage the amount of progress of each sharer kernel. The scheme more accurately achieves a performance goal than the coarse-grained strategy, which adjusts how many SMs a kernel receives. Due to more precise control, our techniques are also more scalable to the number of sharers and the number of QoS goals. The contributions of this work are:

- *Lightweight cycle-level QoS management techniques for fine-grained GPU sharing.* We develop simple quota allocation schemes, in terms of instruction count to control the progress of each sharer kernel. Quotas are derived from QoS goals, and are added to the unmodified warp scheduling algorithm to preserve its original property. The schemes provide just enough quotas for reaching QoS goals. Excessive quotas are used to maximize GPU throughput when all QoS goals are met.
- *A static resource adjustment technique for sharer kernels.* We develop a static resource allocation scheme in unit of TBs for GPU sharers. This scheme provides enough but not excessive thread-level parallelism to kernels so as to optimize overlapped execution. Resource allocation is also performed with caution to reduce context switch overhead, achieving better efficiency.

In our evaluation, we used 90 kernel pairs and 60 kernel trios from Parboil, sweeping through 10 QoS goals. Our approach not only achieves more QoS goals than coarse-grained sharing, but it is also more scalable when increasing the number of kernels and QoS goals. On average, our techniques achieve QoS goals 43.8% more often for trios and 12.2% more often for pairs than spatial partitioning-based management. The total GPU throughput is also 20.5% higher for trios and 15.9% higher for kernel pairs, on average.

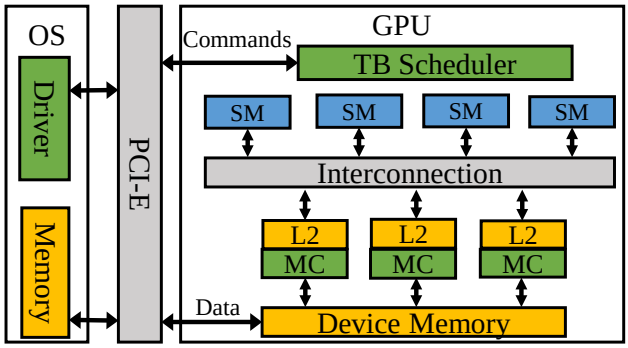


Figure 1: System Overview

2 BACKGROUND AND PRIOR ART

In this section, we describe the architecture of a GPU and its execution model. We use NVIDIA/CUDA terminology, but most descriptions also apply to the GPUs of other vendors.

2.1 GPU Architecture

GPUs are co-processors that cannot initialize on their own. Instead, the runtime and its driver in the OS control the GPU by sending commands over the PCI-E bus to perform initialization, data transfers, kernel launch and synchronization between the OS and the GPU. Figure 1 shows an overview of a GPU system architecture.

The GPU has multiple Streaming Multiprocessors (SMs), each containing many compute cores and resources such as registers, shared memory and L1 cache. SMs share device memory through an interconnect network. Memory requests are distributed to memory controllers according to address. Each memory controller has its own L2 cache.

2.2 GPU Execution Model

A GPU application consists of multiple kernels, each performing a specific task. Kernels are SIMT (Single Instruction Multiple Threads) programs for a GPU that may or may not have dependencies among them. The programmer writes code for one thread, and the GPU generates many threads executing the same code. The total number of threads is specified by the programmer.

Threads are grouped into thread blocks (TB), which are dispatched to the GPU one at a time. The resource demand per thread is determined by the compiler and the number of threads per TB is specified by the programmer. Hence, the GPU can easily calculate the resource requirements of a TB, and how many TBs an SM can hold. An SM can continue taking an integer number of TBs, until

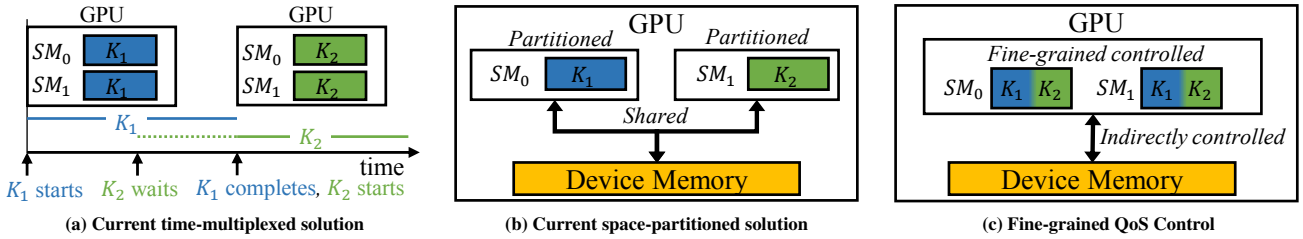


Figure 2: Different GPU sharing strategies.

one of the required resources, such as the registers, the scratchpad memory (also called Shared Memory in CUDA terminology), or the maximum number of threads and TBs, reaches the SM limit. Other remaining resources are left unused. If the total GPU resources are not enough to dispatch all TBs in a kernel, the remaining TBs wait for executing kernels to finish and release resources.

Once a TB is dispatched to an SM, its threads are batched into warps by the hardware, 32 at a time (the SIMD width of the GPU). Registers, shared memory and caches are shared by the warps within one SM. An SM has one or more warp schedulers, and warps are distributed equally to the schedulers. Schedulers select SIMD instructions from different warps to execute according to certain policies to optimize kernel performance. Warps can be stalled due to long latency operations. The schedulers select instructions from ready warps, if there are any, to keep the pipeline busy. Hence, long latency operations can be hidden under execution cycles, achieving high GPU throughput.

2.3 Existing GPU Sharing Mechanisms and QoS

The conventional way of running multiple kernels concurrently is to use multiple GPUs, one for each kernel. This guarantees the performance of each individual kernel. However, many past studies demonstrated that on-chip resources of GPUs are often greatly underutilized when running a single kernel. This previous work showed that sharing the GPU among multiple kernels improves resource utilization, overall GPU throughput, and energy efficiency more than running a single kernel alone [30, 31, 37, 39, 41, 42, 44]

To date, there are four kinds of sharing mechanisms for a GPU. In the first type, the state-of-the-art GPU architecture has preliminary support for launching kernels from different applications concurrently, e.g., Hyper-Q [6] and MPS [26]. However, the software has no control over how different kernel's TBs are dispatched into SMs. Most likely, different kernels will still execute sequentially rather than in parallel [43].

The second type of sharing is a software approach that fuses two kernels into one kernel through code transformation [30, 39]. The two kernels effectively become one kernel so that both can be resident in each SM, achieving a fine-grained sharing mechanism. Xu et al. [44] designed a profiling-based TB allocation scheme for sharer kernels to improve performance and fairness. The limitation is that hardware recognizes multiple kernels as one kernel, and hence, it cannot control the execution progress of each kernel. Therefore, performance of particular kernels and QoS cannot be guaranteed. Additionally, such sharing is enabled statically so dynamically arriving high-priority kernels cannot be serviced by the GPU.

The third type of sharing is implemented through time multiplexing kernels, in a way analogous to scheduling jobs in a CPU [7, 10, 12, 35, 46]. This can be achieved by intercepting the kernel launch command from different applications. An application that has consumed long GPU time is blocked from launching more kernels, and yields to an application that needs more GPU time [8]. This method uses MPS [26] to allow concurrent kernel execution; it suffers from the same limitation that most kernels are still executed sequentially. Hence, it essentially does coarse-grained sharing and does not improve resource utilization, overall kernel throughput or energy efficiency. QoS can only be done among multiple applications, but not among multiple kernels which could be long-running jobs. Figure 2a shows an example of this strategy. Kernel K_1 and K_2 use the GPU sequentially, even though they temporally overlap.

The fourth type of sharing is enabled by hardware support for preemption, which can be done through saving context of a running kernel one SM at a time [31, 37], or partial SM at a time [41, 42]. A new kernel's context is then loaded to start execution on the same GPU, sharing on-chip resources with the exiting kernel. Sharer kernels can either partition the SMs spatially within the GPU, as shown in Figure 2b [31, 37], or can share every single SM as shown in Figure 2c [41, 42]. Preemption-based sharing subsumes the other three types of sharing. Sharer kernels can achieve higher overall GPU throughput while allowing dynamic switching among different kernels. More importantly, QoS among the sharer kernels is now possible by dynamically adjusting resource usage. For example, QoS for spatially partitioned sharing can be performed by adjusting the number of SMs for each kernel to achieve performance goals with hill-climbing [3]. Fine-grained sharing through Simultaneous Multikernel (SMK) [41, 42], manages resources to achieve *fair* execution among sharer kernels, meaning that the kernel's performance in a shared mode degrades equally when compared with isolated execution. However, for QoS management, if a kernel's performance goal should be achieved, then policies for fairness should not be enforced.

In this paper, we build on fine-grained sharing of GPUs by providing QoS management for the sharer kernels. We demonstrate that this control leads to better QoS enforcement than the best previously proposed scheme.

3 QOS DESIGN WITH FINE-GRAINED SHARING

We assume that among the sharer kernels, there are one or more kernels, termed "QoS kernels", that have QoS goals. Other kernels are termed "non-QoS kernels". For QoS kernels, the objective is to meet each kernel's individual QoS goal. For non-QoS kernels, the objective is to maximize their total throughput. In this work, we assume that the QoS kernels are repeatedly executing datacenter-scale workloads, and their performance and execution length can be predicted [8]. QoS management will allocate resources dynamically such that QoS kernels receive just enough resources to achieve their goals, while leaving unused resources for non-QoS kernels.

3.1 Resources to Manage

The first kind of resource is static, such as registers, shared memory and threads, represented as number of TBs. More TBs means more threads and higher TLP. Having sufficient amount of TLP is necessary to keep kernels busy. However, too much kernel TLP can become overkill due to the possibility of high cache contention in a compute-intensive kernel [20], or too much memory traffic in a memory-intensive kernel [3, 19]. Moreover, QoS cannot be managed by static resource allocation alone because the warp scheduler might bias towards one kernel over another irrespective of the amount of TLP present. However, it is important to start from a good static resource allocation to ensure enough TLP per kernel and permit dynamic resource management to quickly reach an allocation that satisfies QoS.

The second kind of resource is dynamic, including memory bandwidth and core compute cycles within each SM. Our experience indicates that managing the memory bandwidth for QoS is difficult, because the correlation between performance and effective memory bandwidth is blurred by two level caching, memory coalescing, and TLP. Hence, it is difficult to capture a bandwidth-performance model for online guidance. In addition, bandwidth requirement of a kernel can vary drastically [33], which makes history-based allocation ineffective. Also, sharing the bandwidth is better than partitioning it among contenders when trying to maximize the overall performance [19]. Hence, QoS cannot be achieved by managing the bandwidth alone. Nevertheless, in our evaluation settings, we experimented with co-running both memory intensive and non-memory intensive kernels together, and the results show that our proposed resource manager is effective in achieving QoS goals and improving overall throughput.

The core compute cycles are managed by warp schedulers that decide which warp to execute in every cycle in order to achieve good performance. This is a more direct way to control the performance of each kernel within one SM. However, warp schedulers have been highly optimized to improve cache locality, memory behavior, synchronization barriers, etc. [4, 16, 22] of a single kernel. With fine-grained sharing, further warp scheduling across kernels for QoS should be done in a non-intrusive way, to keep the quality of the schedule already optimized for a single kernel.

As we can see, QoS control cannot be achieved by managing a single type of resources. We propose mechanisms that integrate the management of all static and dynamic resources, directly or indirectly for achieving QoS goals of sharer kernels in an SM.

3.2 From QoS Goals to Architectural Metrics

QoS goals are typically specified at the application level, which is independent of hardware, and hence, the goals cannot be directly used for architectural level QoS control. Therefore, the first problem we solve is to translate high-level QoS goals to architectural metrics which can be measured and controlled by the hardware. QoS goals can be in different forms, e.g., frame rate or data rate. In many GPU applications, such as video processing, a common programming style is to use one kernel to process one frame of data. Frame rate is equivalent to kernel completion rate, and a target frame rate can be enforced by setting the required execution time of every kernel. This method is also adopted by previous QoS work on GPU [8]. Therefore, our QoS management approach ensures *execution time*, or *average IPC* (IPC goal, as derived below) of each kernel, rather than the execution rate within a kernel.

The translation from QoS goals to IPC goals is done in the OS resident kernel scheduler. The end-to-end application level QoS requirement includes the pure kernel execution time, and other latencies such as memory copies (synchronous or asynchronous), contention over PCIe bus, and queuing. We assume the kernel scheduler is fully aware of those factors, and can calculate the true requirement for kernel execution time by calculating other timings and subtracting them from the overall QoS goal. As an example, the memory copy time in unified GPU architecture is negligible because the device driver can map the physical memory region to GPU virtual memory space. In a discrete GPU architecture, kernel data needs to be transferred via the PCI-E bus. Hence, the data transfer time is linear to the transferred data size, and can be calculated with the fixed latency of PCI-E bus and its bandwidth. There also might be contention on the PCIe bus, and different queuing delays may cause variability in starting a memory copy and kernel execution. For those reasons, in our evaluation configurations, we sweep the QoS goals through a range of goals from low to high, indicating a relatively easy-to-achieve to hard-to-achieve QoS requirement to accommodate different situations in a real system.

To calculate the IPC goal from pure kernel execution time, the total number of instructions of the kernel is also needed. In data centers, workloads are relatively stable and can be accurately predicated by the runtime or application with machine learning algorithms according to previous work [8]. With this information, IPC goals are calculated as below, and are passed to the GPU upon dispatching the kernel so that internal resource management can be done to achieve such an IPC.

$$IPC = \frac{Instructions\_of\_Kernel}{Frequency \times Kernel\_Execution\_Time}$$

In our evaluation, we assume that the IPC goal (IPC_{goal}, converted from the QoS goal) for an application can always be achieved when the application is run in isolation with $IPC_{isolated}$, and $IPC_{isolated} \geq IPC_{goal}$. The distance from IPC_{goal} to $IPC_{isolated}$ varies from application to application. In evaluation, we sweep the IPC_{goal} from a low percentage of $IPC_{isolated}$, e.g. 50%, to a high percentage of $IPC_{isolated}$, e.g., 95% to cover a wide range IPC_{goal} values and to show the effectiveness of our approaches.

Benefit to OS resident kernel schedulers. Our proposed mechanism will strengthen the capability of OS-level kernel schedulers that

aim to achieve QoS for applications sharing the GPU [8, 18, 25, 38]. OS-level scheduling either assumes isolated kernel execution [18, 25], or co-executed but un-managed kernels [8, 38]. Once kernels are scheduled to the GPU, no further control within the GPU can be made on the progress of each kernel. Hence, they rely on *when* a kernel should be dispatched to the GPU to achieve QoS. Our design fills in this gap to control how sharer kernels should use the resources within the GPU to achieve QoS, which increases the likelihood of meeting QoS goals even if a kernel has a late start. Likewise, our mechanism also relieves the burden of an OS scheduler in that the timing of dispatching a kernel to the GPU can be more relaxed than before. Furthermore, our mechanism improves the throughput of non-QoS kernels and leads to better energy efficiency.

3.3 Architecture Overview

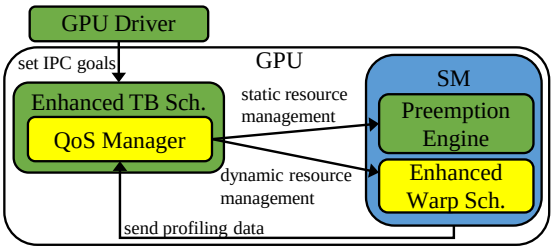


Figure 3: Overview of architecture extensions (in yellow) for QoS in fine-grained sharing of GPUs.

Figure 3 shows an overview of architecture extensions to enable QoS. Current GPUs can possibly allow co-execution of TBs from different kernels in a single SM, e.g., via MPS [26]. Although this type of sharing cannot be controlled, is unpredictable and not all architecture details are disclosed, the basic aspects (such as TLB and memory management) are incorporated in previous work [31, 37, 42] and this paper as well. Hence, in this paper, we focus only on the QoS design within the GPU. The original TB scheduler is enhanced with fine-grained sharing [42] and a QoS Manager. The Enhanced TB Scheduler interacts with each SM to perform static and dynamic resource management. Static resource management determines how many TBs from each kernel should be hosted by each SM. This allocation varies at run-time, via the preemption engine, according to QoS achievement. Dynamic resource management determines the progress of each kernel with a quota-based strategy. The QoS manager passes quotas to the warp scheduler, the Enhanced Warp Scheduler (EWS), which is extended to be QoS aware. Both static and dynamic resource management are necessary as the former ensures that the latter has the right amount of thread-level parallelism to enforce the progress of each kernel. The QoS manager collects run-time statistics to make allocation decisions.

Quota-Based Management Schemes. EWS allocates core compute cycles following quotas defined by the QoS Manager. Each kernel in an SM is given a quota that indicates how much progress it should make on an epoch-by-epoch basis. This strategy is compatible with previous work to manage fairness among sharer kernels at a fine grain [42], which allows QoS and fairness management to coexist. The GPU firmware can simply switch between different policies as needed by the run-time system.

As stated earlier, QoS control differentiates a metric and guarantees it for only a subset of sharers. A simple method could prioritize and allocate QoS kernels with enough quota for them to reach their goals, and then allocate any remaining cycles to non-QoS kernels. Yet this scheme has some problems. First, it degenerates to nearly sequential execution of sharer kernels, similar to conventional QoS management for CPUs where threads are given different time slices to execute on the CPU in a time-shared manner. The performance of each thread is linear to the total amount of CPU time slices obtained. In GPUs, however, the execution of kernels are greatly overlapped. Time slices allocated to one kernel are also consumed by all other kernels that run in parallel. Hence, it is difficult to account for how much time is consumed by which kernel. Second, if kernels are executed in a prioritized order, the parallel execution capability of the GPU is lost, resulting in poor hardware resource utilization and poor overall GPU performance. We report results for CPU-like QoS management in the evaluation.

With quotas, EWS does the usual scheduling but checks in each cycle if the quota of a kernel has been reached. Once a kernel consumes all of its quota, EWS no longer schedules any instructions from that kernel. This design has minimum impact on the quality of warp scheduling since the original warp scheduling algorithm is used throughout the lifetime of kernels, except that QoS kernels are throttled once their quotas are exhausted. Using quotas also indirectly controls memory bandwidth consumption because throttled QoS kernels do not generate additional memory traffic.

3.4 QoS Algorithms for QoS Kernels

The objective of QoS management is to satisfy QoS goals of QoS kernels while maximizing throughput of non-QoS kernels. Managing quota allocation and deallocation is the key component. As discussed before, an application's QoS requirement is translated into IPC_{goal} which is expressed as a quota in terms of *number of instructions* that should be executed per epoch. Since IPC_{goal} is the average IPC a kernel should eventually reach, the actual IPC achieved per epoch may vary due to factors such as instruction diversity and sharer kernel's varying behavior. Hence, an effective QoS algorithm is mandatory. We developed four quota allocation algorithms, as discussed below.

3.4.1 Naïve Allocation. The first scheme is a naïve one that calculates the quota based on the IPC_{goal} and the length of an epoch (T_{epoch}). Given an IPC_{goal} as a percentage of $IPC_{isolated}$, and T_{epoch}, we can calculate the total number of instructions that a kernel, k, should complete in an epoch:

$$Quota_k = IPC_{goal} \times T_{epoch} \quad (1)$$

Completing $Quota_k$ instructions within one epoch is accomplished by all SMs. The QoS manager calculates this quota for each kernel and distributes it to all SMs proportionally to the TBs hosted by each SM. For example, if k has T TBs in total, and SM_i is hosting T_i of them, then the local quota ($quota_k$) of SM_i is $Quota_k \times \frac{T_i}{T}$. Hence, $Quota_k$ is distributed into each SM in a balanced way, which will lead to better utilization of TLP.

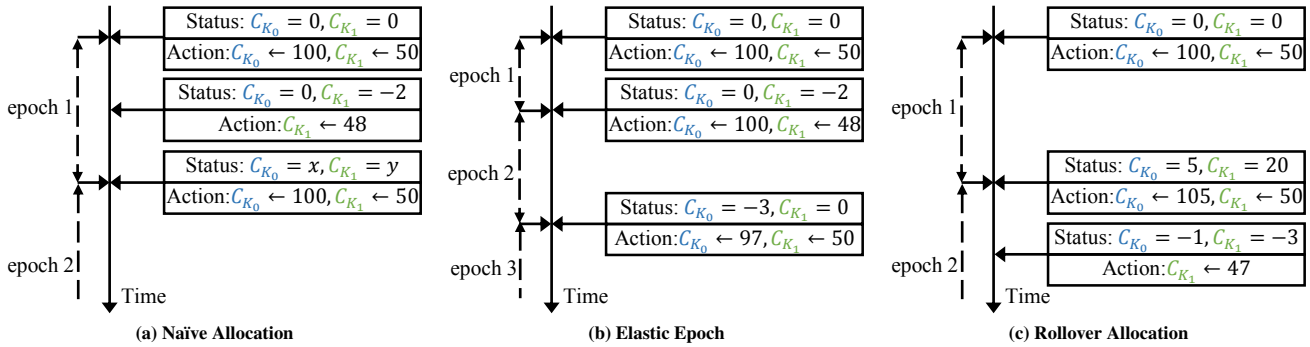


Figure 4: Overview of quota allocation schemes. K_0 is a QoS kernel, K_1 is an non-QoS kernel, and C_{K_i} is the quota counter for kernel K_i.

$Quota_k$ is allocated at the beginning of an epoch. Suppose there is a local counter C_k for storing $quota_k$. When a warp instruction of k is completed, C_k is decremented by the number of instructions that are actually executed in the warp instruction (≤ 32 due to branch divergence). If a portion of the quota (C_k is positive) is unused at the end of an epoch, then naïve allocation simply discards the excess and resets C_k to $quota_k$ for the next epoch.

If C_k drops to zero or negative before the end of an epoch, it means that the SM is capable of executing more instructions. However, the QoS kernel k has reached its goal in the current epoch. It is thus reasonable to allocate the remaining (unused) cycles to non-QoS kernels. We still enforce certain quotas on non-QoS kernels to prevent them from over-taking cycles from QoS kernels. Their quota calculation is discussed in Section 3.5. The naïve scheme checks if C_ks for *all* ks in the SM are zero or negative, to ensure all QoS kernels have exhausted their quotas. If so, some quantity, e.g., $quota_k$, is added to C_k for all non-QoS kernels to keep them running until the end of the current epoch. For QoS kernels, once the IPC goal has been reached, no further quota is given. At the beginning of the next epoch, C_k is reset to $quota_k$ again for all k.

Figure 4a shows an example of the naïve allocation scheme. The quotas of K_0 (100), a QoS kernel, and K_1, a non-QoS kernel (50), are allocated at the beginning of epoch 1. Before finishing the epoch, both kernels exhaust their quotas: C_{k_0} is 0 and C_{k_1} is over-consumed as it is decremented *after* a warp of 32 instructions (or fewer due to branch divergence) is finished. C_{k_1} is then reallocated by adding 50, but C_{k_0} is not. At the end of epoch 1, also the beginning of epoch 2, quotas are reallocated to both kernels, and unused quotas are discarded.

3.4.2 History-based Quota Adjustment. The naïve scheme has the clear limitation that it does not consider the variance of a kernel. There could be epochs where the QoS kernel cannot reach its goal due to performance fluctuation. In other words, $Quota_k$ is calculated according to an *average* IPC_{goal} for the entire duration of the kernel. Some epochs may not be able to achieve a local goal, but other epochs that could exceed it are capped at their local goals. Hence, the resulting IPC of each epoch is never above IPC_{goal} of k, leading to failure to meet QoS in the end.

To address this problem, we could raise $Quota_k$ slightly, so that the final IPC of a kernel, which is likely to be under what $Quota_k$ can produce, could reach IPC_{goal}. The increase in $Quota_k$ is determined by the IPC achieved in all past epochs ($IPC_{history}$).

$$Quota_k = \alpha_k \times IPC_{goal} \times T_{epoch}$$

where,

$$\alpha_k = \max\{\frac{IPC_{goal}\ of\ k}{IPC_{history}\ of\ k}, 1\}$$

α_k is calculated at the beginning of each epoch. Hence, if $IPC_{history}$ is below IPC_{goal}, $Quota_k$ is scaled up. For example, if the IPC_{goal} of kernel k is 125 and the average IPC from the beginning to the current epoch is 100, α_k will be 1.25 and the allocated quota of k will scale up by 1.25. Giving more quota to k means that it will have more dynamic resources to catch up and meet IPC_{goal}.

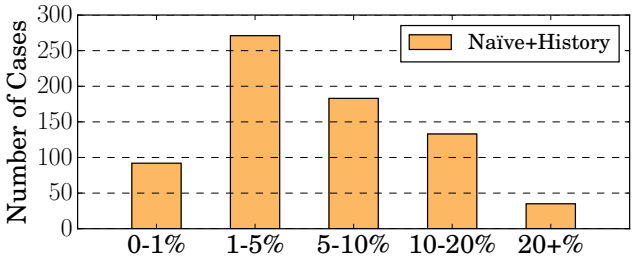


Figure 5: The number of cases (out of 900) that Naïve Allocation with History-based Adjustment misses the IPC_{goal} vs. how much it misses the goal.

Figure 5 shows the number of cases that history-based quota adjustment misses (undershoots) QoS goals, out of the total number of cases tested (900 as discussed in evaluation). We put the failure cases into categories by how much they miss the QoS goals. We can see that the total number of missed cases is over 700, even with history-based quota adjustment, and most of them are only within 5% of the QoS goals. The successful cases (186 cases), which reach the QoS goals, overshoot by 1.3% on average. We observe that better decisions can be made on how new quotas are allocated when current

quotas are consumed fast (or slowly). We develop two alternative schemes, "Elastic Epoch" and "Rollover Allocation", to address those two scenarios.

3.4.3 Elastic Epoch. Figure 5 indicates that many QoS kernels still miss their performance goals, but not by too much. However, we experimented with more aggressive α adjustment and found that the results would benefit QoS kernels but not the non-QoS kernels so that the total throughput is lowered.

Instead of allocating quotas only at the beginning of each epoch, Elastic Epoch makes the epoch length flexible. Once *all* kernels consume their quota before the end of an epoch, a new epoch immediately starts. As shown in Figure 4b, the quotas of K_0 and K_1 are allocated at the beginning of their execution. When all quotas are used up before epoch 1 ends (C_k's are zero or negative), the C_k values are added to the new quota values as if epoch 2 has started.

3.4.4 Rollover Allocation. If quotas are consumed too slowly, unused quotas of QoS kernels indicate that they did not reach their performance goal in the last epoch. Hence, these kernels should be given more resources to catch up in future epochs. A simple change is to keep the unused quota of QoS kernels, rather than discarding it.

We develop "Rollover Allocation" to keep unused quotas for QoS kernels. When allocating quotas in the next epoch, the unused quota of QoS kernels from the last epoch are added to the quota of this epoch. The quotas of non-QoS kernels are discarded as usual. In this way, QoS kernels have more dynamic resources in the next epoch to make up for their performance loss in the previous epoch. Take the example in Figure 4c, the counters for K_0 and K_1 are adjusted at the beginning of epoch 2. At that time, there are unused quota of K_0, and it is kept because K_0 is a QoS kernel. The unused quota of K_1 is discarded because it is a non-QoS kernel.

The elastic epoch scheme uses a variable epoch length. Its effectiveness is related to how often quotas are consumed faster than the duration of one epoch. The rollover allocation scheme uses fixed epoch length. Its effectiveness is related to the potential of the rolled over quota to regain the performance of QoS kernels. The proposed schemes adjust the quota allocation of the future epochs based on the result of previous epochs to mitigate the unsatisfied performance of QoS kernels within an epoch during execution. Our evaluation shows that both schemes can achieve QoS goals much more often than naïve allocation with history-based quota adjustment and previously proposed designs.

3.5 Managing Non-QoS Kernels

The quota allocated to non-QoS kernels cannot be derived in the same way as QoS kernels because non-QoS kernels do not have QoS requirements. Not allocating, or allocating a very small quota will make the sharing fall back to time-multiplexed execution. The purpose of finding proper quotas for non-QoS kernels is to ensure good progress of QoS kernels while utilizing execution cycles to achieve the best throughput for non-QoS kernels.

We develop a simple scheme to search for a proper quota for a non-QoS kernel. The search procedure relies on how well QoS kernels are achieving their performance goals. If the cumulative performance of QoS kernels in the previous epoch (IPC_{epoch}) are well above their QoS goals, then a non-QoS kernel can be given higher quotas. Otherwise, lower quotas will be used. Quotas are updated at the beginning of each epoch, using profiled information from past epochs. A quota of a non-QoS kernel is calculated from an artificial performance goal as the following:

$$IPC_{goal} = IPC_{epoch} \times \max_{\forall k \in QoS\ kernels} \frac{IPC_{epoch}\ of\ k}{\alpha_k \times IPC_{goal}\ of\ k}$$

Once the IPC_{goal} of a non-QoS kernel is calculated, its quota can be computed from equation (1). The IPC_{epoch} of the non-QoS kernel is initially set to a conservatively small value (keeping the initial quota small). The initial IPC_{epoch} is 1 in our evaluation. QoS kernels can then benefit and achieve their QoS goals in early epochs. To avoid resource under-utilization, the IPC_{goal} of the non-QoS kernel will increase, but it will not be high enough to threaten the QoS kernels. α_k is used in this equation to take the history-based adjustment into consideration when limiting the performance of non-QoS kernels. From our observation, the initial value of IPC_{epoch} has minimal impact on the final outcome.

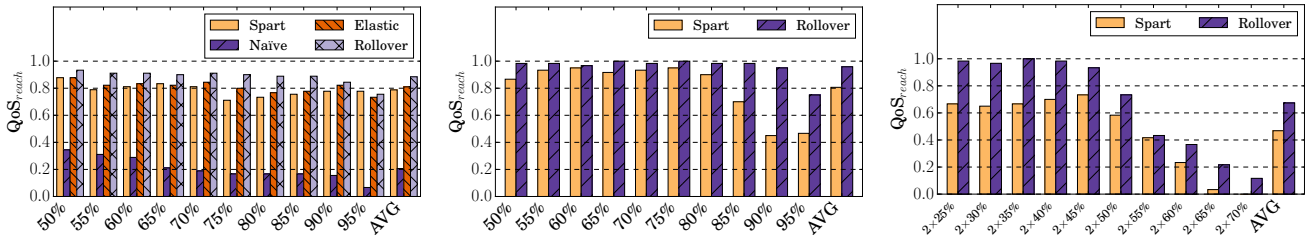
Note, the IPC_{goal} of the non-QoS kernel dynamically changes during execution. As studied in [33], a kernel can behave differently during execution, and the same IPC goal for non-QoS kernels may have different impact on the performance of QoS kernels. If non-QoS kernels take too much resources (e.g. compute core cycle or memory bandwidth) and the IPC of the QoS kernel is lower than its goal, the IPC_{goal} of the non-QoS kernel will be scaled down to leave more resources to the QoS kernel. Hence, our design dynamically limits the performance of non-QoS kernels when QoS kernels require more resources, and lifts the limit when QoS kernels require fewer resources. As a result, our scheme works for both regular and irregular kernels.

3.6 Static Resource Allocation and Adjustment

Static resources (registers, shared memory, etc.), as the number of TBs dispatched from each kernel, are also managed. This is because QoS goals for some QoS kernels are hard to reach due to the lack of TLP and having more TBs will increase the kernel's TLP.

Symmetric TB allocation for two or more kernels. Initially, QoS kernels are evenly distributed to every SM so that every SM has the same number of threads to balance the TLP as a starting point. For non-QoS kernels, previous studies [41, 42] show that having too many kernels within one SM may not always be beneficial. Hence, we partition SMs to non-QoS kernels equally. Each kernel is then symmetrically dispatched to its own partition of SMs. For example, consider one QoS kernel and two non-QoS kernels on a GPU with 16 SMs. The QoS kernel will run on 16 SMs and each non-QoS kernel will run on 8 SMs. Within each SM, an equal number of threads are assigned to the kernels on that SM. This allocation serves as a baseline for further static resource management, which is performed with partial context switching [42].

Run-time adjustment. At run time, the execution of kernels is monitored to determine if their TB allocation should change. During each epoch, we sample the number of idle warps (IW) for all kernels. IWs have ready instructions but are not scheduled to execute due to a full pipeline, which typically indicates that a kernel has excessive TLP [33]. IWs occupy static resources, but do not contribute to the progress of the kernel. Hence, a lower TLP can probably achieve the



(a) Two-kernel pairs. (b) One QoS kernel in the trio. (c) Two QoS kernels in the trio.

Figure 6: QoS$_{reach}$ vs. QoS goals. Each bar is averaged over all 90 pairs (or 60 trios) of benchmarks.

same performance. However, a kernel with few IWs can utilize all its TLP and additional warps may further improve performance.

At the beginning of an epoch, the number of IWs is collected for each kernel in each SM. If the number of IWs equals the number of warps per TB, then swapping out one TB has the same TLP as swapping out those IWs. We call these TBs "idle TBs". If for a QoS kernel, the number of idle TBs is no more than one and $IPC_{history}$ has not achieved its goal, then one more TB will be allocated to increase TLP. TBs of a victim kernel will be swapped out, if more resources are needed. The victim kernel is chosen if one of the following conditions is satisfied:

- It is a non-QoS kernel
- The kernel has at least $n+1$ idle TBs if n of them are needed to vacate enough resources.
- The IPC$_{history}$ of the kernel is so high that $IPC_{history} \times \left(1 - \frac{n}{N}\right) > IPC_{goal}$, where N is the total number of TBs of this kernel.

Hence, either a non-QoS kernel, or a QoS kernel with excessive TLP and enough IPC margin to lose will be selected as a victim kernel. Lastly, to limit the overhead of preemption, swapping only happens if there are no pending preemption requests from any kernel.

4 EXPERIMENTAL EVALUATION

4.1 Methodology

GPU Param.	Value	SM Param.	Value
Core Freq.	1216MHz	Registers	256KB
Mem. Freq.	7GHz	Shared Memory	96KB
# of SMs	16	Threads	2048
# of MC	4	TB Limit	32
Sched. Policy	GTO	Warp Scheduler	4

Table 1: Simulation parameters.

Simulator. To evaluate our designs, we use the latest version of GPGPU-Sim [4], with the simulation parameters in Table 1. These parameters are close to those used in previous work [41, 42]. We modified GPGPU-Sim to support spatial partitioning and fine-grained sharing (e.g., SMK), and follow the same assumptions and implementation reported in previous work [3, 37, 41].

Co-run Benchmarks; Scalability. We use 10 benchmarks from the Parboil benchmark set [36]. *bfs* is not used because it is too small to interfere with any sharer kernels in our setting. We used the largest datasets for all kernels. We experimented with sharing the GPU by two and three kernels to measure the scalability of our designs. To co-run two kernels, $10 \times 9 = 90$ pairs of kernels are generated: one is a QoS kernel and the other is a non-QoS kernel. To co-run three kernels, 60 trios of all possible combinations were tested due to the excessive number of runs. Either one or two kernels of the trio are QoS and the remaining one(s) are non-QoS kernels.

We ran 2M cycles since according to [1], the results are accurate when the simulation is longer than 1M cycles. If one program ends before 2M cycles, it is re-executed. If the benchmark has multiple kernels or the kernel is executed multiple times, we use the total number of instructions and cycles from the benchmark to calculate the IPC. The epoch length is 10K cycles, which is determined empirically as a past study [17] showed that the same epoch length is sufficiently good. We sample 100 times per epoch for the number of idle warps and use the average for TB adjustment.

Metric. To evaluate QoS compliance, we use *the percentage of QoS goals that are reached* (QoS$_{reach}$) as our metric in comparing each management scheme: Spatial Partition with hill climbing [3] (**Spart**), Naïve Quota Allocation (`Naïve`), Elastic Epoch (`Elastic`) and Rollover Quota Allocation (`Rollover`). The QoS$_{reach}$ is defined as $\frac{\#\ of\ Success\ Cases}{\#\ of\ Total\ Cases}$. As explained in Section 3.2, the QoS goal is set as a percentage of IPC$_{isolated}$, ranging from 50% to 95%, with a 5% step size. In co-running two/three kernels, 90 pairs/60 trios and 10 QoS goals generate 900/600 test cases. For 2-QoS-kernel cases, we sweep the QoS goals from (25%, 25%) to (70%, 70%) with a (5%, 5%) step size. When calculating the throughput of kernels, only the cases that meet the QoS goal are included. Higher QoS$_{reach}$ means a design can reach more QoS goals. It is similar to the multiple QoS requirements in [3], but with a larger and more demanding set of QoS goals.

4.2 QoS$_{reach}$ Comparison

Figure 6 shows QoS$_{reach}$ for different QoS schemes when co-running two kernels (Figure 6a) and three kernels (Figures 6b and 6c). When co-running two kernels, `Naïve` has the lowest QoS$_{reach}$ (20.6%) due to inefficient use of quotas, while `Rollover` has the best result (88.4%). `Spart` covers 78.8% of the 900 cases. `Rollover` is better than `Spart` in almost all cases, with an average of 12.2% improvement over Spart in QoS$_{reach}$.

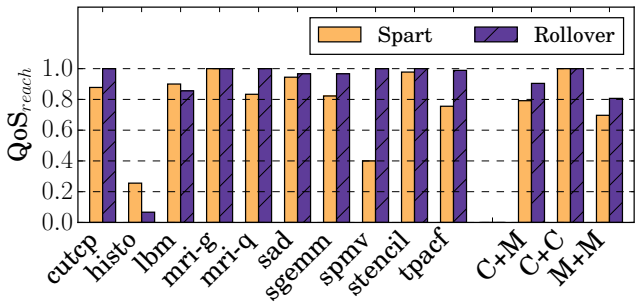


Figure 7: QoS$_{reach}$ vs. QoS kernel in two-kernel sharing.

`Elastic` and `Rollover` are significantly better than `Naïve` because they overcome major limitations of `Naïve`. `Rollover` is also better than `Elastic` because `Rollover` helps QoS kernels directly when they did not reach their performance goals (having unused quota at the end of an epoch). `Elastic` did not target this problem directly. Instead, it gives QoS kernels more opportunities to perform better when they are already doing well (using up quotas fast).

Figures 6b and 6c show the QoS$_{reach}$ for one and two QoS kernels per trio respectively. We compare `Spart` only with `Rollover` since it is the best among our proposed schemes. Both figures show that `Rollover` reaches QoS goals more often than `Spart` by 18.8% (6b) and 43.8% (6c). `Rollover` has higher improvement over `Spart` when there are more sharers and more QoS kernels, i.e., QoS requirement is higher, because `Spart` cannot fully utilize the GPU. For example, `Spart` performs poorly for cases over (60%, 60%) in Figure 6c, and failed to reach any QoS goal in the (70%, 70%) cases. As we can see, fine-grained QoS can control multiple resources better, especially the execution cycles among sharer kernels, while coarse-grained sharing and QoS control such as `Spart` has only one knob to turn (the number of SMs) which becomes limited when there are more QoS requirements. Hence, `Rollover` has better scalability than `Spart`.

We report in Figure 7 the QoS$_{reach}$ of `Rollover` and `Spart` with respect to each QoS kernel. Each bar is averaged over 9 pairs with 10 QoS goals. The summary results for pairing compute- and compute-intensive ("C+C"), compute- and memory-intensive ("C+M"), and memory- and memory-intensive ("M+M") kernels are also shown. We find for C+C kernels, `Spart` and `Rollover` meet QoS goals in all cases. However, for M+M kernels, `Spart` does worse than `Rollover` because QoS for `Spart` does not have a mechanism to control memory bandwidth. Although it does not directly manage bandwidth, `Rollover` throttles instructions by applying quotas. This control effectively reduces memory traffic once quotas are used up, which mitigates contention among sharer kernels. The same principle applies to the C+M kernels where `Rollover` also outperforms `Spart`.

`Rollover` achieves 100% of the QoS goals for six benchmarks. For *histo*, neither scheme performs well due to the short running nature of this benchmark's kernels. `Rollover` has profiling overhead in each epoch while (more static) `Spart` does not. Alternatively, a kernel-level QoS scheduler may also perform well since kernels are short.

4.3 Throughput of Non-QoS Kernels

Figure 8 shows the throughput normalized to the isolated execution for the non-QoS kernels in two-kernel and three-kernel sharing cases. We only include the results from the cases that meet the QoS goals. From these results, we observe:

(1) The performance of the non-QoS kernels decreases as the QoS goal increases.
(2) `Rollover` has higher throughput than `Spart` in *all* cases. The improvement increases as QoS requirements increase. In two-kernel sharing, the average improvement is 15.9%. This number increases to 19.9% and 20.5% in three-kernel sharing with one (Figure 8b) and two (Figure 8c) QoS kernels, respectively.
(3) The improvement also increases along the x-axis (higher QoS goals). As an example, the largest improvement of 75.5% in Figure 8b occurs in the 95% category. In Figure 8c, a $> 10\times$ improvement is observed in the last three categories.

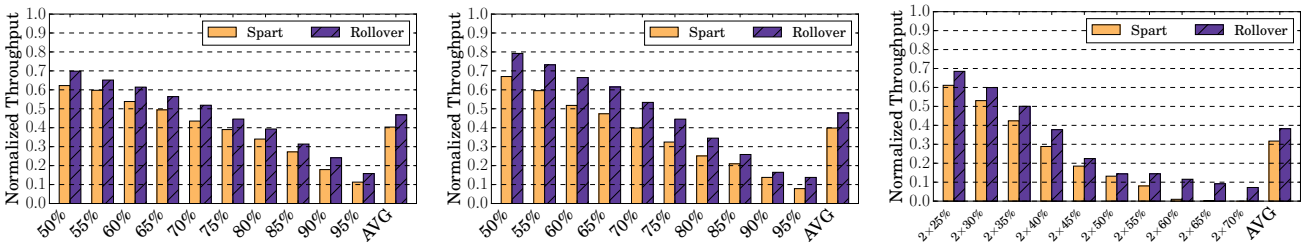
The results demonstrate that fine-grained resource allocation is superior to coarse-grained resource allocation. `Spart` makes it difficult to give resources to the non-QoS kernel if the QoS kernels need less than the resources of one complete SM. That is, an SM cannot be divided between QoS and non-QoS kernels. Whereas `Rollover` can allocate just enough resources to the QoS kernels, which leaves all the remaining resources to non-QoS kernels. We also expect that the benefit of `Rollover` over `Spart` will be more obvious as the number of sharers and number of QoS kernels increase.

4.4 Throughput of QoS Kernels

For QoS kernels, the goal is to achieve their QoS goals but not by too much, leaving more resources for non-QoS kernels. Figure 9 shows the actual throughput of QoS kernels, normalized to their QoS goals. As we can see, `Spart` exceeds the given goal by 11.6% on average, greatly reducing the resources for the non-QoS kernel, because the QoS kernel uses more resources than it actually needs. On the contrary, `Rollover` only exceeds the goal by 2.8%. Such effectiveness mainly comes from its capability of fine-grained control of resources, such as which kernel's instruction to execute on a per-cycle basis. Whereas in `Spart`, the granularity of QoS management is one SM which has large resources which are indivisible, even when a QoS kernel only needs a portion of the SM.

4.5 QoS via Prioritizing Kernels

As explained in Section 3.3, conventional QoS with prioritization as in CPUs is not suitable for GPUs. Figure 10 and 11 compare `Rollover` and a time-multiplexed warp scheduling for `Rollover` (`Rollover-Time`) that blocks non-QoS kernels until the QoS kernels finish their quotas, in QoS$_{reach}$ and non-QoS kernel's throughput. As we can see, both schemes have similar QoS$_{reach}$ with a difference of only 3% on average, indicating they have similar capability of achieving QoS goals. However, `Rollover-Time` degraded performance of non-QoS kernels by 1.47X (Figure 11). Hence, allowing the overlapped execution of different kernels with `Rollover` benefits throughput because kernels may have complementary behavior.



(a) Two-kernel pairs. (b) One QoS kernel in the trio. (c) Two QoS kernels in the trio.

Figure 8: Throughput normalized to isolated execution of non-QoS kernels. The x-axis is QoS goals.

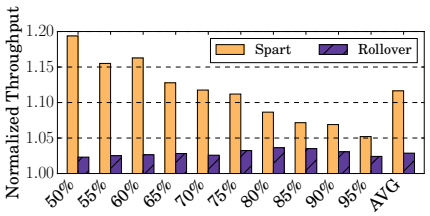


Figure 9: Actual throughput of QoS kernels, normalized to their QoS goals.

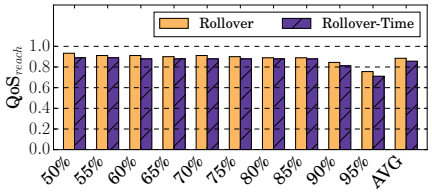


Figure 10: QoS$_{reach}$ of QoS kernels.

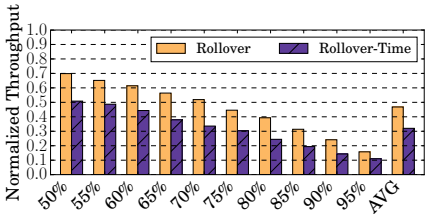


Figure 11: Throughput, normalized to the isolated execution, for non-QoS kernels.

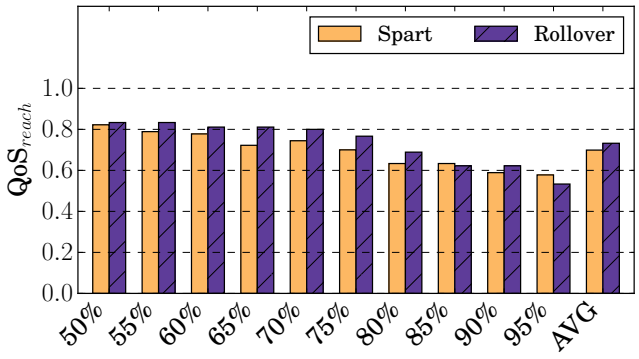


Figure 12: QoS$_{reach}$ vs. QoS goals for 56 SMs. Each bar is averaged over all 90 pairs of benchmarks.

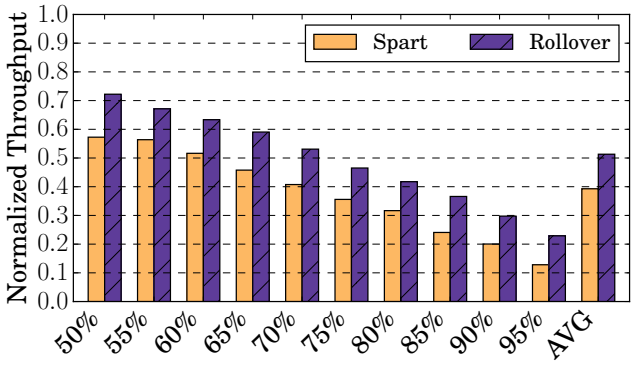


Figure 13: Throughput normalized to isolated execution of non-QoS kernels for 56 SMs. The x-axis is QoS goals.

4.6 Scalability with Number of SMs

Recent GPU architecture design has introduced more SMs in a GPU [28]. To further test the scalability of our design with the number of SMs, we simulate a GPU with 56 SMs, each SM having two warp schedulers. Other simulation parameters are the same as the ones in Table 1. Because the new parameters lead to new isolated performance of each kernel and QoS goals, the results here cannot be directly compared with the results in previous sections.

Figure 12 and 13 show the QoS$_{reach}$ and normalized throughput in `Spart` and `Rollover`. As shown in the figure, having more SMs improves QoS$_{reach}$ for Spart because it can now allocate resources at a finer granularity, but the average is still 4.76% short of `Rollover`. Also, in the throughput of non-QoS kernels, `Rollover` is much better than Spart, achieving an improvement of 30.65% on average. This shows that `Rollover` has good QoS$_{reach}$ and improves the utilization regardless of the number of SMs.

4.7 Power Efficiency

Finally, we expect that fine-grained sharing and QoS management for GPUs will achieve better power efficiency due to better utilization and management of resources. To see this effect, we used GPUWattch [21], a GPU power model embedded in GPGPU-Sim, to measure the power consumption of different QoS schemes. Figure 14 shows the improvement of instructions per watt for `Rollover` over `Spart`, in the two-kernel sharing scenario. As shown in the figure, `Rollover` improves power efficiency by 9.3% on average,

	CPU QoS	Kernel Fusion[39]	SMK[42]	Spatial QoS[3]	Warped-Slicer[44]	Baymax[8]	Fine-grained QoS
Software/Hardware	S	S	H	H	H	S	H
QoS Awareness	✓			✓		✓	✓
Work on GPUs		✓	✓	✓	✓	✓	✓
Preemption	✓		✓	✓			✓
Active GPU Sharing		✓	✓	✓	✓		✓
Sharing within SMs		✓	✓		✓		✓
Fine Perf. Control	✓						✓
Adaptive TLP					✓		✓

Table 2: Comparison between fine-grained QoS and prior work.

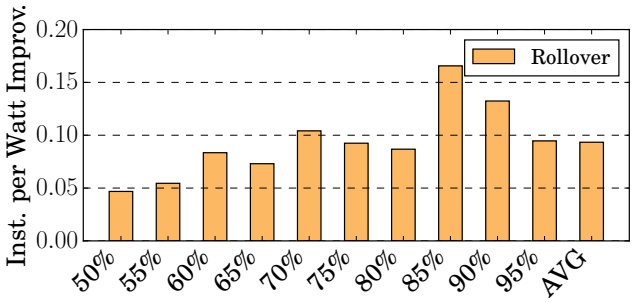


Figure 14: Energy efficiency improvement over `Spart`.

due to better resource utilization. Since our design has better performance and power efficiency compared to coarse-grained sharing at the same number of SMs, our design can possibly use fewer SMs to achieve the same performance as coarse-grained sharing, creating opportunities for power gating.

4.8 Preemption Overhead and Other Results

In the interest of space, we summarize our evaluations on preemption overhead, the effect of history-based quota adjustment, and the effect of static resource management. Overall, the preemption overhead is 1.93% on the throughput of non-QoS kernels, as most of the memory operations are overlapped with the execution of other non-preempted TBs. When disabling history-based quota adjustment, QoS_{reach} drops significantly for all categories. Overall, enabling it covers 86.4% more cases than disabling it. The results clearly show that having adjustment results in much fewer QoS goal misses than not having the adjustment. Static resource management also has positive effect on performance. Enabling it improves the throughput of non-QoS kernels in the M+M combination by 13.3% due to a better balancing of kernel TLP by TB re-allocation.

4.9 Hardware Overhead

Our design mainly adds logic overhead to the fine-grained sharing mechanism such as SMK [41, 42]. As illustrated in Figure 3, the additional logic needed are: (1) the QoS Manager which collects statistics for each kernel in each SM, updates quotas, and decides adjustment of TB allocation per kernel; and (2) the enhanced warp scheduler which incorporate quotas. The quota schemes require counting instructions every cycle, similar to existing performance counters. The quotas are calculated once per epoch (10K cycles in our evaluation), and they are not on the critical path of execution. The calculation can be done through a specialized thread or a dedicated ALU identical to what a GPU already has. Hence, the additional logic has minimal impact on the performance. Registers for saving quotas, epoch, and counters for counting instructions are necessary, which are already provided in SMK. A new bit vector is needed to flag QoS kernels as well. Overall, we consider the hardware overhead of having QoS management rather modest.

5 RELATED WORK

Many research on QoS have been done for CPUs where processes can be preempted, and time-multiplexed without degrading performance [7, 10, 12, 35, 46], or for networks where packets can be throttled [9, 34]. Sharing GPUs by multiple kernels has become necessary to achieve better resource utilization and overall throughput [30, 37]. Studies have been performed on the benefits of sharing and how resources should be allocated among sharers [1, 44]. However, QoS management for sharers has been very limited primarily due to the lack of hardware support. Due to the drastic differences in execution model and architecture between GPUs and CPUs, applying QoS designs for CPUs to GPUs results in inferior performance, as shown in our experiments.

At high level, several system level solutions have been proposed for QoS on GPUs. TimeGraph [18] manages fairness by ordering the commands in the command queue of a GPU. Fairness can also be managed by trapping and holding the commands from demanding programs, through the MMIOs of the GPU driver [25]. Both solutions require modifications or reverse engineering the proprietary GPU driver [24] which is difficult. Baymax [8] manages QoS by predicting the execution time of a kernel, and schedules kernels to leave enough time for all QoS kernels. Mystic [38] used machine learning to predict whether kernels can share a GPU efficiently, and distribute kernels in a cluster. All those designs are orthogonal to our work. They can utilize our proposed mechanism to have more control on the execution of kernels. Also, they work at the granularity of kernels, and cannot handle long-running kernels well. Due to the lack of preemption support, an already running kernel cannot be preempted, and it may occupy the GPU for a long time, blocking others requests.

One way to workaround this problem is to allow multiple kernels to run on GPU concurrently, using software solutions. Kernel fusion [39] and KernelMerge [13] statically merge code from two kernels into one, using conditional statements to separate the execution paths. Elastic kernel [30] also implements this method, but focuses

on the benefit of running kernels concurrently in the same SM. Lee et al. [20] also explores this benefit briefly. Changing the resource allocation to a certain extent at run time was also attempted [23, 43] In those approaches, kernel resource allocation is determined at compile time, so no new kernels can be launched at run time and no hot swapping among kernels can be done. Also, source code must be available for concurrent execution and recompilation, which is often not possible.

To enable true sharing at run time, architectural support for preemption have been proposed [31, 37, 41, 42]. As discussed in Section 2.3, those approaches by themselves do not provide effective solutions for QoS control.

Aguilera et al. developed a profiling-based QoS strategy that divides SMs among sharer kernels [3]. A linear model between performance and number of SMs was used for performance prediction. However, the model heavily depends on the sharer kernels, because they compete for memory bandwidth which is not partitioned among SMs. Also, the performance tuning granularity is SMs, which is overly coarse as we studied. Table 2 summarizes the main novelties of our proposed techniques with prior GPU sharing and QoS techniques.

Performance estimation is important since it requires to convert application requirements to architectural metrics. For 3D rendering applications, different operations mapped to different rendering APIs have similar cost. Thus, the kernels can be grouped and estimated from history information. GERM [5], TimeGraph [18] and VGRIS [45] all adopt this approach to provide QoS control. However, general purpose applications have drastically different execution time so previous approaches cannot be directly applied here. Each TB in the same kernel has similar number of instructions [29], which can be used to estimate the progress of a kernel. Different from previous work, we translate kernel rate to IPC, and enforce the IPC requirement in hardware.

6 CONCLUSIONS

State-of-the-art GPUs do not have proper support for QoS management which is critical when multiple applications share a GPU in modern systems. We found that coarse-grained management at the SM level is insufficient and does not scale when the number of sharer kernels is increased. We describe a novel design that enables fine-grained QoS control for multiple kernels. This mechanism allows kernels to satisfy QoS goals with just enough resources, leaving the remaining resources for kernels without QoS goals to execute at high instruction throughput. Moreover, we propose a method to translate application QoS requirements to architecture metrics. Evaluation results show that our approaches lead to significant improvement in reaching QoS goals and improving power efficiency versus coarse-grained QoS management.

7 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive feedback. This work is partially sponsored by the National Basic Research 973 Program of China (No. 2015CB352403), the Scientific Innovation Act of STCSM (No. 13511504200), and the EU FP7 CLIMBER project (No. PIRSES-GA-2012-318939). This work is also supported in part by NSF grants CNS-1012070, CNS-1305220, CCF-1617071, CCF-1422331 and CCF-1535755. This work was carried out while Zhenning Wang visited the University of Pittsburgh on a CSC scholarship.

REFERENCES

[1] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. 2012. The case for GPGPU spatial multitasking. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. 1–12.

[2] Paula Aguilera, Katherine Morrow, and Nam Sung Kim. 2014. Fair share: Allocation of GPU resources for both performance and fairness. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*. 440–447. https://doi.org/10.1109/ICCD.2014.6974717

[3] Paula Aguilera, Katherine Morrow, and Nam Sung Kim. 2014. QoS-aware dynamic resource allocation for spatial-multitasking GPUs. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*. 726–731. https://doi.org/10.1109/ASPDAC.2014.6742976

[4] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. 163–174.

[5] Mikhail Bautin, Ashok Dwarakinath, and Tzi-cker Chiueh. 2008. Graphic engine resource management. *Proc. SPIE* 6818 (2008), 68180O–68180O–12. https://doi.org/10.1117/12.775144

[6] Thomas Bradley. 2012. Hyper-Q example. (2012).

[7] Abhishek Chandra, Micah Adler, Pawan Goyal, and Prashant Shenoy. 2000. Surplus Fair Scheduling: A Proportional-share CPU Scheduling Algorithm for Symmetric Multiprocessors. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4 (OSDI'00)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1251229.1251233

[8] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 681–696. https://doi.org/10.1145/2872362.2872368

[9] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *Symposium Proceedings on Communications Architectures &Amp; Protocols (SIGCOMM '89)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/75246.75248

[10] Kenneth J. Duda and David R. Cheriton. 1999. Borrowed-virtual-time (BVT) Scheduling: Supporting Latency-sensitive Threads in a General-purpose Scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*. ACM, New York, NY, USA, 261–276. https://doi.org/10.1145/319151.319169

[11] HSA Foundation. 2015. HSA Platform System Architecture Specification. (2015).

[12] Pawan Goyal, Xingang Guo, and Harrick M. Vin. 1996. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the 2nd USENIX Conference on Operating Systems Design and Implementation (OSDI '96)*. USENIX Association.

[13] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. 2012. Fine-grained resource sharing for concurrent GPGPU kernels. In *4th USENIX Workshop on Hot Topics in Parallelism (HotPar). Berkeley, CA*.

[14] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. 2015. DjiNN and Tonic: DNN As a Service and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 27–40. https://doi.org/10.1145/2749469.2749472

[15] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. 2008. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, 260–269.

[16] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 395–406. https://doi.org/10.1145/2451116.2451158

[17] Adwait Jog, Onur Kayiran, Ashutosh Pattnaik, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2016. Exploiting Core Criticality for Enhanced GPU Performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS '16)*. ACM, New York, NY, USA, 351–363. https://doi.org/10.1145/2896377.2901468

[18] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*. 17–30.

[19] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. 2014. Managing GPU Concurrency in Heterogeneous Architectures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 114–126. https://doi.org/10.1109/MICRO.2014.62

[20] Minseok Lee, Seokwoo Song, Joosik Moon, J. Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. 260–271. https://doi.org/10.1109/HPCA.2014.6835937

[21] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 487–498. https://doi.org/10.1145/2485922.2485964

[22] Jiwei Liu, Jun Yang, and Rami Melhem. 2015. SAWS: Synchronization Aware GPGPU Warp Scheduling for Multiple Independent Warp Schedulers. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*.

[23] Christos Margiolas and Michael F. P. O'Boyle. 2016. Portable and Transparent Software Managed Scheduling on Accelerators for Fair Resource Sharing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO 2016)*. ACM, New York, NY, USA, 82–93. https://doi.org/10.1145/2854038.2854040

[24] Konstantinos Menychtas, Kai Shen, and Michael L Scott. 2013. Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack.. In *USENIX Annual Technical Conference*. 291–296.

[25] Konstantinos Menychtas, Kai Shen, and Michael L. Scott. 2014. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, 301–316.

[26] NVIDIA. 2012. Sharing a GPU between MPI processes: multi-process service(MPS). (2012).

[27] Nvidia. 2014. Programming Guide. (2014).

[28] NVIDIA. 2016. GP100 Pascal Whitepaper. (2016). https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[29] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. 2014. Preemptive Thread Block Scheduling with Online Structural Runtime Prediction for Concurrent GPGPU Kernels. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 483–484. https://doi.org/10.1145/2628071.2628117

[30] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU Concurrency with Elastic Kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, 407–418.

[31] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 593–606.

[32] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, 233–248.

[33] Ankit Sethia and Scott Mahlke. 2014. Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 647–658. https://doi.org/10.1109/MICRO.2014.16

[34] Madhavapeddi Shreedhar and George Varghese. 1995. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '95)*. ACM, New York, NY, USA, 231–242. https://doi.org/10.1145/217382.217453

[35] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K Baruah, Johannes E Gehrke, and C Greg Plaxton. 1996. A Proportional Share Resource Allocation Algorithm for Real-time, Time-shared Systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*. IEEE Computer Society, Washington, DC, USA, 288–. http://dl.acm.org/citation.cfm?id=827268.828976

[36] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-M Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* (2012).

[37] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling Preemptive Multiprogramming on GPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, 193–204.

[38] Yash Ukidave, Xiangyu Li, and David Kaeli. 2016. Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 353–362. https://doi.org/10.1109/IPDPS.2016.73

[39] Guibin Wang, Yisong Lin, and Wei Yi. 2010. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*. 344–350.

[40] Minjie Wang, Tianjun Xiao, Jianpeng Li, Jiaxing Zhang, Chuntao Hong, and Zheng Zhang. 2014. Minerva: A scalable and highly efficient training platform for deep learning. (2014).

[41] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2015. Simultaneous Multikernel: Fine-grained Sharing of GPGPUs. *Computer Architecture Letters* PP, 99 (2015), 1–1. https://doi.org/10.1109/LCA.2015.2477405

[42] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-Grained Sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 358–369. https://doi.org/10.1109/HPCA.2016.7446078

[43] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations. In *ICS' 15*.

[44] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *Proceeding of the 43st Annual International Symposium on Computer Architecuture (ISCA '16)*. IEEE Press.

[45] Miao Yu, Chao Zhang, Zhengwei Qi, Jianguo Yao, Yin Wang, and Haibing Guan. 2013. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing (HPDC '13)*. ACM, New York, NY, USA, 203–214. https://doi.org/10.1145/2462902.2462914

[46] Wanghong Yuan and Klara Nahrstedt. 2003. Energy-efficient Soft Real-time CPU Scheduling for Mobile Multimedia Systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 149–163. https://doi.org/10.1145/945445.945460

[47] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *Parallel and Distributed Systems, IEEE Transactions on* 25, 6 (June 2014), 1543–1552. https://doi.org/10.1109/TPDS.2013.111