Parallel Computing 113 (2022) 102958

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

QoS-aware dynamic resource allocation with improved utilization and energy efficiency on GPU

Qingxiao Sun [a,b], Liu Yi [b], Hailong Yang [a,b,*], Mingzhen Li [b], Zhongzhi Luan [b], Depei Qian [b]

[a] State Key Laboratory of Software Development Environment, Beijing, China
[b] School of Computer Science and Engineering, Beihang University, Beijing, China

ARTICLE INFO

Keywords:
Graphics processing units
Quality of service
Dynamic resource management
Throughput
Power efficiency

ABSTRACT

Although GPUs have been indispensable in data centers, meeting the Quality of Service (QoS) under task consolidation on GPU is extremely challenging. Previous works mostly rely on the static task or resource scheduling and cannot handle the QoS violation during runtime. In addition, existing works fail to exploit the computing characteristics of batch tasks, and thus waste the opportunities to reduce power consumption while improving GPU utilization. To address the above problems, we propose a new runtime mechanism *SMQoS* that can dynamically adjust the resource allocation during runtime to meet the QoS of latency-sensitive (LS) tasks and determine the optimal resource allocation for batch tasks to improve GPU utilization and power efficiency. We implement the proposed mechanism on both simulator (*SMQoS*) and real GPU hardware (*RH-SMQoS*). The experimental results show that both *SMQoS* and *RH-SMQoS* can achieve better QoS for LS tasks and higher throughput for batch tasks compared to the state-of-the-art works. With hardware extension, the *SMQoS* can further reduce the power consumption by power gating idle computing resources.

1. Introduction

Graphics Processing Units (GPUs) have been widely adopted for accelerating general-purpose computation such as web service, social media, finance, and deep learning. GPUs utilize massive Thread Level Parallelism (TLP) to provide high computing capability. Due to the continuous improvement of GPU computing capability, it is difficult for a single task to utilize all its resources [1–5]. Therefore, multiple tasks are co-located on GPU to improve resource utilization. Based on the requirement for Quality of Service (QoS), GPU tasks can be divided into latency-sensitive (LS) tasks and batch tasks. For LS tasks, failure to meet QoS can result in an unsatisfactory user experience, such as game lag and dropped frames [6].

When multiple tasks are co-running on GPU, it is desirable to maximize the throughput of batch tasks while meeting the QoS of LS tasks. In addition, GPU tasks can generally be divided into memory-intensive (MI) tasks and compute-intensive (CI) tasks according to their performance sensitivity to computing resources. However, the computing characteristics of the tasks are not only related to the hardware architecture, but also the resource contention between co-running tasks. Therefore, a runtime system is needed to dynamically determine the computing resource requirements of tasks. When the batch task is MI, the runtime system can constrain its usage of computing resources to reduce power consumption.

Task consolidation on GPUs has received wide attention from both industry and academia. In the industry, Hyper-Q [7] in Nvidia Kepler architecture supports concurrent execution of multiple kernels on a single GPU with multiple independent queues. Multi-Process Server (MPS) [8] is also provided to support concurrent execution of GPU kernels from multiple applications on the same GPU. However, both methods lack effective control of GPU resources, and whether the kernels can execute concurrently depends on the resource status of the GPU. The Volta architecture improves MPS at the hardware level and provides QoS by limiting the number of available threads [9]. The Multi-Instance GPU (MIG) feature in the Ampere architecture can divide a single GPU into multiple GPU instances to provide QoS support [10]. Both of them are applied before kernel execution and does not support dynamic resource adjustment.

Meanwhile, two primary mechanisms are proposed in academia to share GPU resources among co-running GPU tasks, including Spatial Multitasking (SMT) [11] and Simultaneous Multikernel (SMK) [12]. SMT divides the Streaming Multiprocessors (SMs) on GPU into several disjoint subsets, each of which is assigned to one of the co-running

* Corresponding author at: School of Computer Science and Engineering, Beihang University, Beijing, China.
E-mail addresses: qingxiaosun@buaa.edu.cn (Q. Sun), yi.liu@buaa.edu.cn (L. Yi), hailong.yang@buaa.edu.cn (H. Yang), lmzhhh@buaa.edu.cn (M. Li), zhongzhi.luan@buaa.edu.cn (Z. Luan), depeiq@buaa.edu.cn (D. Qian).

https://doi.org/10.1016/j.parco.2022.102958
Received 24 May 2021; Received in revised form 15 July 2022; Accepted 17 July 2022
Available online 21 July 2022
0167-8191/© 2022 Elsevier B.V. All rights reserved.

tasks. SMK allows multiple tasks to co-run on a single SM simultaneously by switching them with time quota. For SMK and SMT, neither is superior over the other because their performance varies depending on the resource partitions and task mixes within each scheme [13]. Moreover, QoS for LS tasks cannot be supported in the original design of the above two mechanisms.

To provide QoS on GPU, existing research works can be primarily divided into two categories: *(1) task and resource scheduling*. The research works in this category propose new task scheduling and resource partition methods in order to meet the QoS requirement [14–18]. These methods are generally applied before task running, and cannot handle performance interference during runtime. *(2) runtime mechanism*. The representative research works in this category include *Spart* [19] and *Rollover* [6]. The drawback of *Spart* is that the linear prediction model it adopts leads to frequent SMs swap in and out, and thus severe performance degradation. Whereas for *Rollover*, the resource contention from intra-SM, such as load-store units and L1 cache, leads to performance degradation. Both of them fail to exploit the computing characteristics of batch tasks for reducing power consumption.

To address the drawbacks of existing works on GPU QoS support, we propose a new runtime mechanism *SMQoS*, which can dynamically adjust the resource allocation during runtime to meet the QoS, and in the meanwhile improve GPU utilization and power efficiency. This paper is an extension of our previous work [20]. Compared to [20], we further implement *SMQoS* on the real GPU hardware as *RH-SMQoS*, which supports the QoS of LS tasks, and achieves higher throughput for batch tasks. The specific contributions are as follows, where all except the fourth contribution require hardware extension.

- We propose a QoS management mechanism that monitors the performance of LS tasks during runtime and dynamically adjusts the SM allocation between LS and batch tasks to meet the QoS target.
- We dynamically determine the optimal SM allocation for batch tasks during runtime so that the idle SM resources can be used for improving utilization or be power gated to reduce power consumption.
- We implement a runtime system *SMQoS* by extending the CUDA API and GPU architecture. The experiment results show that *SMQoS* can effectively improve the throughput of batch tasks and reduce system power consumption while satisfying the QoS of LS tasks.
- We implement *RH-SMQoS* on real GPU hardware by proposing the mechanisms of task remapping and kernel transformation. The experiment results show that *RH-SMQoS* can effectively support the QoS of LS tasks in addition to the higher throughput of batch tasks.

The rest of this paper is organized as follows: Section 2 presents the background, and Section 3 discusses the related work. Section 4 presents the details of *SMQoS* and *RH-SMQoS* methodologies, the former requires hardware extension and the latter applies to existing GPU hardware. Section 5 and Section 6 present the evaluation results of *SMQoS* and *RH-SMQoS*. Section 7 concludes this paper.

2. Background

2.1. GPU terminology and execution model

The GPU consists of multiple SMs. As shown in Fig. 1, each SM contains hundreds of computing cores and other resources such as registers, shared memory, and L1 cache. To execute on GPU, the task is launched on the CPU host (①) and the parallel portion of the task, namely kernel is offloaded to the GPU through runtime API (②). A GPU kernel is tagged with a Software Work Queue (SWQ) ID (③), and pushed into Pending Kernel Pool located in Grid Management

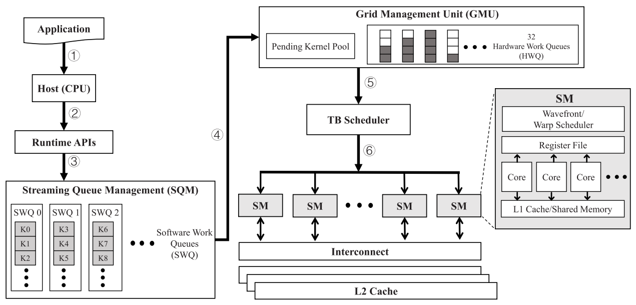


Fig. 1. The GPU task execution model.

Unit (GMU) (④). Kernels with the same SWQ ID are mapped into the same hardware work queue (HWQ). The thread blocks (TBs) from the head-of-the-line kernel in a chosen HWQ are dispatched to SMs by the TB scheduler (⑤⑥). According to the publicly available documents from NVIDIA [7], the number of HWQs is 32. Therefore, the maximum number of kernels that can concurrently run on the GPU is 32. Note that a TB needs to wait in GMU if its required resources are not available or the hardware limits are reached. The waiting delay depends on the execution time of currently running TBs.

2.2. GPU task characterization

To better understand the computing characteristics of GPU tasks, we perform a detailed analysis of representative benchmarks on NVIDIA Tesla P100 with 56 SMs. We control the number of SMs assigned to the task through *cudaStream* [3] and observe its performance. We use Instruction Per Cycle (IPC) as the performance metric. Refer to Section 5.1 for details of selected CUDA benchmarks.

The experiment results motivate us to classify the GPU tasks into memory-intensive (MI) tasks and compute-intensive (CI) tasks depending on their performance sensitivity to the number of SMs allocated, which are shown in Fig. 2. For CI tasks, their performance increases linearly with the number of SMs allocated (Fig. 2(a)). In this case, allocating more SMs to CI tasks improves GPU utilization. For MI tasks, their performance saturates with an increasing number of SMs allocated (Fig. 2(b)). In this case, with the number of SMs allocated for MI tasks to reach the optimal performance, the remaining SMs can be power gated to reduce GPU power consumption. Note that when multiple tasks are sharing GPUs, the optimal number of SMs needs to be determined during runtime. Therefore, a runtime system is required to dynamically analyze the computing characteristics of batch tasks and determine the optimal number of SMs to be allocated. Such a runtime system can improve GPU utilization and reduce power consumption while meeting QoS requirement for LS tasks.

2.3. Existing GPU QoS mechanisms

To meet QoS, the current research mainly shares GPU resources through three mechanisms. The first mechanism of sharing is implemented through time multiplexing. To provide QoS, time multiplexing supports resource preemption. When the high-priority task requires extra resources, it preempts the resources of the low-priority task. However, the preempted task can only be executed by re-launching the whole kernel task. The latency of kernel preemption and re-dispatching can significantly degrade the throughput of batch tasks.

The second mechanism is based on SMT, which divides SMs into several disjoint subsets, each is assigned to different tasks to co-run. For each subset, SMT adopts the default round-robin strategy to schedule TBs until the resource or design limit is reached. The representative work of SMT-based QoS management is *Spart* [19]. In *Spart*, the number of SMs required for each task is determined by estimating

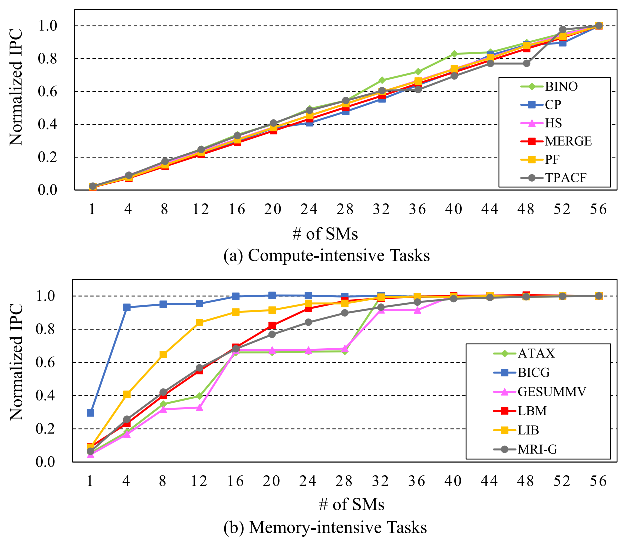


Fig. 2. The IPC of benchmark normalized to the performance with maximum SMs allocated on the NVIDIA P100 GPU.

the performance of LS tasks with the linear prediction model. The third mechanism is based on SMK, which simultaneously co-executes multiple tasks on a single SM. The representative research work of SMK is *Rollover* [6]. It meets the QoS requirement by using a quota-based strategy to dictate the execution of LS tasks and allocating just enough resource quota to LS tasks. However, both SMT and SMK approaches have drawbacks, such as performance instability and resource contention. Moreover, none of them exploits the computing characteristics of batch tasks to power gate the idle SMs for reducing power consumption.

In the meanwhile, we notice the changes of computing resources among different NVIDIA GPU generations [9]. We observe that the architecture development trend of GPUs is as follows: *(1) The number of SMs is growing rapidly*. The latest Volta, Turing and Ampere architectures are equipped with 80 SMs (Tesla V100), 72 SMs (Turing TU102) and 108 SMs (Tesla A100), respectively. *(2) The intra-SM resources remain almost constant.* The intra-SM resources include register file, shared memory, L1 cache, and GPU cores. In particular, the new MIG feature in NVIDIA A100 supports spatial sharing of tasks through resource partitioning. However, the resource partitioning in MIG is not flexible. The NVIDIA driver APIs only provide five profiles to create up to seven GPU instances. Specifically, the fractions of memory and SMs that the users can specify are (1/8, 1/7), (2/8, 2/7), (4/8, 3/7), (4/8, 4/7) and (full, 7/7) [21]. From the above observations, we believe that the future GPU architecture is more inclined to treat SMs as independent processing units and improve the computation capability of a single SM under the constraint of limited resources. Therefore, the SMT-based QoS mechanism is chosen as the baseline we compare with.

To address the drawbacks of existing SMT-based QoS management approaches, this paper proposes a runtime QoS management mechanism, which meets the QoS requirements of LS tasks by monitoring the QoS changes of LS tasks and dynamically adjusting SM resources allocated to LS tasks. Moreover, the mechanism can recognize the computing characteristics of batch tasks, and thus manage the idle SM resources more efficiently. Specifically, the idle SMs can be allocated to batch tasks to improve system throughput or power gated to reduce power consumption.

3. Related work

We classify the works related to this paper into GPU sharing mechanisms, QoS management on GPU, and GPU power-saving techniques. The details are described below.

GPU Sharing Mechanisms. There is a large body of research works focusing on executing multiple tasks on GPU to improve resource utilization [2–5,11–13,22–27]. Lee et al. [25] enabled multiple kernels to be allocated to the same core to maximize resource utilization. Spatial Multitasking (SMT) [11] enabled concurrent applications to share GPUs at SM granularity, whereas Simultaneous Multikernel (SMK) [12] proposed fine-grained resource management than SMT, where multiple applications share a single SM. *Maestro* [13] combined the advantages of SMT and SMK to achieve better performance for multiple tasks sharing GPU resources. *FLEP* [26] predicted the kernel duration and enabled preemption to explore different scheduling policies on GPU. *EffiSha* [27] transformed the program to enable preemption and switched kernels with low overhead of data saving. *CD-search* [3] improved GPU system throughput by classifying workloads using a novel off-SM bandwidth mode. *GPU Weaver* [2] maximized the use of sub-resources by adding a shared resource controller (SRC) between neighboring SMs. *Salus* [5] proposed two primitives (e.g., *fast job switching* and *memory sharing*) to achieve fine-grained GPU sharing between multiple deep learning applications. *Slate* [4] scheduled concurrent kernels with complementary demands through workload-aware design to alleviate resource contention. However, none of the above research works is capable of supporting QoS on GPU.

QoS Management on GPU. Existing works providing QoS on GPU can be mainly classified into two categories:

(1) Extending GPU task execution model [14–16,18,28,29]. TimeGraph [15] provided fairness by re-ordering the commands in the command queue of a GPU. *Baymax* [14] guaranteed QoS by predicting the execution time of the kernel, and scheduling kernels to satisfy the QoS of LS tasks. *Prophet* [28] utilized interference models to predict the performance degradation of LS tasks and identified “safe” co-locations to improve utilization. *SMGuard* [16] implemented resource reservation on the SM and preempted batch tasks if the reserved resources failed to meet the QoS. *Laius* [29] predicted the kernel duration and allocated more resources to the unexecuted kernels if the query runs slower than expected. The approaches in this category are uniformly applied before task running, and cannot effectively handle the performance interference during runtime.

(2) Hardware extensions to the GPU [6,19,30–32]. Spart [19] dynamically partitioned GPU resources between concurrently running applications. Tanasic et al. [30] proposed an SM-draining technique that improves the performance of high-priority processes by enabling preemptive scheduling on GPU. Li et al. [32] proposed a priority-based cache allocation (PCAL) mechanism to give preferential access to the cache and other on-chip resources to high-priority threads. Park et al. [31] proposed SM-flushing to immediately stop the execution of a kernel and flush all intermediate results. *Rollover* [6] controlled the kernel execution on cycle basis and the amount of thread-level parallelism to meet QoS. However, all the above approaches fail to exploit the computing characteristics of batch tasks for reducing power consumption while providing QoS support on GPU.

GPU Power-saving Techniques. Existing research works mainly focus on managing the cache, DRAM and network-on-chip (NoC) of GPU to reduce power consumption [31,33–40]. Hong et al. [35] proposed an empirical power model that relies on dynamic power events to reduce runtime GPU energy consumption. Park et al. [31] proposed SM-flushing, which can immediately stop the execution of a kernel and flush all intermediate results. Aghilinasab et al. [38] improved the idleness opportunity of the Warped Gates scheduler by static instruction re-ordering. Tabbakh et al. [39] proposed a sharing-aware TB scheduler that assigns data sharing TBs to the same SM in order to reduce data movements. *ITAP* [40] efficiently reduced the static power consumption of GPU execution units by exploiting their idleness. In contrast, *SMQoS* exploits the computing characteristics of batch tasks to power gate idle SMs. The above power-saving approaches further consider the influence of data locality or instruction dispatch, which are orthogonal to this paper.

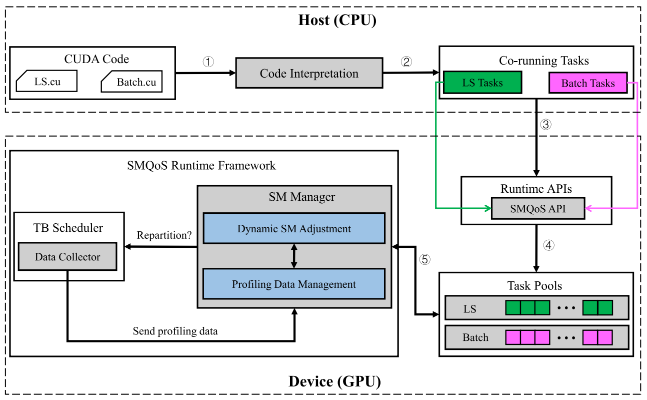


Fig. 3. The design overview of *SMQoS*.

Table 1
The interface of runtime API proposed by *SMQoS*.

SMQoS API	Parameters	Description
$cudaSetQoS$	void $*$ *kernel*	Pass the kernel pointer to the GPU through runtime API to identify the LS task.
	double IPC_{target}	Set the QoS target of the LS task to IPC_{target}.

This paper focuses on task co-location and QoS management on GPU, and thus we compare our proposed mechanism with the most relevant works in the following. Specifically, we compare *SMQoS* and *RH-SMQoS* with state-of-the-art works including *Spart*, *Rollover*, and *Slate* in the above.

4. SMQoS methodology

4.1. Design overview

SMQoS aims to meet the QoS of LS tasks by monitoring the performance of LS tasks and dynamically adjusting the SM allocation of LS tasks during runtime. The QoS metric can be IPC (evaluated in simulator) or task duration (evaluated on the real hardware). Since *SMQoS* extends GPU hardware in simulator, we choose IPC as the QoS metric. The design overview of *SMQoS* is shown in Fig. 3. The gray modules are designed or extended by *SMQoS*. The GPU kernels from multiple applications are interpreted into co-running GPU tasks through the *Code Interpretation* module (①②). Since the QoS requirement and the input data scale remain almost constant for a particular application across runs [41], we add a new API call *cudaSetQoS*, which is used to specify the LS task and its QoS target by the user (Table 1). The kernel function pointers are used to identify the LS tasks when invoking *cudaSetQoS*. IPC_{target} is used as the QoS target for LS task by *cudaSetQoS*. In *SMQoS*, IPC_{target} is the average IPC that the LS task needs to achieve during runtime. Since this paper focuses on QoS management under task co-location, we assume that IPC_{target} for a task can be achieved when run in isolation.

After *cudaSetQoS* is invoked (③), the GPU tasks are offloaded to GPU and pushed into one of the two task pools according to their task types (④). The task pools are used to manage LS tasks and batch tasks. The *SM Manager* manages the task pools through two sub-modules (⑤), the *Profiling Data Management (PDM)* module and the *Dynamic SM Adjustment (DSMA)* module. *SMQoS* determines whether to dynamically adjust SM allocation of LS tasks on an epoch-by-epoch basis. Note that the number of cycles per epoch is application-independent and should be specified regarding the GPU hardware. To collect the IPC of each task during the epoch, we extend the TB Scheduler with the *Data Collector (DC)* module. The *DC* module accumulates the number of instructions executed by the task and divides by the number of cycles to obtain the average IPC of the task during each epoch. At the end of each epoch, the *DC* module sends the IPC of each task to the *PDM* module in *SM Manager*. The *PDM* module records the IPC of each task as well as current SM allocation, and then sends the information to the *DSMA* module. The *DSMA* module determines the SM allocation for the next epoch using our proposed SM allocation algorithm (Sections 4.2 and 4.3). After that, the *SM manager* informs the TB scheduler to re-allocate the SMs according to the decision from the *DSMA* module. If the SMs need to be re-allocated, the TB scheduler swaps out/in the TBs on the SMs. Note that we allow the SM to be swapped out only when the TBs on it have completed normally, which avoids the overhead of handling dirty data and TB re-launching.

Note that *SMQoS* does not require any offline analysis or characterization of the tasks. During runtime, the *DSMA* module can determine the SM allocation dynamically through the corresponding algorithms (Sections 4.2 and 4.3), which does not rely on the computation characteristics of a task.

4.2. Maintaining QoS for LS tasks

In the case of concurrent execution, *SMQoS* gives priority to ensure that the QoS is met. *SMQoS* aims to dynamically adjust the number of SMs occupied by the LS task so that the average performance during execution reaches the QoS target.

As shown in Fig. 3, the *SM Manager* requires the TB scheduler to swap in the SM in time to meet the QoS of the LS task according to the profiling result from the *DSMA* module. Note that we do not need to determine the computing characteristics of the LS task since its QoS requirement always needs to be met. When the LS task needs more resources by swapping in SMs, the idle SMs are selected first to avoid swapping out SMs from the batch task. If there are no idle SMs available, the batch task is selected to swap out SMs, which are then re-allocated to the LS task. Algorithm 1 shows the SM allocation algorithm for the LS task, where IPC_{ave} is the average IPC of the task, N_{epoch} is the number of epochs elapsed for task execution, SM_{ls} is the number of SMs allocated to the LS task, and IPC_{epoch} is the average IPC of the task during current epoch. LS_{to_swapin} and $LS_{to\_swapout}$ are two boolean variables that specify the type of operation to be performed on the LS task. Specifically, the LS_{to_swapin} indicates the LS task to swap in an SM, whereas the $LS_{to\_swapout}$ indicates the LS task to swap out an SM.

Algorithm 1 Dynamic SM adjustment to maintain QoS.

```
1: Input: $N_{epoch}$, $IPC_{ave}$, $IPC_{target}$, $IPC_{epoch}$, $SM_{ls}$
2: Output: $SM_{ls}$
3: if $IPC_{ave} < IPC_{target}$ or $IPC_{epoch} < IPC_{target}$ then
4:     $LS_{to\_swapin} \leftarrow true$  // The LS task fails to meet the QoS target
5: else
6:     if $\frac{IPC_{ave} \times N_{epoch}}{N_{epoch}+1} > IPC_{target}$ and $IPC_{epoch} > IPC_{target}$ then
7:         $LS_{to\_swapout} \leftarrow true$  // The LS task will meet the QoS target in the next epoch
8:     end if
9: end if
10: // The LS task requires swapping operation
11: if $LS_{to\_swapin} == true$ then
12:     $SM_{ls} \leftarrow SM_{ls} + 1$  // The LS task swaps in an SM
13: else
14:     if $LS_{to\_swapout} == true$ then
15:         $SM_{ls} \leftarrow SM_{ls} - 1$  // The LS task swaps out an SM
16:     end if
17: end if
```

As illustrated in Algorithm 1, when the LS task fails to meet its QoS target (IPC_{ave} or IPC_{epoch} is less than IPC_{target}, Line 3), it immediately requires an SM to swap in (Line 4). To avoid the QoS oscillation, only one SM is swapped for the LS task at a time. To meet the QoS of the LS task, the conditions for swapping out are more restricted (both IPC_{ave} and IPC_{epoch} are greater than IPC_{target}, Line 6). The SM swapped out by the LS task (Line 7) can be allocated to the batch task to increase GPU utilization, or power gated to reduce power consumption. If the SMs need to be re-allocated, the TB scheduler swap in/out the TBs on the SM (Line 11–17). The optimal SM allocation for the batch tasks is determined by the computing characteristics of the batch task identified by *SMQoS* during runtime (Section 4.3).

4.3. Determining optimal resource allocation for batch tasks

SMQoS gives priority to increase the throughput of the batch task on the premise of meeting the QoS requirements of the LS task. However, when the throughput of the batch task converges as the number of SMs increases, *SMQoS* considers power gating idle SMs to reduce power consumption.

When the LS task swaps out an SM, whether to allocate the idle SM to the batch task depends on its computing characteristics. The challenge is how to determine the optimal number of SMs (i.e., opt_k) allocated to the batch task during runtime. *SMQoS* introduces $upper_k$, $lower_k$ and $threshold$ to determine opt_k, where $upper_k$ and $lower_k$ are boolean types. The $upper_k$ and $lower_k$ indicate whether the SM allocation reaches the upper bound and lower bound of the optimal allocation, respectively. $threshold$ controls the sensitivity to the QoS changes. We choose the optimal $threshold$ based on empirical studies. When the batch task performs the swapping operation, *SMQoS* records the settings of SM_{batch}, opt_k, $upper_k$ and $lower_k$ as history information to determine opt_k of batch task for next epoch, where SM_{batch} is the number of SMs allocated to the batch task. The history information also includes SM_{last} and IPC_{last}, which are the number of SMs occupied and the average IPC in the previous epoch respectively. An example to illustrate the procedure for determining opt_k is shown in Algorithm 2. *SMQoS* restricts that the GPU task can only swap in/out one SM at a time. Therefore, when both $upper_k$ and $lower_k$ are true, the value of opt_k is uniquely determined. To avoid frequent SM swapping of the batch task, opt_k remains unchanged during the rest of the execution.

Algorithm 2 Determine opt_k for corresponding batch task.

```
1: Input: $threshold$, $IPC_{last}$, $IPC_{epoch}$, $SM_{last}$, $SM_{batch}$
2: Output: $lower_k$, $upper_k$, $opt_k$
3: Precondition 1: Batch task swaps in an SM in the previous epoch
4: if $SM_{last} == opt_k$ then
5:     if $IPC_{epoch} > IPC_{last} \times (1 + threshold)$ then
6:         $lower_k \leftarrow true$  // $opt_k$ reaches the lower bound
7:         $opt_k \leftarrow SM_{batch}$
8:     else
9:         $upper_k \leftarrow true$  // $opt_k$ reaches the upper bound
10:    end if
11: end if
12: Precondition 2: Batch task swaps out an SM in the previous epoch
13: if $SM_{last} == opt_k$ then
14:     if $IPC_{epoch} < IPC_{last} \times (1 - threshold)$ then
15:         $lower_k \leftarrow true$  // $opt_k$ reaches the lower bound
16:     else
17:         $upper_k \leftarrow true$  // $opt_k$ reaches the upper bound
18:         $opt_k \leftarrow SM_{batch}$
19:     end if
20: end if
```

The detailed SM allocation strategy for the batch task is illustrated in Algorithm 3, where SM_{total} and SM_{active} are the total number of SMs and the number of active SMs, respectively. For Precondition 1, if all SMs on the GPU are active (occupied by batch task) (Line 5), SMQoS requires that the batch task swap out an SM to meet the QoS of the LS task (Line 6). Otherwise, the LS task will be allocated with an idle SM. For Precondition 2, when the number of SMs allocated to the batch task is less than opt_k (Line 10), the idle SM is allocated to the batch task (Line 11). Otherwise, *SMQoS* determines whether the upper bound is reached (Line 13). If not, the idle SM is still allocated to the batch task (Line 14); otherwise, *SMQoS* determines whether the lower bound is reached (Line 16). If not, the batch task swaps out an SM (Line 17). If the batch task swaps in the SM (Line 22), then the TB scheduler schedules the TBs on the SM (Line 23). Otherwise, the SM is power-gated to reduce power consumption (Line 25–30). Finally, *SMQoS* updates the details of active SMs occupied by the tasks (Line 32). In general, if the batch task is CI, there is no upper bound for opt_k. Therefore, the SM swapped out by the LS task is immediately swapped in by the batch task to increase utilization. If the batch task is MI, when the number of SMs allocated reaches opt_k, the idle SM is power gated to reduce power consumption.

Algorithm 3 Adaptive resource allocation for batch task.

```
1: Input: $SM_{total}$, $SM_{active}$, $SM_{ls}$, $SM_{batch}$, $opt_k$, $upper_k$, $lower_k$
2: Output: $SM_{active}$, $SM_{batch}$
3: Precondition 1: LS task requires to swap in an SM
4: // Determine whether the batch task needs to swap out an SM
5: if $SM_{active} == SM_{total}$ then
6:     $Batch_{to\_swapout} \leftarrow true$
7: end if
8: Precondition 2: LS task swaps out an SM
9: // Determine whether the batch task needs to swap in an SM
10: if $SM_{batch} < opt_k$ then
11:     $Batch_{to\_swapin} \leftarrow true$  // The number of SMs allocated is less than $opt_k$
12: else
13:     if $upper_k == false$ then
14:         $Batch_{to\_swapin} \leftarrow true$  // $opt_k$ does not reach the upper bound
15:     else
16:         if $lower_k == false$ then
17:             $Batch_{to\_swapout} \leftarrow true$  // $opt_k$ does not reach the lower bound
18:         end if
19:     end if
20: end if
21: // The batch task requires swapping operation
22: if $Batch_{to\_swapin} == true$ then
23:     $SM_{batch} \leftarrow SM_{batch} + 1$  // The batch task swaps in the SM
24: else
25:     if $Batch_{to\_swapout} == true$ then
26:         $SM_{batch} \leftarrow SM_{batch} - 1$  // The batch task swaps out an SM
27:         Power gate the SMs
28:     else
29:         Power gate the SM
30:     end if
31: end if
32: $SM_{active} \leftarrow SM_{ls} + SM_{batch}$  // Update the number of active SMs
```

4.4. Applying to real GPU hardware

We discuss the implementation of *SMQoS* on the real GPU hardware (namely *RH-SMQoS*), where the task duration is used as the QoS metric. Since the task cannot be explicitly assigned to the specified SMs on the real hardware, the main challenge of *RH-SMQoS* implementation is how to support SMT without incurring significant overhead. To achieve above goal, *RH-SMQoS* needs to change the default execution model of the GPU task through code transformation and explicitly manage the mapping of GPU kernel to the specified SMs.

The previous works that manage resource allocation on commodity GPU include *SMGuard* [16] and *Slate* [4], which are closely related to *SMQoS*. Both *SMGuard* and *Slate* rely on persistent thread [42] to control GPU resource allocation among tasks. *SMGuard* implemented resource reservation and dynamic scheduling on GPU through multiple workers. When the LS task starts on GPU, QoS can be satisfied by reserved resources or by preempting the resources occupied by batch tasks. Moreover, the block-tasks that are not completed due to eviction will be remapped to remaining workers to avoid unnecessary kernel relaunch. *Slate* implemented the idea of SMT by adding arguments to the kernel function and launching the task to a designated range of SMs. When a new task starts on GPU, the previous task has to exit and relaunch to the adjusted range of SMs if needed. Note that when adjusted, the tasks will be re-launched multiple times; for each time, partial TBs will be placed within the specified range of SMs until all TBs have been scheduled.

In general, the design of *SMGuard* is similar to the idea of Simultaneous Multikernel (SMK), which is different from the design philosophy of *SMQoS*. In addition, although *SMGuard* reduces the performance degradation of the batch task through online task remapping, it still leads to workload imbalance. In contrast to *SMGuard*, the design philosophy of *Slate* is similar to *SMQoS* that adopts the idea of Spatial Multitasking (SMT). However, *Slate* causes the task to be re-launched for multiple times, which results in considerable performance overhead. In addition, *Slate* does not support QoS management on GPU.

To address the above drawbacks, we propose *RH-SMQoS*, which implements *SMQoS* on the real GPU hardware. *RH-SMQoS* is used to

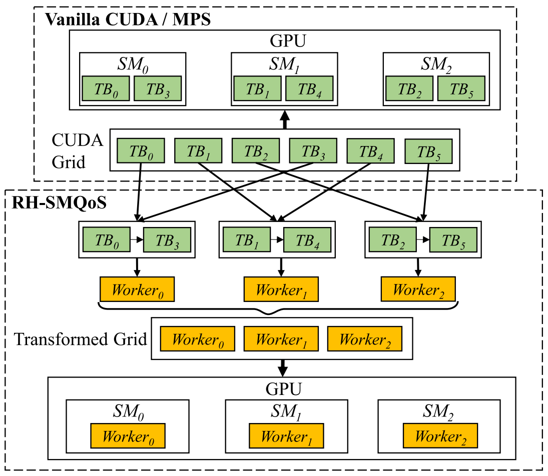


Fig. 4. The design overview of *RH-SMQoS* task remapping.

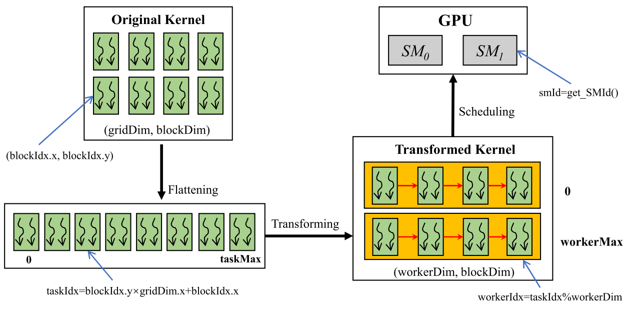
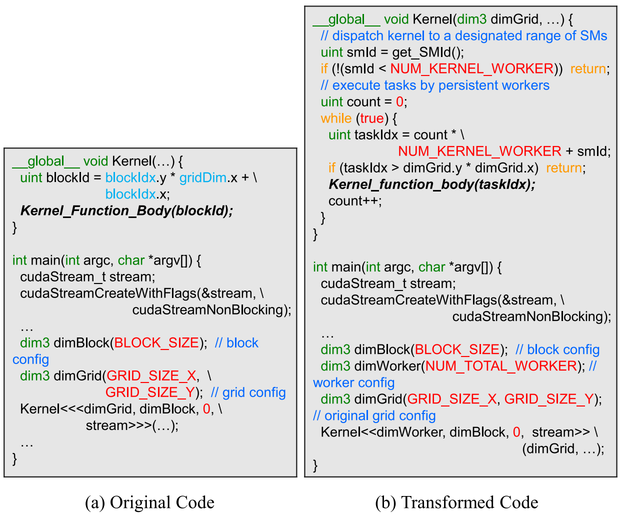


Fig. 5. The kernel transformation and worker scheduling in *RH-SMQoS*.



```
__global__ void Kernel(…) {
  uint blockId = blockIdx.y * gridDim.x + \
                 blockIdx.x;
  Kernel_Function_Body(blockId);
}

int main(int argc, char *argv[]) {
  cudaStream_t stream;
  cudaStreamCreateWithFlags(&stream, \
                         cudaStreamNonBlocking);
  …
  dim3 dimBlock(BLOCK_SIZE);  // block
config
  dim3 dimGrid(GRID_SIZE_X,  \
               GRID_SIZE_Y);  // grid config
  Kernel<<<dimGrid, dimBlock, 0, \
           stream>>>(…);
  …
}
```

(a) Original Code

```
__global__ void Kernel(dim3 dimGrid, …) {
  // dispatch kernel to a designated range of SMs
  uint smId = get_SMId();
  if (!(smId < NUM_KERNEL_WORKER))  return;
  // execute tasks by persistent workers
  uint count = 0;
  while (true) {
    uint taskIdx = count * \
                NUM_KERNEL_WORKER + smId;
    if (taskIdx > dimGrid.y * dimGrid.x)  return;
    Kernel_function_body(taskIdx);
    count++;
  }
}

int main(int argc, char *argv[]) {
  cudaStream_t stream;
  cudaStreamCreateWithFlags(&stream, \
                         cudaStreamNonBlocking);
  …
  dim3 dimBlock(BLOCK_SIZE);  // block config
  dim3 dimWorker(NUM_TOTAL_WORKER); //
worker config
  dim3 dimGrid(GRID_SIZE_X, GRID_SIZE_Y);
// original grid config
  Kernel<<dimWorker, dimBlock, 0,  stream>> \
                              (dimGrid, …);
}
```

(b) Transformed Code

Fig. 6. An example of transformed GPU program for 2D grids with *RH-SMQoS*.

evaluate the feasibility of *SMQoS* on real hardware, which does not rely on any hardware extensions. Considering that *RH-SMQoS* should not incur additional overhead such as kernel re-launch [4] and workload imbalance [16], it does not support dynamic resource allocation. Instead, *RH-SMQoS* utilizes the persistent thread mechanism and offline code transformation to schedule each co-located task to a designated range of SMs. In such a way, *RH-SMQoS* achieves explicit SMT-based SM allocation (the fundamental idea of *SMQoS*) on real GPU hardware. However, *RH-SMQoS* does not realize the algorithms in *SMQoS*, which requires hardware support for resource adjustment and data collection at runtime. The detailed implementation of *RH-SMQoS* is described below.

Persistent Worker. When the task launches on GPU, Vanilla CUDA or MPS assigns the TBs to the SMs using Round-Robin (RR) policy. The vanilla CUDA does not support concurrent task scheduling due to the different contexts from multiple tasks. Prior to Volta architecture, MPS can combine multiple contexts into one context, enabling concurrent scheduling of multiple tasks. However, only the GPU resources left by the current task execution can be utilized to launch a new task. The Volta MPS (referred to as *Volta-MPS*) provides hardware support for concurrent execution of tasks, which enables MPS clients to submit tasks directly to the work queues within the GPU [9]. Moreover, *Volta-MPS* can provide QoS support by limiting the number of available threads on GPU. Note that when adjusting the limiting factor of available threads during runtime, only the tasks created afterward obey the new constraint, whereas the tasks already existed are not affected.

As shown in Fig. 4, unlike Vanilla CUDA or MPS, *RH-SMQoS* implements the idea of SMT by abstracting CUDA grids to persistent workers. In the worker, multiple TBs are executed in order. The number of workers is the same as the number of SMs they occupy so that workers can be mapped to SMs correspondingly. After that, we can specify the number of workers and SM range to manage QoS with the software approach. Note that *RH-SMQoS* is implemented based on the CUDA stream, which can be applied to broader GPU architectures. In contrast, *Volta-MPS* can only be applied to GPUs with Volta architecture and beyond.

Kernel Transformation. Fig. 5 shows the strategy of kernel transformation and worker scheduling proposed in *RH-SMQoS*. *RH-SMQoS* transforms a 1D, 2D or 3D grid to a 1D grid, without modifying the internal structure of TBs. After transformation, the threads in each TB remain the same. Fig. 5 shows an example of the transformation of a 2D grid. The attributes of the original kernel include *gridDim* and *blockDim*, which specify the size of the grid and TB respectively. Since it is a 2D grid, the position of each TB is specified by two coordinates: *blockIdx.x* and *blockIdx.y*. *RH-SMQoS* flattens the original grid into a 1D queue. In order to track the queue status, *RH-SMQoS* introduces *taskIdx* to index the tasks, which ranges from 0 to *taskMax*. The *taskIdx* can be calculated based on original kernel parameters.

After the grid transformation, *RH-SMQoS* creates a set of workers to which the tasks in the queue are scheduled, and introduces *workerIdx* to index the workers. Since the values of *gridDim* and *blockIdx* no longer represent the grid dimensions, to maintain the original kernel semantics, *RH-SMQoS* replaces *gridDim* and *blockIdx* with *workerDim* and *workerIdx* respectively. The *workerDim* is the number of workers specified by the transformed kernel, and the *workerIdx* can be calculated based on *taskIdx* and *workerDim*. Finally, the workers can be mapped to the specific SM range through customized worker scheduling strategy, which can control the SM occupancy of the kernel.

Worker-SM Mapping. *RH-SMQoS* maps a kernel to a range of SMs and ensures that the kernel only runs on these SMs. However, it cannot guarantee one-to-one mapping between workers and SMs. Although we can obtain the SM ID occupied by the kernel using the software approach [3], when the kernels are executed concurrently, the SM ID occupied by the same task index is not always the same. This indicates the TB scheduling on GPU does not follow a strict RR policy.

To address the above problem, we consider using the SM ID to specify the task index in the queue. Similar to *SMGuard*, we use LibTooling and LibASTMatchers [43] to achieve source-to-source code transformation as shown in Fig. 6. Fig. 6(b) shows the transformed GPU program for 2D grids by *RH-SMQoS*, and Fig. 6(a) shows the original code. Before launching the kernel, *RH-SMQoS* introduces the variable *dimWorker* with the size of *NUM_TOTAL_WORKER*, which is the total number of SMs in the GPU (e.g., 80 in Volta). The *dimGrid* is replaced by *dimWorker* when launching the kernel. In addition, the original *dimGrid*

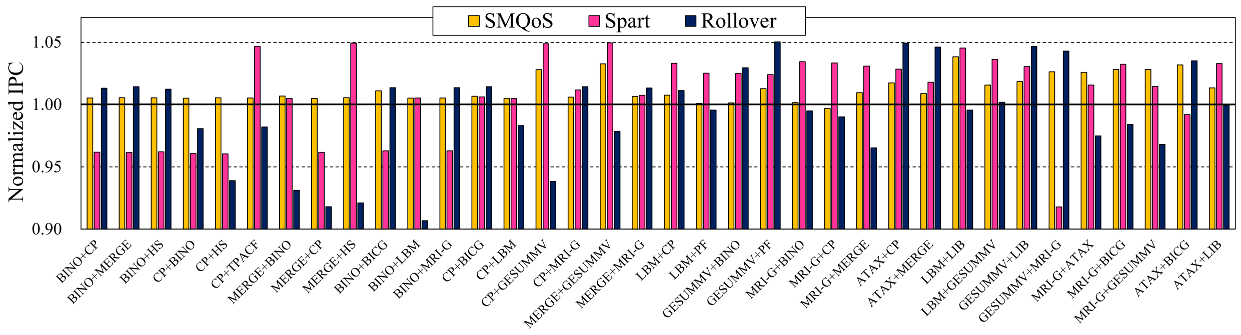


Fig. 7. The normalized IPC when setting the QoS policy to 95%. The x-axis is the co-running tasks with the first benchmark as the LS task.

Table 2
GPGPU-Sim configurations.

Number of SMs	24
Core configuration	32 SIMT lanes, 1.4 GHz, GTO Warp Scheduler
SM configuration	16 KB L1 D-cache, 4-way assoc, 128B block 48 KB Shared Memory, 32678 Registers
L2 unified cache	768 KB total, 128 KB/channel 8-way assoc, 128B block
Instruction cache	2 KB, 4-way assoc, 128B block
Texture cache	12 KB, 24-way assoc, 128B block, LRU
Constant cache	8 KB, 2-way assoc, 64B block
Interconnection configuration	2D mesh, 1.4 GHz, 32B channel width

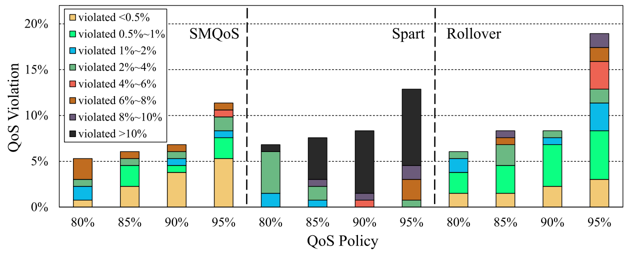


Fig. 8. The percentage of QoS violation under different QoS policies.

is added to the kernel parameters. In the kernel function, the worker-SM mapping is materialized with code injection. If the provisioned SM does not fall into the designated range, the worker exits.

The *NUM_KERNEL_WORKER* is the number of workers assigned to the kernel by the user. If the worker is assigned to the specified SM range, the worker executes the tasks sequentially according to the *taskIdx*. Unlike existing works, we use the SM ID to specify *taskIdx*, thus avoiding multiple re-launches of the kernel. When the *taskIdx* reaches *taskMax*, which means there are no more tasks to be scheduled. The workers exit after completing all tasks.

5. Evaluation on simulator

5.1. Experimental setup

To evaluate *SMQoS*, we use GPGPU-Sim v3.2.2 [44] with the simulation configuration same as [3] (Table 2). Meanwhile, we rely on GPUWattch [45] to measure power consumption. GPUWattch is a GPU power model integrated into GPGPU-Sim. We select 12 benchmarks from Rodinia [46], Parboil [47], NVIDIA SDK [48], ISPASS-2009 [44] and PolyBench [49], including six CI tasks and six MI tasks (Table 3). We select two tasks to co-run as a task mix, which consists of one LS task and one batch task. We divide the task co-running into four categories based on the computing characteristics of the task mixes: CI–CI, CI–MI, MI–CI, and MI–MI. For instance, CI–MI indicates that the LS task is CI, and the batch task is MI. Note that the task classification such as CI and MI is only used to guide representative task mixes, which is not required by *SMQoS*. Since the simulation results are accurate when running longer than $1M$ cycles [11], we run $2M$ cycles for each task mix. The length of one epoch is set to $10K$ cycles according to [6]. If one task ends before $2M$ cycles, it is re-executed. If one task is executed multiple times, we use the total number of instructions and cycles to calculate its IPC. Initially, the SMs are evenly partitioned between the LS task and the batch task.

To evaluate meeting QoS requirement, we use the percentage of QoS targets that are reached (QoS_{reach}) for comparison. We use IPC as the QoS metric and compare with *Spart* and *Rollover*. The QoS_{reach} is defined as $\frac{\#of\ Sucess\ Cases}{\#of\ Total\ Cases}$. Consistent with existing works [6,19], the QoS target (IPC_{target}) is defined as the percentage (i.e., QoS policy) of IPC when running isolated ($IPC_{isolated}$). The QoS policy ranges from 80% to 95%, with a stride of 5%. We compare the throughput of the batch task and power consumption using different QoS approaches. Due to the page limit, we only present the experimental results not included in our previous work [20]. The readers can refer to [20] for the throughput and power consumption results of *SMQoS*.

5.2. Overall performance comparison

In this section, we evaluate the efficiency of *SMQoS* to meet QoS target. Fig. 7 shows the IPC of LS tasks normalized to their QoS targets when the QoS policy is 95%. Although higher results mean better QoS, the goal is to achieve the QoS target just enough so that more resources can be used by batch tasks for higher utilization or power gated for less power consumption. As shown in Fig. 7, *SMQoS* reaches the QoS target just enough in most cases, whereas *Spart* and *Rollover* often end up with two extreme cases: *(1)* fail to reach the QoS target; *(2)* exceed the QoS target too much. The throughput of the batch task degrades significantly in case *(2)*.

For *SMQoS*, the runtime QoS monitoring and feedback control are more effective for SM re-allocation, which achieve stable QoS for LS tasks. To further understand the QoS results, Fig. 8 presents how much the QoS is violated within the percentage of QoS violations across different QoS policies. The percentage of QoS violations is defined as the number of co-runnings that suffer from QoS violations divided by the total number of co-runnings. For all QoS policies, *SMQoS* violates the QoS target by less than 8%, whereas *Spart* and *Rollover* violate the QoS target by more than 10% and 8% in the worst cases, respectively. Especially for 95% QoS policy, only 3% and 17.4% of the violations is less than 8% with *Spart* and *Rollover*, respectively. This demonstrates *SMQoS* is effective even when the QoS policy is tight for the LS task.

Fig. 9 presents the change in the number of SM_active (SM_{active}) and SM_optimal (opt_k) as epoch increases at 95% QoS policy. We randomly select four benchmarks from Table 3, including two CI tasks (*BINO* and *CP*) and two MI tasks (*BICG* and *LBM*). Furthermore, we randomly

Table 3
Benchmarks used for evaluation.

Benchmark	Kernel	Domain	Source	Type
BINO	binomialOptionsKernel	Finance	CUDA SDK [48]	CI
MERGE	mergeSortSharedKernel	Sorting Algorithms		
CP	cuda_cutoff_potential	Biomolecular Simulation	Parboil [47]	
TPACF	gen_hists	Astronomy		
HS	calculate_temp	Physics Simulation	Rodinia [46]	
PF	dynproc_kernel	Grid Traversal		
ATAX	atax_kernel	Linear Algebra	PolyBench [49]	MI
BICG	bicg_kernel	Optoelectronic Engineering		
GESUMMV	gesummv_kernel	Linear Algebra		
LIB	Pathcalc_Portfolio_Kernel	Market Model	ISPASS-2009 [44]	
LBM	performStreamCollide_kernel	Fluid Dynamics	Parboil [47]	
MRI-G	binning_kernel	Image Processing		

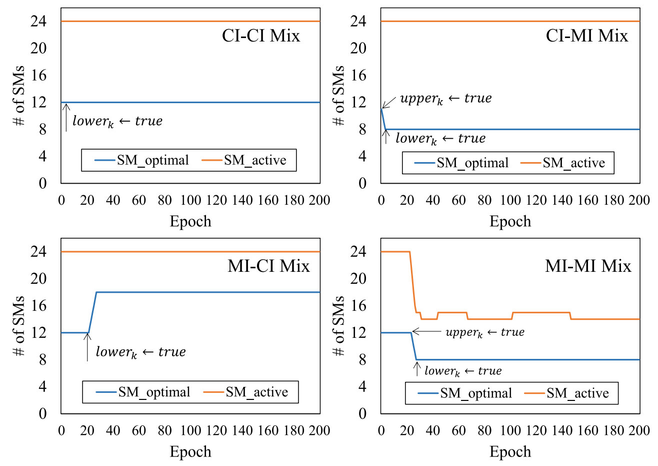


Fig. 9. When setting the QoS policy to 95%, the number of active SMs (SM_active) and the optimal number of SMs (SM_optimal) for the batch task during the dynamic adjustment of *SMQoS*.

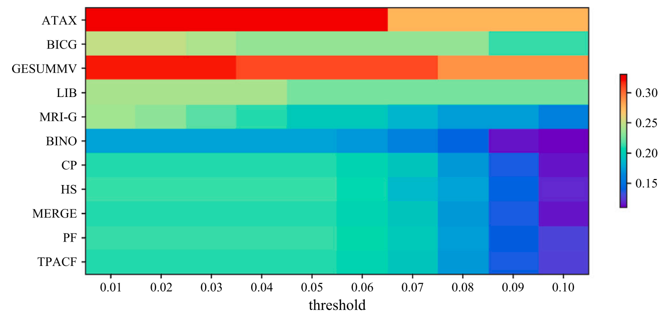


Fig. 10. The threshold sensitivity analysis when the LS task is *LBM* with 95% QoS policy. The *y*-axis indicates the throughput of batch tasks normalized to isolated execution.

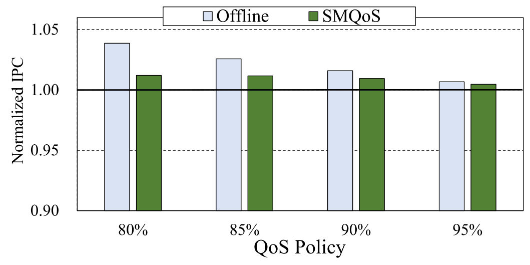


Fig. 11. The IPC of *SMQoS* and *Offline* normalized to QoS targets of LS tasks.

form four different types of task mixes, such as CI–CI (*BINO-CP*), CI–MI (*BINO-LBM*), MI–CI (*BICG-CP*), and MI–MI (*BICG-LBM*). When the batch task is CI, $lower_k$ is quickly set to *true*, and $upper_k$ keeps $false$ until the end of the program execution. This is because the performance of CI tasks increases linearly with the number of SMs. In contrast, when the batch task is MI, both $upper_k$ and $lower_k$ are quickly set to *true*, which means that opt_k has been uniquely determined. This is because the performance of MI tasks saturates with an increasing number of SMs allocated.

SMQoS can effectively improve the throughput of batch tasks while meeting QoS requirements of LS tasks [20]. We also implement the *SMQoS* without dynamic adjustment algorithms (namely *Static-SMQoS*), which statically sets the SM allocation ratio of the LS task as QoS policy. The QoS_{reach} of *SMQoS* and *Static-SMQoS* under 95% QoS policy are 88.6% and 58.3%, respectively. Furthermore, *Static-SMQoS* reduces the throughput of batch tasks by an average of 32.5% compared to *SMQoS*. The results indicate that dynamic adjustment algorithms are necessary to avoid QoS violations and GPU under-utilization.

5.3. Threshold sensitivity analysis

In Section 4.3, *SMQoS* introduced *threshold* to determine the opt_k of batch tasks. To understand the sensitivity of *threshold*, we measure the throughput of batch tasks under different settings of *threshold* when co-running with the LS task *LBM* at 95% QoS policy. Other task mixes exhibit similar tendency, which are omitted for conciseness. The throughput of each batch task is normalized to isolated execution. As shown in Fig. 10, the throughput of each batch task remains the same until the threshold exceeds a certain value. Beyond that value, the throughput decreases significantly across all batch tasks. This is because when the threshold is small (e.g., 0.01–0.05), *opt_k* can hardly reach by the upper limit set by the threshold, since there are few SMs allocated to the batch task during runtime in order to satisfy the QoS of LS task. When the threshold is large enough (e.g., 0.06–0.1), it forces opt_k converge quickly bounded by the upper limit, which leaves the value of *opt_k* smaller than its optimal setting, and thus deteriorates the throughput of batch tasks. In our experiments, setting the threshold to 0.01 gives us the best results across all task mixes. However, the threshold can be optimized when adopting *SMQoS* to customized hardware configurations.

5.4. Overhead analysis

SMQoS may introduce runtime overhead (e.g., SM swapping and data profiling) when re-allocating the number of SMs occupied by each task. To quantify the runtime overhead, we compare *SMQoS* against the optimal SM allocation determined through offline analysis (*Offline*). The optimum SM allocation satisfies: *(1)* the LS task occupies just enough SMs to meet its QoS, and *(2)* the remaining SMs are assigned

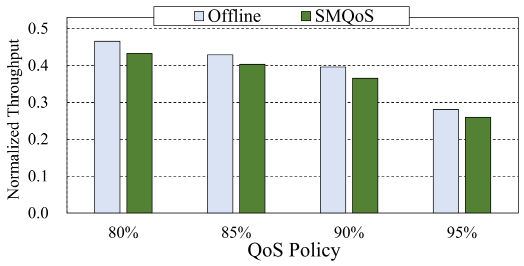


Fig. 12. The throughput of *SMQoS* and *Offline* normalized to isolated execution of batch tasks.

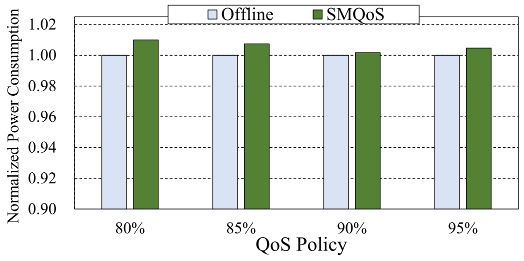


Fig. 13. The power consumption of *SMQoS* normalized to *Offline*.

to batch tasks for better throughput. Fig. 11 shows the IPC of *SMQoS* and *Offline* normalized to the QoS targets of LS tasks. *SMQoS* is closer than *Offline* to the QoS target under all co-runnings. This is due to the dynamic adjustment of *SMQoS* during runtime. Fig. 12 shows the throughput of *SMQoS* and *Offline* normalized to the isolated execution of batch tasks. *SMQoS* achieves lower throughput than *Offline* by 7.07% on average. This is because *SMQoS* performs swapping operations that decreases the throughput. Fig. 13 shows that the power consumption of *SMQoS* is 0.6% higher than *Offline* on average.

Future generation of GPUs may be equipped with more SMs. In that case, the *SM manager* needs to record more SM status information (i.e., idle or busy). However, this can be achieved through 1-bit registers, and the hardware overhead is negligible. For runtime overhead, the strategy of only swapping in or out one SM at a time avoids the scheduling latency. Besides, the complexity of related algorithms remains unchanged as the number of SMs increases. In sum, we believe the overhead of *SMQoS* is acceptable for future GPUs.

5.5. Beyond pairwise

To demonstrate that *SMQoS* can be applied beyond pairwise co-running tasks, we select three CI tasks and three MI tasks from Table 3. We evaluate the co-running with three tasks: one task as the LS task and the remaining two as the batch tasks. The experiments end up with six kinds of task mixes, where *(1)* the LS task is CI or MI, and *(2)* the batch task mix is CI–CI, CI–MI or MI–MI. Under each QoS target, $6 \times C_5^4 = 60$ task co-runnings are evaluated. Fig. 14 shows the IPC of LS tasks normalized to their QoS targets in the co-running. In general, the LS tasks meet their QoS requirements under all QoS targets. In most cases, *SMQoS* only exceeds the QoS targets by less than 2%. Especially when the LS task is *LBM* and the QoS target is set to 90%, *SMQoS* only exceeds the QoS target by 0.32%. This is because *SMQoS* enables the LS task to achieve the QoS target just enough, and thus more computing resources can be allocated to batch tasks for higher utilization or power gated for less power consumption.

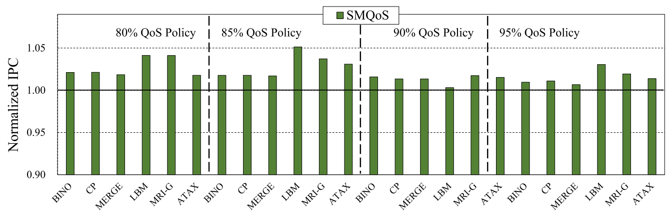


Fig. 14. The IPC of the LS task normalized to the QoS target in the co-running of three tasks. The *x*-axis indicates the LS task.

Table 4
Hardware and software specifications.

Hardware	Software
CPU: Intel(R) Xeon(R) CPU E5-2680 v4 @2.40 GHz	OS: Linux version 3.10.0
GPU: NVIDIA Tesla V100	GPU driver: 410.79

6. Evaluation on real GPU hardware

6.1. Experimental setup

The hardware and software specifications are presented in Table 4. We randomly select eight benchmarks from Table 3, including four CI tasks and four MI tasks. Among the selected tasks, two CI tasks (*HS* and *MERGE*) and two MI tasks (*LIB* and *LBM*) are chosen as LS tasks, whereas the rest tasks (*PF*, *BINO*, *GESUMMV* and *ATAX*) are left as batch tasks. CUDA MPS is used to enable concurrent task execution on GPU for the *Volta-MPS* method. In addition, we extend *Slate* to support QoS (named as *SlateQoS*). *SlateQoS* achieves the TB-SM mapping by re-launching the kernel multiple times. Since *RH-SMQoS* and *SlateQoS* map workers to SMs sequentially to support SMT, each SM can only be occupied by one TB (1,024 threads). For *Volta-MPS* and *Baseline* (isolated execution), we control the TB size to 1,024 and keep the thread utilization consistent with *RH-SMQoS* to provide a fair comparison.

For *RH-SMQoS* and *SlateQoS*, the SM allocation ratio for the LS task is equal to the QoS policy. Likewise, the thread limit ratio of *Volta-MPS* is set according to the QoS policy. We still use the percentage of task co-locations with QoS target reached (QoS_{reach}) to evaluate the effectiveness of different approaches for meeting QoS requirement. We use the normalized performance (NPM) to measure the performance of LS tasks. NPM is defined as $\frac{T_i^s}{\alpha T_i^c}$, where α is the QoS policy, T_i^c is the task duration when the task is executed under co-location, and T_i^s is the task duration when the task is executed alone (*Baseline*). NPM greater than one indicates that the QoS target is reached. We use the normalized throughput (NTP) defined as $\frac{T_i^s}{T_i^c}$ to measure the throughput of batch tasks. To make the evaluation statistically significant, each task co-location is executed ten times and the average result is reported. Note that we do not include the evaluation results on power consumption since it is infeasible to control the power gating of the SM on real GPU hardware.

6.2. Results for achieving QoS

Fig. 15 presents the normalized performance of LS tasks under different QoS policies. In general, the QoS_{reach} of *RH-SMQoS* under all cases is 93.8%, whereas *SlateQoS* and *Volta-MPS* are 84.4% and 76.6%, respectively. Especially when the QoS policy is tight such as 95%, the QoS_{reach} of *RH-SMQoS* is 100%, whereas *SlateQoS* and *Volta-MPS* are 75% and 68.8%. This indicates *RH-SMQoS* is able to guarantee the QoS of LS tasks even at stringent QoS target.

Moreover, we observe significant performance improvement for LS tasks under various task co-locations with *RH-SMQoS*. The overhead of re-launching and atomic operations leads to the inefficiency of

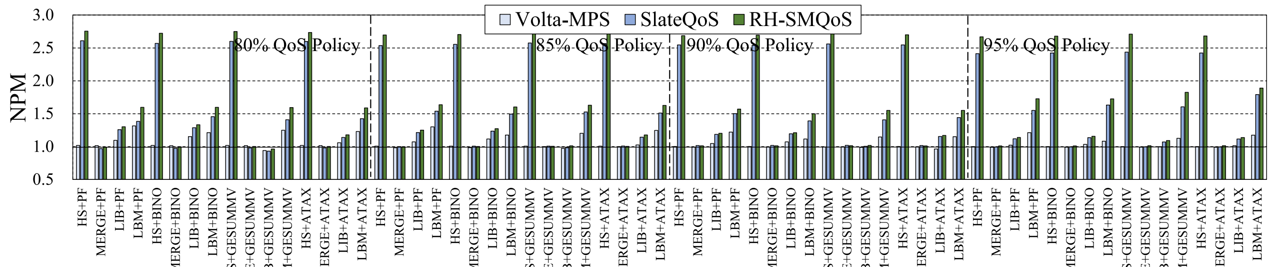


Fig. 15. The normalized performance of LS tasks under different QoS policies. The NPM greater than one means the QoS target is satisfied.

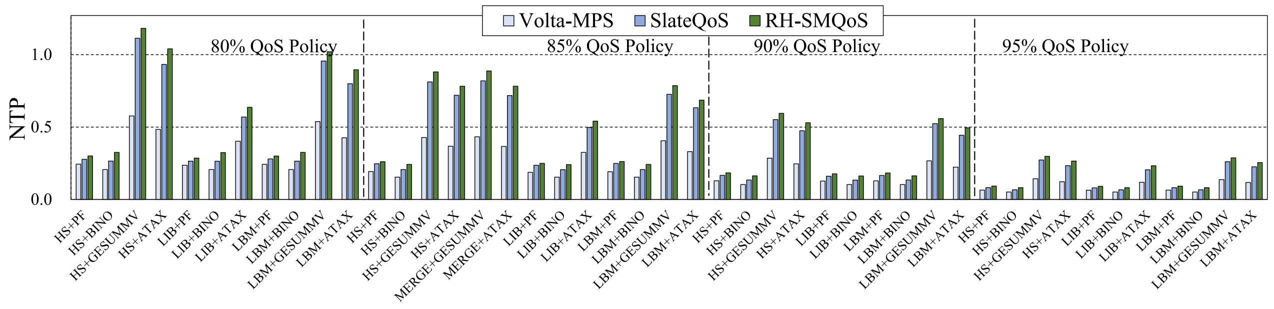


Fig. 16. The normalized throughput of batch tasks under different QoS policies.

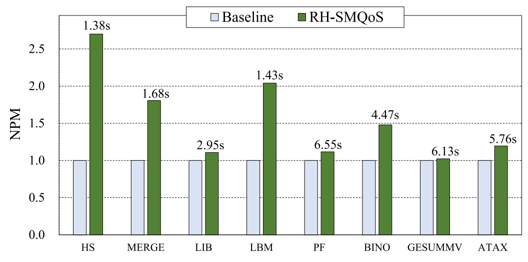


Fig. 17. The performance of each benchmark with *RH-SMQoS* normalized to *Baseline*.

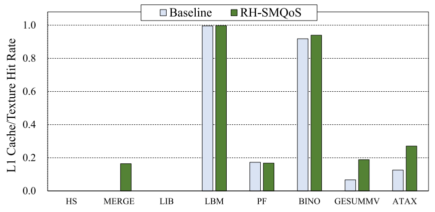


Fig. 18. The L1 cache/texture hit rate of each benchmark with *RH-SMQoS* and *Baseline*. The missing bar indicates the accesses never hit in the L1 cache/texture.

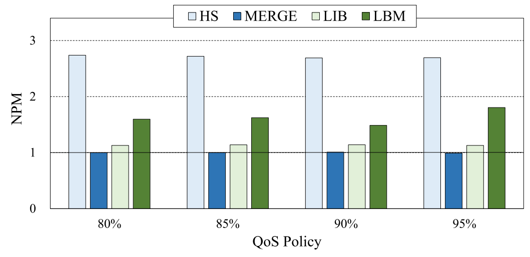


Fig. 19. The normalized performance of the LS task in the co-running of three tasks. The x-axis indicates the QoS policy.

SlateQoS. Both *RH-SMQoS* and *SlateQoS* achieve the most significant performance speedup when the LS task is *HS*. The reason is that *HS* is particularly sensitive to intra-SM resource contention, which can be effectively mitigated by SMT. *Volta-MPS* achieves the best performance when the LS task is *LIB*. This is because *LIB* is heavily memory-intensive and insensitive to intra-SM resource contention. All three approaches perform worst when the LS task is *MERGE*. This is because *MERGE* exhibits a large number of thread synchronization operations, which dominates the execution time. The performance gap between *RH-SMQoS* and *Volta-MPS* becomes larger as the QoS policy becomes tighter.

6.3. Results for improving throughput

Fig. 16 shows the normalized throughput of batch tasks under different QoS policies. Note that we only select the task co-locations that all three approaches meet QoS. Overall, *RH-SMQoS* achieves the highest throughput in all task co-locations. In general, the average normalized throughput of *RH-SMQoS* is 41.2%, whereas *SlateQoS* and *Volta-MPS* are 37.3% and 23.9%, respectively. The results demonstrate that *RH-SMQoS* achieves the highest throughput of batch tasks while meeting the QoS requirement of LS tasks. The reason for the inefficiency of *SlateQoS* compared to *RH-SMQoS* is the overhead of re-launching and synchronization operations. Moreover, we observe that all three approaches achieve high throughput when the batch task is *GESUMMV* or *ATAX*. This is because both *GESUMMV* and *ATAX* are memory-intensive, which are insensitive to thread allocation. Even with these two batch tasks, *RH-SMQoS* achieves the highest throughput due to our proposed techniques such as persistent worker, kernel transformation and worker-SM mapping.

6.4. Overhead analysis

The overhead introduced in *RH-SMQoS* includes: *(1)* using persistent workers to execute TBs in order; *(2)* obtaining the SM ID through the software approach; *(3)* adding extra judgment logic to implement SMT. To evaluate the overhead, we compare the performance of each benchmark in the original and transformed implementation. We use the

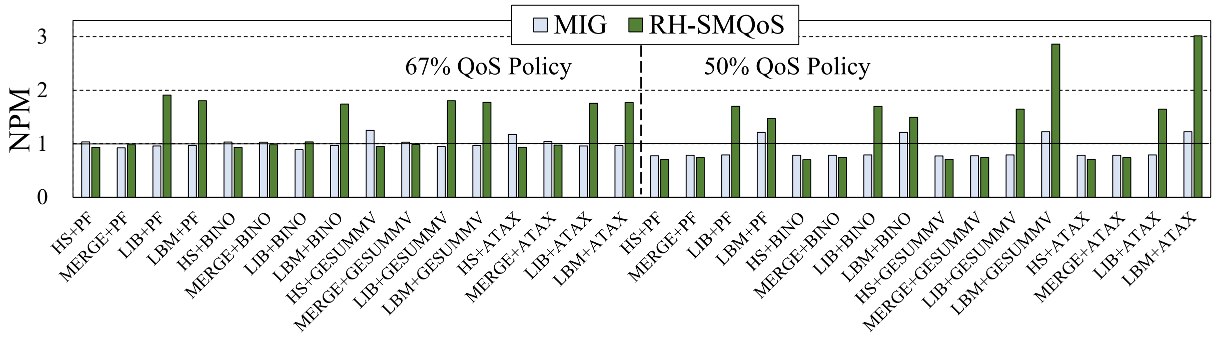


Fig. 20. The normalized performance of LS tasks under 67% and 50% QoS policy on A100 GPU.

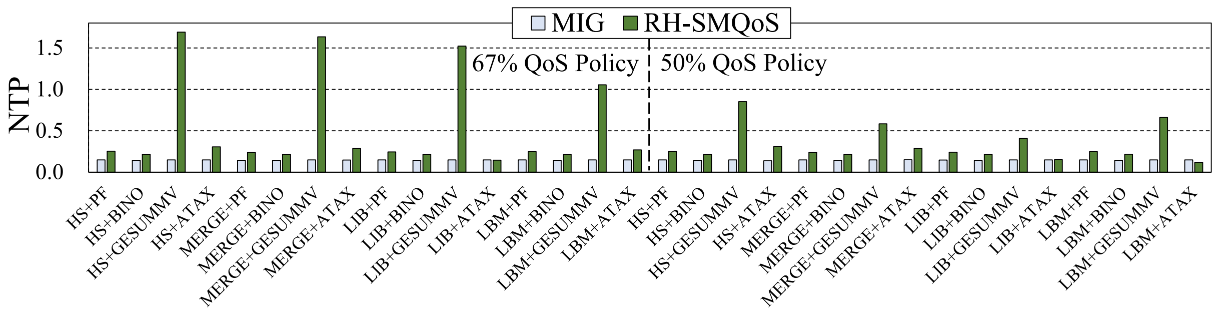


Fig. 21. The throughput of batch tasks normalized for isolated execution on a single A100 GPU.

nvprof tool with event collection [50] to collect performance metrics. This performance profiling is non-intrusive to task execution.

Fig. 17 presents the performance of each benchmark with *RH-SMQoS* normalized to *Baseline*. Besides, we have also added the task duration of *RH-SMQoS* in the figure. Note that *NPM* in this subsection is defined as $\frac{T_i^o}{T_i^p}$, where T_i^o is the task duration of original implementation (*Baseline*), and T_i^p is the task duration of transformed implementation with *RH-SMQoS*. Overall, *RH-SMQoS* achieves higher performance than *Baseline* for all tasks. Particularly for tasks sensitive to intra-SM resource contention (e.g., *HS* and *LBM*), *RH-SMQoS* achieves significant performance improvement. This is because the implementation of SMT avoids intra-SM resource contention among different tasks. Whereas for memory-intensive tasks (e.g., *LIB* and *GESUMMV*), the performance with *RH-SMQoS* is still slightly improved. The L1 cache/texture hit rate of each task with *RH-SMQoS* and *Baseline* is shown in Fig. 18. *RH-SMQoS* achieves higher L1 cache/texture hit rate than *Baseline* for most of the tasks. This is due to the technique of persistent worker adopted in *RH-SMQoS*, which achieves better locality of L1 cache.

For future GPU generations (with more SMs available), *RH-SMQoS* can launch more persistent workers to perform calculations in tasks. Besides, for the default TB scheduling method, a larger number of SMs cannot reduce intra-SM resource contention during runtime. Therefore, we believe *RH-SMQoS* could achieve higher performance improvement with future GPU architectures.

6.5. Beyond pairwise

To demonstrate *RH-SMQoS* can be applied beyond pairwise, we randomly select *PF* and *GESUMMV* as co-running batch tasks. Fig. 19 presents the normalized performance of the LS task under the co-running of these three tasks. We can observe that the results in Fig. 19 are similar to that in Fig. 15. This indicates that *RH-SMQoS* can meet QoS in the case of co-location of multiple tasks regardless of the number and characteristics of batch tasks. The reason is that *RH-SMQoS* implements strict spatial multitasking, which avoids intra-SM resource contention during concurrent execution. When the number of SMs is larger, *RH-SMQoS* can allocate more SMs to batch tasks to improve their throughput. Therefore, *RH-SMQoS* can be well adopted for future GPUs.

6.6. Comparison with MIG on NVIDIA A100 GPU

The MIG feature allows multiple GPU instances to run in parallel on a single physical A100 GPU. Since *MIG* supports limited profiles to create GPU instances, it cannot provide the stringent QoS policies (80%–95%) as the previous experiments. Instead, we select two resource allocation schemes for evaluation: the resource ratio of the LS task to the batch task is set to two (67% QoS policy) and one (50% QoS policy).

Fig. 20 presents the normalized performance of LS tasks under two QoS policies on A100 GPU. In general, the QoS_{reach} of *RH-SMQoS* and *MIG* are 50% and 34.3%, respectively. This indicates that it is increasingly difficult to guarantee QoS with static partitioning schemes for advanced GPU architectures (e.g., Ampere [10]). Therefore, the dynamic resource scheduling strategy of *SMQoS* is necessary for current and future GPU generations. Fig. 21 presents the throughput of batch tasks normalized for isolated execution on a single A100 GPU. Overall, *RH-SMQoS* achieves higher throughput in most task co-locations. The reason mainly boils down to the fine-grained resource scheduling of *RH-SMQoS* based on the multi-stream and persistent thread [51]. In addition, *RH-SMQoS* does not require physical partitioning of the GPU so that tasks with different characteristics can fully utilize the memory bandwidth.

Note that the CUDA stream and MPS can still be applied with GPU instances. Therefore, *RH-SMQoS* can be integrated with *MIG* to support complex QoS requirements under task co-locations. For instance, when there are multiple LS tasks, they can be distributed to different GPU instances without interference. On each GPU instance, *RH-SMQoS* can be applied to perform fine-grained SM allocation. In sum, *RH-SMQoS* is still effective to support QoS of LS tasks on the latest Ampere GPUs.

7. Conclusion and future work

In this paper, we propose a new runtime mechanism *SMQoS* to meet the QoS requirement for task consolidation on GPU with improved utilization and power efficiency. During runtime, *SMQoS* monitors the performance of LS tasks and dynamically adjusts the SM allocation in order to meet the QoS target. In the meanwhile, based on the mixes of the co-running tasks, *SMQoS* can either allocate more SMs to the batch tasks for higher throughput, or power gate idle SMs to reduce power consumption. Furthermore, we implement *RH-SMQoS* on real GPU hardware, which is based on the idea of *SMQoS*, to provide QoS for the LS task while improving GPU utilization. The experiment results show that *SMQoS* is more efficient than the state-of-the-art approaches to improve GPU utilization and reduce power consumption, in addition to meeting the QoS of LS tasks. Moreover, *RH-SMQoS* can meet the QoS of LS tasks with a higher throughput of batch tasks than *Volta-MPS* and *SlateQoS*. For the future work, we would like to extend *SMQoS* to support the QoS of complex GPU tasks that contain non-identical kernels with dependencies.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.parco.2022.102958.

Data availability

Data will be made available on request.

Acknowledgments

This work is supported by National Key Research and Development Program of China (Grant No. 2020YFB1506703), National Natural Science Foundation of China (Grant No. 62072018 and 61732002), and State Key Laboratory of Software Development Environment of China (Grant No. SKLSDE-2021ZX-06).

References

[1] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C.J. Rossbach, O. Mutlu, Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency, in: ACM SIGPLAN Notices, Vol. 53, (2) ACM, 2018, pp. 503–518.
[2] J. Kim, J. Cha, J.J.K. Park, D. Jeon, Y. Park, Improving GPU multitasking efficiency using dynamic resource sharing, IEEE Comput. Archit. Lett. 18 (1) (2018) 1–5.
[3] X. Zhao, Z. Wang, L. Eeckhout, Classification-driven search for effective SM partitioning in multitasking GPUs, in: Proceedings of the 2018 International Conference on Supercomputing, ACM, 2018, pp. 65–75.
[4] T. Allen, X. Feng, R. Ge, Slate: Enabling workload-aware efficient multiprocessing for modern GPGPUs, in: 2019 IEEE international parallel and distributed processing symposium (IPDPS), IEEE, 2019, pp. 252–261.
[5] P. Yu, M. Chowdhury, Salus: Fine-grained GPU sharing primitives for deep learning applications, 2019, arXiv preprint arXiv:1902.04610.
[6] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, M. Guo, Quality of service support for fine-grained sharing on gpus, ACM SIGARCH Comput. Archit. News 45 (2) (2017) 269–281.
[7] C. Nvidia, Nvidias next generation cuda compute architecture: Kepler gk110, 2012, Whitepaper (2012).
[8] NVIDIA, Sharing a GPU between MPI processes: multi-process service, 2012.
[9] T. NVIDIA, V100 GPU architecture, Technical Report, 2017.
[10] NVIDIA, NVIDIA A100 tensor core GPU architecture, 2020, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.
[11] J.T. Adriaens, K. Compton, N.S. Kim, M.J. Schulte, The case for GPGPU spatial multitasking, in: High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on, IEEE, 2012, pp. 1–12.
[12] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, M. Guo, Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing, in: High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on, IEEE, 2016, pp. 358–369.
[13] J.J.K. Park, Y. Park, S. Mahlke, Dynamic resource management for efficient utilization of multitasking GPUs, Oper. Syst. Rev. 51 (2) (2017) 527–540.
[14] Q. Chen, H. Yang, J. Mars, L. Tang, Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers, ACM SIGARCH Comput. Archit. News 44 (2) (2016) 681–696.
[15] S. Kato, K. Lakshmanan, R. Rajkumar, Y. Ishikawa, TimeGraph: GPU scheduling for real-time multi-tasking environments, in: Proc. USENIX ATC, 2011, pp. 17–30.
[16] C. Yu, Y. Bai, H. Yang, K. Cheng, Y. Gu, Z. Luan, D. Qian, Smguard: A flexible and fine-grained resource management framework for GPUs, IEEE Trans. Parallel Distrib. Syst. (2018).
[17] Z. Qi, J. Yao, C. Zhang, M. Yu, Z. Yang, H. Guan, Vgris: Virtualized GPU resource isolation and scheduling in cloud gaming, ACM Trans. Archit. Code Optim. (TACO) 11 (2) (2014) 17.
[18] Y. Ukidave, X. Li, D. Kaeli, Mystic: Predictive scheduling for gpu based cloud servers using machine learning, in: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2016, pp. 353–362.
[19] P. Aguilera, K. Morrow, N.S. Kim, QoS-aware dynamic resource allocation for spatial-multitasking GPUs, in: ASP-DAC, 2014, pp. 726–731.
[20] Q. Sun, Y. Liu, H. Yang, Z. Luan, D. Qian, SMQoS: Improving utilization and energy efficiency with QoS awareness on GPUs, in: 2019 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2019, pp. 1–5.
[21] NVIDIA, NVIDIA multi-instance GPU user guide, 2020, https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.
[22] S. Pai, M.J. Thazhuthaveetil, R. Govindarajan, Improving GPGPU concurrency with elastic kernels, in: ACM SIGPLAN Notices, Vol. 48, (4) ACM, 2013, pp. 407–418.
[23] C.J. Rossbach, J. Currey, M. Silberstein, B. Ray, E. Witchel, Ptask: operating system abstractions to manage GPUs as compute devices, in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ACM, 2011, pp. 233–248.
[24] M.S. Orr, B.M. Beckmann, S.K. Reinhardt, D.A. Wood, Fine-grain task aggregation and coordination on GPUs, ACM SIGARCH Comput. Archit. News 42 (3) (2014) 181–192.
[25] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, S. Ryu, Improving GPGPU resource utilization through alternative thread block scheduling, in: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2014, pp. 260–271.
[26] B. Wu, X. Liu, X. Zhou, C. Jiang, Flep: Enabling flexible and efficient preemption on gpus, ACM SIGPLAN Notices 52 (4) (2017) 483–496.
[27] G. Chen, Y. Zhao, X. Shen, H. Zhou, Effisha: A software framework for enabling effficient preemptive scheduling of gpu, in: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2017, pp. 3–16.
[28] Q. Chen, H. Yang, M. Guo, R.S. Kannan, J. Mars, L. Tang, Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers, in: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, 2017, pp. 17–32.
[29] W. Zhang, W. Cui, K. Fu, Q. Chen, D.E. Mawhirter, B. Wu, C. Li, M. Guo, Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters, in: Proceedings of the ACM International Conference on Supercomputing, 2019, pp. 58–68.
[30] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, M. Valero, Enabling preemptive multiprogramming on GPUs, in: Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on, IEEE, 2014, pp. 193–204.
[31] J.J.K. Park, Y. Park, S. Mahlke, Chimera: Collaborative preemption for multitasking on a shared GPU, ACM SIGARCH Comput. Archit. News 43 (1) (2015) 593–606.
[32] D. Li, M. Rhu, D.R. Johnson, M. O'Connor, M. Erez, D. Burger, D.S. Fussell, S.W. Redder, Priority-based cache allocation in throughput processors, in: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2015, pp. 89–100.
[33] E. Atoofian, A. Manzak, Power-aware l1 and l2 caches for gpgpus, in: European Conference on Parallel Processing, Springer, 2014, pp. 354–365.
[34] V. Jatala, J. Anantpur, A. Karkare, Reducing GPU register file energy, in: European Conference on Parallel Processing, Springer, 2018, pp. 77–91.
[35] S. Hong, H. Kim, An integrated GPU power and performance model, in: ACM SIGARCH Computer Architecture News, Vol. 38, (3) ACM, 2010, pp. 280–289.
[36] H. Kim, J. Kim, W. Seo, Y. Cho, S. Ryu, Providing cost-effective on-chip network bandwidth in GPGPUs, in: 2012 IEEE 30th International Conference on Computer Design (ICCD), IEEE, 2012, pp. 407–412.
[37] J. Zhao, G. Sun, G.H. Loh, Y. Xie, Optimizing GPU energy efficiency with 3D die-stacking graphics memory and reconfigurable memory interface, ACM Trans. Archit. Code Optim. (TACO) 10 (4) (2013) 24.
[38] H. Aghilinasab, M. Sadrosadati, M.H. Samavatian, H. Sarbazi-Azad, Reducing power consumption of GPGPUs through instruction reordering, in: Proceedings of the 2016 International Symposium on Low Power Electronics and Design, ACM, 2016, pp. 356–361.
[39] A. Tabbakh, M. Annavaram, X. Qian, Power efficient sharing-aware GPU data management, in: Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International, IEEE, 2017, pp. 698–707.
[40] M. Sadrosadati, S.B. Ehsani, H. Falahati, R. Ausavarungnirun, A. Tavakkol, M. Abaee, L. Orosa, Y. Wang, H. Sarbazi-Azad, O. Mutlu, ITAP: Idle-time-aware power management for GPU execution units, ACM Trans. Archit. Code Optim. (TACO) 16 (1) (2019) 3.
[41] Q. Liu, Z. Yu, The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from Alibaba trace, in: Proceedings of the ACM Symposium on Cloud Computing, ACM, 2018, pp. 347–360.

[42] K. Gupta, J.A. Stuart, J.D. Owens, A study of persistent threads style GPU programming for GPGPU workloads, in: 2012 Innovative Parallel Computing (InPar), IEEE, 2012, pp. 1–14.
[43] C. Lattner, LLVM and clang: Next generation compiler technology, in: The BSD Conference, Vol. 5, 2008.
[44] A. Bakhoda, G.L. Yuan, W.W. Fung, H. Wong, T.M. Aamodt, Analyzing CUDA workloads using a detailed GPU simulator, in: Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, IEEE, 2009, pp. 163–174.
[45] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N.S. Kim, T.M. Aamodt, V.J. Reddi, Gpuwattch: enabling energy optimizations in GPGPUs, in: ACM SIGARCH Computer Architecture News, Vol. 41, (3) ACM, 2013, pp. 487–498.
[46] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, Ieee, 2009, pp. 44–54.
[47] J.A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G.D. Liu, W.-m.W. Hwu, Parboil: A revised benchmark suite for scientific and commercial throughput computing, in: Center for Reliable and High-Performance Computing, Vol. 127, 2012.
[48] NVIDIA, NVIDIA CUDA SDK Code Samples, https://developer.nvidia.com/cuda-downloads.
[49] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, J. Cavazos, Auto-tuning a high-level language targeted to GPU codes, in: Innovative Parallel Computing (InPar), 2012, IEEE, 2012, pp. 1–10.
[50] NVIDIA, Cuda toolkit documentation, 2017, https://docs.nvidia.com/cuda/profiler-users-guide/index.html.
[51] F. Yu, D. Wang, L. Shangguan, M. Zhang, C. Liu, X. Chen, A survey of multi-tenant deep learning inference on GPU, 2022, arXiv preprint arXiv:2203.09040.