SMQoS: Improving Utilization and Energy Efficiency with QoS Awareness on GPUs

Qingxiao Sun, Yi Liu, Hailong Yang, Zhongzhi Luan, and Depei Qian
Sino-German Joint Software Institute
School of Computer Science and Engineering
Beihang University, Beijing, China, 100191
{qingxiaosun,yi.liu,hailong.yang,zhongzhi.luan,depeiq}@buaa.edu.cn

***Abstract*—Meeting the Quality of Service (QoS) requirement under task consolidation on the GPU is extremely challenging. Previous work mostly relies on static task or resource scheduling and cannot handle the QoS violation during runtime. In addition, the existing work fails to exploit the computing characteristics of batch tasks, and thus wastes 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 satisfy the QoS of latency-sensitive tasks and determine the optimal resource allocation for batch tasks to improve GPU utilization and power efficiency. The experimental results show that with *SMQoS*, 2.27% and 7.58% more task co-runnings reach the 95% QoS target than *Spart* and *Rollover* respectively. In addition, *SMQoS* achieves 23.9% and 32.3% higher throughput, and reduces the power consumption by 25.7% and 10.1%, compared to *Spart* and *Rollover* respectively.**

***Index Terms*—Graphics processing units, Quality of Service, Dynamic resource management, Throughput, Power efficiency**

I. Introduction

Graphics Processing Unit (GPU) utilizes massive Thread Level Parallelism (TLP) to provide high computing capability. Due to the continuous improvement of GPU peak performance, it is difficult for a single task to fully utilize all its computing resources. 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.

Task consolidation on GPUs has received wide attention from both industry and academia. In the industry, Hyper-Q [18] in Nvidia Kepler architecture supports concurrent execution of multiple kernels on a single GPU with multiple independent queues. Multi-Process Server (MPS) [17] 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. Moreover, the GPU schedules concurrent kernels based on the first-in-first-out (FIFO) policy, which is unable to satisfy the QoS for LS tasks.

Meanwhile, two primary mechanisms are proposed in academia to share GPU resources among co-running GPU tasks including Spatial Multitasking (SMT) [1] and Simultaneous Multikernel (SMK) [29]. SMT divides the Streaming Multiprocessors (SMs) on GPU into several disjoint subsets, each of which is assigned to one of the co-running tasks. SMK allows multiple tasks to co-run on a single SM simultaneously by switching them with time quota. QoS for LS tasks is not supported in the original design of the above two mechanisms.

978-1-7281-4734-5/19/$31.00 ©2019 IEEE

To provide QoS on GPU, existing research works can be primarily divided into two categories: *1) task and resource scheduling*. Research works in this category propose new task scheduling and resource partition methods in order to meet the QoS requirement. 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* [2] and *Rollover* [30]. The drawback of *Spart* is that the linear prediction model it adopted leads to frequent SM swapping, and thus severely deteriorates the performance. Whereas for *Rollover*, the resource contention from intra-SM, such as load-store units and L1 cache, leads to unexpected performance degradation. Moreover, it fails 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*. The specific contributions are as follows:

- 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 in order to satisfy the QoS target.
- We dynamically determine the optimal SM allocation for batch tasks 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 experimental results show that *SMQoS* can effectively improve the throughput of batch tasks and reduce system power consumption while satisfying the QoS of LS tasks.

II. Related Work

GPU Sharing Mechanisms. Existing works focus on executing multiple tasks on GPU to improve resource utilization [1], [10], [11], [13], [19], [20], [22], [24], [29], [31]. Lee et al. [13] propose mixed concurrent kernel execution that enables multiple kernels to be allocated to the same core to

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on June 30,2023 at 07:50:11 UTC from IEEE Xplore. Restrictions apply.

maximize resource utilization. Kim et al. [10] propose Pipeline Parallelism-aware CTA Scheduler to overlap the execution of dependent kernels by exploiting implicit pipeline parallelism. Spatial Multitasking (SMT) [1] enables concurrent applications to share GPUs at SM granularity, whereas Simultaneous Multikernel (SMK) [29] has finer-grained resource management than SMT, where multiple applications share a single SM. Maestro [22] combines SMT and SMK to achieve better performance for sharing GPU resources. However, all above methods fail to support QoS requirement on GPU.

QoS Management on GPU. Existing research works provide QoS guarantee on GPU can be mainly divided into two categories: *1) Extending GPU task execution model* [3], [6], [9], [23], [28], [32]. Baymax [6] provides QoS by predicting the execution time of the kernel. SMGuard [32] implements resource reservation on the SM and preempts batch tasks if the reserved resources fail to meet the QoS of LS tasks. Above approaches cannot effectively handle the performance interference during runtime. *2) Hardware extensions to the GPU* [2], [15], [21], [27], [30]. Aguilera et al. [2] propose a runtime mechanism to dynamically partition GPU resources between concurrently running applications. Wang et al. [30] propose to control the kernel execution on a per cycle basis and the amount of thread-level parallelism to meet QoS. However, all above approaches fail to exploit the computing characteristics of batch tasks for reducing power consumption.

GPU Power Saving Techniques. Hong et al. [8] propose an empirical power model that relies on dynamic power events to reduce runtime GPU energy consumption. Zhao et al. [33] propose a reconfigurable 3D die-stacking memory design that integrates wide-interface graphics DRAMs to optimize the GPU energy efficiency. Lee et al. [12] exploit model/architecture co-optimization to utilize large on-chip caches of GPUs with improved energy efficiency. Tabbakh et al. [26] propose a sharing-aware TB scheduler that assigns data sharing TBs to the same SM in order to reduce data movements. All the above works are orthogonal to our paper.

III. SMQoS Methodology

A. Design Overview

Figure 1 shows the design overview of *SMQoS*. 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 (① and ②). We add a new API call *cudaSetQoS*, which is used to specify the LS task and its QoS target. *cudaSetQoS* has two parameters: $*kernel$ and IPC_{target}. $*kernel$ is used to identify the LS task when invoking *cudaSetQoS*. In addition, IPC_{target} is used as the QoS target for LS tasks by *cudaSetQoS*. In *SMQoS*, IPC_{target} is the average IPC that the LS task needs to achieve during runtime.

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 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. 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 next epoch using our proposed SM allocation algorithm (Section III-B and III-C). 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.

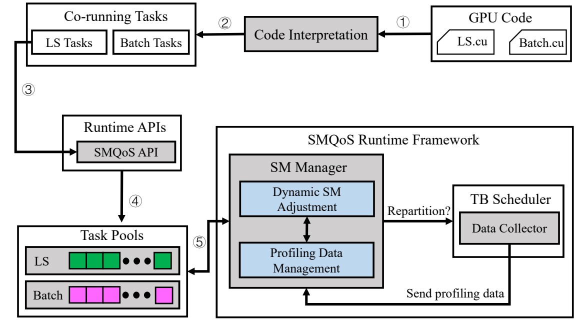

Fig. 1: The Design Overview of *SMQoS*.

B. Maintaining QoS for LS Tasks

Algorithm 1 Dynamic SM adjustment to maintain QoS.

Input: N_{epoch}, IPC_{ave}, IPC_{target}, IPC_{epoch}
Output: SM_k

1: // Calculate whether the LS task needs to swap in or swap out an SM
2: **if** $IPC_{ave} < IPC_{target}$ or $IPC_{epoch} < IPC_{target}$ **then**
3: // The LS task fails to meet QoS target
4: $kernel_{to_swapin} \Leftarrow true$
5: **else**
6: **if** $\frac{IPC_{ave} \times N_{epoch}}{N_{epoch}+1} > IPC_{target}$ and $IPC_{epoch} > IPC_{target}$ **then**
7: // The LS task will meet QoS target in the next epoch
8: $kernel_{to_swapout} \Leftarrow true$
9: **end if**
10: **end if**
11: // The LS task requires swapping operation
12: **if** $kernel_{to_swapin} == true$ **then**
13: // The LS task swaps in an SM
14: $SM_k \Leftarrow SM_k + 1$
15: **else**
16: **if** $kernel_{to_swapout} == true$ **then**
17: // The LS task swaps out an SM
18: $SM_k \Leftarrow SM_k - 1$
19: **end if**
20: **end if**

In *SMQoS*, IPC_{target} is the average IPC that the LS task needs to reach during runtime. 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

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on June 30,2023 at 07:50:11 UTC from IEEE Xplore. Restrictions apply.

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 the IPC_{ave} is the average IPC of the task, N_{epoch} is the number of epoches elapsed for the task, SM_k is the number of SMs allocated to the task, and IPC_{epoch} is the IPC of the task during current epoch. The $kernel_{to_swapin}$ and $kernel_{to_swapout}$ are two boolean type variables that specify the type of operation to be performed on the GPU task.

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 2), it immediately requires an SM to swap in (line 4). To avoid the QoS oscillation, only one SM is swapped at a time. To satisfy 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 8) 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 swaps in/out the TBs on the SM (line 11-20).

C. Determining Optimal Resource Allocation for Batch Tasks

Algorithm 2 Adaptive resource allocation for batch task.

Input: opt_k, SM_k, $upper_k$, $lower_k$
Output: SM_k
Precondition: LS task swaps out an SM

```
 1: // Determine whether the batch task needs to swap in an SM
 2: if SM_k < opt_k then
 3:     // The number of SMs allocated is less than opt_k
 4:     kernel_to_swapin ⇐ true
 5: else
 6:     if upper_k == false then
 7:         // opt_k does not reach the upper bound
 8:         kernel_to_swapin ⇐ true
 9:     else
10:         if lower_k == false then
11:             // opt_k does not reach the lower bound
12:             kernel_to_swapout ⇐ true
13:         end if
14:     end if
15: end if
16: // The batch task requires swapping operation
17: if kernel_to_swapin == true then
18:     // The batch task swaps in the SM
19:     SM_k ⇐ SM_k + 1
20: else
21:     if kernel_to_swapout == true then
22:         // The batch task swaps out an SM
23:         Power gate the SM
24:         SM_k ⇐ SM_k − 1
25:     else
26:         Power gate the SM
27:     end if
28: end if
```

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 (opt_k) allocated to the batch task during runtime. *SMQoS* introduces $upper_k$, $lower_k$, and $threshold$ to determine opt_k. The $upper_k$ and $lower_k$ are bool-type variables that indicate whether the SM allocation reaches the bounds of the optimal allocation. The $threshold$ controls the sensitivity of LS task to the SM changes. When the batch task performs the swapping operation, *SMQoS* records SM_k, opt_k, $upper_k$ and $lower_k$ as history information to determine opt_k for next epoch. The history information also includes the IPC of the task during the last epoch (IPC_{last}).

Assuming that the batch task swaps in an SM at Epoch 0, then SM_k increases by one. When Epoch 1 ends, the *SM manager* analyzes the profiling data and updates $upper_k$, $lower_k$ and SM_k: *1)* If $IPC_{epoch} \leq IPC_{last} \times (1 + threshold)$, then $upper_k = true$, in such case opt_k reaches the upper bound; *2)* If $IPC_{epoch} > IPC_{last} \times (1 + threshold)$, then $lower_k = true$ and $opt_k = SM_k$, in such case opt_k reaches the lower bound. Accordingly, assuming that the batch task swaps out an SM at Epoch 0, SM_k reduces by one. When Epoch 1 ends, the *SM manager* updates $upper_k$, $lower_k$ and SM_k: *1)* If $IPC_{epoch} < IPC_{last} \times (1 - threshold)$, then $upper_k = true$; *2)* If $IPC_{epoch} \geq IPC_{last} \times (1 - threshold)$, then $lower_k = true$ and $opt_k = SM_k$. When both $lower_k = true$ and $upper_k = true$, it indicates current SM_k is the optimal value for opt_k. To avoid the impact of frequent SM swapping of the batch tasks, the opt_k remains unchanged during the rest of the execution.

The detailed SM allocation policy for the batch task is illustrated in Algorithm 2. When the number of SMs allocated to the batch task is less than opt_k (line 2), the idle SM is allocated to the batch task (line 4). Otherwise, it determines whether the upper bound is reached (line 6). If not, the idle SM is still allocated to the batch task (line 8), otherwise it determines whether the lower bound is reached (line 10). If not, the batch task swaps out an SM (line 12). If the batch task swaps in the SM (line 17), then the TB scheduler schedules the TBs on the SM (line 19). Otherwise, the SM is power-gated to reduce power consumption (line 21-27). The *SMQoS* restricts that the GPU task can only swap in/out one SM at a time. Therefore, the value of opt_k is uniquely determined when opt_k reaches the upper and lower bounds.

IV. EVALUATION

A. Experimental Setup

To evaluate *SMQoS*, we use GPGPU-Sim v3.2.2 [4] with the simulation configuration same to [34]. Meanwhile, we rely on GPUWattch [14] to measure power consumption. We select 12 benchmarks including six computation-intensive (CI) tasks and six memory-intensive (MI) tasks with details listed in Table I. 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. We run 2M cycles for each task mix according to [30]. The length of one epoch is 10k cycles. Initially, the SMs are evenly partitioned between LS task and batch task.

We use the percentage of QoS targets that are reached (QoS_{reach}) as our metric. We use IPC to represent the QoS, and compare with two state-of-the-art works *Spart* [2] and *Rollover* [30]. The QoS_{reach} is defined as $\frac{\#of\,Sucess\,Cases}{\#of\,Total\,Cases}$. We define the QoS target (IPC_{target}) as the percentage (specified by the QoS policy) of IPC when the LS task runs

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on June 30,2023 at 07:50:11 UTC from IEEE Xplore. Restrictions apply.

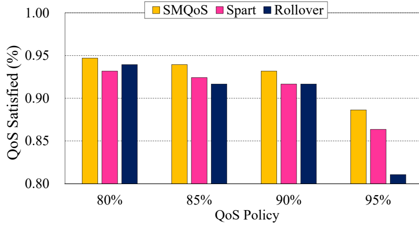

Fig. 2: The percentage of task co-runnings with QoS satisfied.

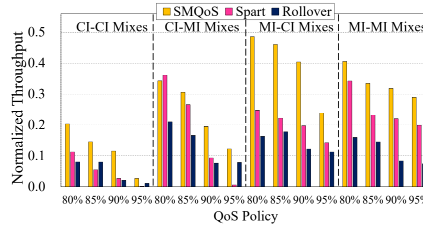

Fig. 3: Throughput normalized to isolated execution of batch tasks.

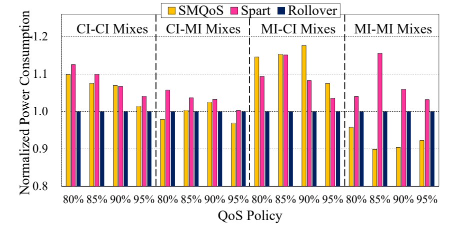

Fig. 4: Power consumption of *SMQoS* and *Spart* normalized to *Rollover*.

TABLE I: Benchmarks used for evaluation.

Benchmark	Source	Type
BinomialOptions (BINO)	CUDA SDK [16]	CI
MergeSort(MERGE)		
Coulombic Potential (CP)	Parboil [25]	
TPACF (TPACF)		
HOTSPOT (HS)	Rodinia [5]	
PATHFINDER (PF)		
ATAX (ATAX)	PolyBench [7]	MI
BICG (BICG)		
GESUMMV (GESUMMV)		
Libor Monte Carlo (LIB)	ISPASS-2009 [4]	
Lattice-Boltzmann Method (LBM)	Parboil [25]	
MRI-Gridding (MRI-G)		

isolated ($IPC_{isolated}$). The QoS policy sets the QoS target ranging from 80% to 95%, with a stride of 5%.

B. Results for Achieving QoS

In this section, we evaluate the efficiency of *SMQoS* to meet QoS target. Figure 2 shows the percentage of task co-runnings with QoS satisfied under different QoS policies. As seen, *SMQoS* achieves the highest percentage of task co-runnings with QoS satisfied across all QoS policies compared to *Spart* and *Rollover*. In general, the QoS_{reach} of *SMQoS* under all cases is 92.4%, which is higher than *Spart* and *Rollover* by 1.70% and 3.03% respectively. Especially when the QoS policy is 95%, SMQoS enables 2.27% and 7.58% more task co-runnings reach the QoS target than Spart and Rollover respectively. This demonstrates *SMQoS* is effective even when the QoS policy is tight for the LS task. The reason is that *Spart* adopts the linear prediction model that leads to frequent SM re-allocation and thus degrades the QoS. Whereas, the SM allocation in *Rollover* introduces intra-SM resource contention and thus hurts the QoS.

C. Results for Improving Throughput

Figure 3 shows the normalized throughput of batch tasks. Overall, *SMQoS* achieves the highest throughput in most task mixes, which is higher than *Spart* and *Rollover* by 10.4% and 16.5% respectively. *Rollover* performs worst due to the intra-SM resource contention. Compared to *Spart*, the proposed algorithms in *SMQoS* ensure that it can allocate more computing resources to the batch task. For instance, under the MI-CI mixes, when the QoS policy is set to 80%, *SMQoS* achieves 23.9% and 32.3% higher throughput than *Spart* and *Rollover* respectively. This is because *SMQoS* allocates more SM resources to CI batch tasks for better GPU utilization. When the QoS policy is set to 95%, the throughput degrades more significantly with *Spart* than *SMQoS*. This indicates that *SMQoS* achieves better throughput even at tight QoS target.

D. Results for Reducing Power Consumption

Figure 4 shows the power consumption of *SMQoS* and *Spart* normalized to *Rollover*. When the batch task is MI, *SMQoS* consumes less power than *Spart* and *Rollover*. Especially when the task mixes are MI-MI and the QoS policy is set to 85%, *SMQoS* consumes 25.7% and 10.1% less power than *Spart* and *Rollover* respectively. This demonstrates that when the co-running tasks are both memory intensive, *SMQoS* can efficiently reduce the power consumption while ensuring the throughput of the batch task (Section IV-C). However, when the batch task is CI, *SMQoS* achieves comparable power consumption as *Spart*, which is higher than *Rollover*. This is due to the design philosophy of *SMQoS* to increase GPU utilization whenever possible.

E. Overhead Analysis

As illustrated in Figure 1, the extra hardware introduced in *SMQoS* consists of *1)* the *DC* module that collects profiling data for each task, and *2) SM Manager* that records SM and task information, and decides the SM allocation during each epoch. In addition, two 16-bit registers are used to store IPC_{last} and IPC_{ave}. Two 8-bit registers are used to store opt_k and SM_k. Four 1-bit registers are used to store $upper_k$, $lower_k$, $kernel_{to_swapin}$ and $kernel_{to_swapout}$. A bit vector is used to specify LS tasks. In sum, the hardware overhead of *SMQoS* is acceptable.

V. Conclusion

In this paper, we propose a new runtime mechanism *SMQoS*. 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. The experimental results show that *SMQoS* is more effective than the state-of-the-art approaches.

Acknowledgment

This work is supported by National Key Research and Development Program of China (Grant No. 2016YFB1000503) and National Natural Science Foundation of China (Grant No. 61502019). Hailong Yang is the corresponding author.

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on June 30,2023 at 07:50:11 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] Adriaens, J.T., Compton, K., Kim, N.S., Schulte, M.J.: The case for gpgpu spatial multitasking. In: High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on. pp. 1–12. IEEE (2012)

[2] Aguilera, P., Morrow, K., Kim, N.S.: Qos-aware dynamic resource allocation for spatial-multitasking gpus. In: ASP-DAC. pp. 726–731 (2014)

[3] Azhar, M.W., Stenström, P., Papaefstathiou, V.: Sloop: Qos-supervised loop execution to reduce energy on heterogeneous architectures. ACM Transactions on Architecture and Code Optimization (TACO) **14**(4), 41 (2017)

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

[5] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. pp. 44–54. Ieee (2009)

[6] Chen, Q., Yang, H., Mars, J., Tang, L.: Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. ACM SIGARCH Computer Architecture News **44**(2), 681–696 (2016)

[7] Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-tuning a high-level language targeted to gpu codes. In: Innovative Parallel Computing (InPar), 2012. pp. 1–10. IEEE (2012)

[8] Hong, S., Kim, H.: An integrated gpu power and performance model. In: ACM SIGARCH Computer Architecture News. vol. 38, pp. 280–289. ACM (2010)

[9] Kato, S., Lakshmanan, K., Rajkumar, R., Ishikawa, Y.: Timegraph: Gpu scheduling for real-time multi-tasking environments. In: Proc. USENIX ATC. pp. 17–30 (2011)

[10] Kim, G., Jeong, J., Kim, J., Stephenson, M.: Automatically exploiting implicit pipeline parallelism from multiple dependent kernels for gpus. In: 2016 International Conference on Parallel Architecture and Compilation Techniques (PACT). pp. 339–350. IEEE (2016)

[11] Kim, G., Lee, M.M.J., Kim, J., Lee, J.W., Abts, D., Marty, M.: Low-overhead network-on-chip support for location-oblivious task placement. IEEE Transactions on Computers **63**(6), 1487–1500 (2012)

[12] Lee, J., Woo, D.H., Kim, H., Azimi, M.: Green cache: Exploiting the disciplined memory model of opencl on gpus. IEEE Transactions on Computers **64**(11), 3167–3180 (2015)

[13] Lee, M., Song, S., Moon, J., Kim, J., Seo, W., Cho, Y., Ryu, S.: Improving gpgpu resource utilization through alternative thread block scheduling. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). pp. 260–271. IEEE (2014)

[14] Leng, J., Hetherington, T., ElTantawy, A., Gilani, S., Kim, N.S., Aamodt, T.M., Reddi, V.J.: Gpuwattch: enabling energy optimizations in gpgpus. In: ACM SIGARCH Computer Architecture News. vol. 41, pp. 487–498. ACM (2013)

[15] Li, D., Rhu, M., Johnson, D.R., O'Connor, M., Erez, M., Burger, D., Fussell, D.S., Redder, S.W.: Priority-based cache allocation in throughput processors. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). pp. 89–100. IEEE (2015)

[16] NVIDIA: Nvidia cuda sdk code samples. https://developer.nvidia.com/cuda-downloads

[17] NVIDIA: Sharing a gpu between mpi processes: multi-process service (2012)

[18] Nvidia, C.: Nvidias next generation cuda compute architecture: Kepler gk110. Whitepaper (2012) (2012)

[19] Orr, M.S., Beckmann, B.M., Reinhardt, S.K., Wood, D.A.: Fine-grain task aggregation and coordination on gpus. ACM SIGARCH Computer Architecture News **42**(3), 181–192 (2014)

[20] Pai, S., Thazhuthaveetil, M.J., Govindarajan, R.: Improving gpgpu concurrency with elastic kernels. In: ACM SIGPLAN Notices. vol. 48, pp. 407–418. ACM (2013)

[21] Park, J.J.K., Park, Y., Mahlke, S.: Chimera: Collaborative preemption for multitasking on a shared gpu. ACM SIGARCH Computer Architecture News **43**(1), 593–606 (2015)

[22] Park, J.J.K., Park, Y., Mahlke, S.: Dynamic resource management for efficient utilization of multitasking gpus. ACM SIGOPS Operating Systems Review **51**(2), 527–540 (2017)

[23] Qi, Z., Yao, J., Zhang, C., Yu, M., Yang, Z., Guan, H.: Vgris: Virtualized gpu resource isolation and scheduling in cloud gaming. ACM Transactions on Architecture and Code Optimization (TACO) **11**(2), 17 (2014)

[24] Rossbach, C.J., Currey, J., Silberstein, M., Ray, B., Witchel, E.: Ptask: operating system abstractions to manage gpus as compute devices. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 233–248. ACM (2011)

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

[26] Tabbakh, A., Annavaram, M., Qian, X.: Power efficient sharing-aware gpu data management. In: Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International. pp. 698–707. IEEE (2017)

[27] Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N., Valero, M.: Enabling preemptive multiprogramming on gpus. In: Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on. pp. 193–204. IEEE (2014)

[28] Ukidave, Y., Li, X., Kaeli, D.: Mystic: Predictive scheduling for gpu based cloud servers using machine learning. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 353–362. IEEE (2016)

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

[30] Wang, Z., Yang, J., Melhem, R., Childers, B., Zhang, Y., Guo, M.: Quality of service support for fine-grained sharing on gpus. ACM SIGARCH Computer Architecture News **45**(2), 269–281 (2017)

[31] Xu, Q., Jeon, H., Kim, K., Ro, W.W., Annavaram, M.: Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In: Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on. pp. 230–242. IEEE (2016)

[32] Yu, C., Bai, Y., Yang, H., Cheng, K., Gu, Y., Luan, Z., Qian, D.: Smguard: A flexible and fine-grained resource management framework for gpus. IEEE Transactions on Parallel and Distributed Systems (2018)

[33] Zhao, J., Sun, G., Loh, G.H., Xie, Y.: Optimizing gpu energy efficiency with 3d die-stacking graphics memory and reconfigurable memory interface. ACM Transactions on Architecture and Code Optimization (TACO) **10**(4), 24 (2013)

[34] Zhao, X., Wang, Z., Eeckhout, L.: Classification-driven search for effective sm partitioning in multitasking gpus. In: Proceedings of the 2018 International Conference on Supercomputing. pp. 65–75. ACM (2018)

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on June 30,2023 at 07:50:11 UTC from IEEE Xplore. Restrictions apply.