IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 6, 2020, accepted March 20, 2020, date of publication March 27, 2020, date of current version April 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2983731

Toward OS-Level and Device-Level Cooperative Scheduling for Multitasking GPUs

XINJIAN LONG, XIANGYANG GONG, YAGUANG LIU, XIRONG QUE, AND WENDONG WANG, (Member, IEEE)
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications (BUPT), Beijing 100876, China
Corresponding author: Xiangyang Gong (xygong@bupt.edu.cn)

This work was partially supported by the National Natural Science Foundation of China (Grant No. 61802022 and No. 61802027), the Beijing Natural Science Foundation (No. L182034) and the Fundamental Research Funds for the Central Universities (Grant No. 2019XD-A12).

ABSTRACT As one of the most popular accelerators, the graphics processing unit (GPU) has been extensively adopted throughout the world. With the burst of new applications and the growing scale of data, co-running applications on limited GPU resources has become increasingly important due to its dramatic improvement in overall system efficiency. Quality of service (QoS) support among concurrent general-purpose GPU (GPGPU) applications is currently one of the most trending research topics. Prior efforts have been focused on providing QoS support either with OS-level or device-level scheduling methods. Each of these scheduling methods possesses pros and cons and may be unable to independently cover all the scheduling cases. In this paper, we propose a cooperative QoS scheduling scheme (C-QoS) that consists of operating-system-level (OS-level) scheduling and device-level scheduling. Our proposed scheme can control the progress of a kernel and provide thorough QoS support for concurrent applications in multitasking GPUs. Due to the accurate resource management of the copy engine and execution engine, C-QoS achieves QoS goals 23.33% more often than state-of-the-art QoS support mechanisms. The results demonstrate that cooperative methods achieve 17.27% higher system utilization than uncooperative methods.

INDEX TERMS Multitasking, parallel architectures, quality of service.

I. INTRODUCTION

Recently, major companies such as Google, Microsoft, and Tesla have adopted GPUs to boost rapid advances in burgeoning areas, such as image recognition, speech processing, natural language processing, disease detection, and autonomous driving. Not only limited to large data centers or high-performance computing (HPC) systems, such as Amazon's GPU cloud [1] and Oak Ridge National Laboratory's Summit [2], the demand for GPUs among small and medium-sized enterprises, research institutes, and universities is increasing due to its massive parallel computation capability and cost efficiency.

Modern data centers and HPC clusters commonly co-execute multiple GPU applications to improve accelerator utilization and to deal with the diurnal user access pattern. In these multitasking GPUs [3], different types of applications, such as artificial intelligence (AI), games, and video encoding and decoding, are unavoidably executing concurrently. For applications with certain quality-of-service (QoS) goals, performance below these goals may cause an unsatisfactory user experience, while performance above these goals may offer no benefits. Improving the utilization of GPU resources while guaranteeing the QoS requirements of concurrent GPGPU applications has become challenging.

A substantial amount of prior work has focused on enforcing the application's QoS and maximizing the system utilization. Researchers have modified the GPU device driver and invoked system call traps and APIs to schedule different types of GPU commands (memory copy, kernel execution, etc.) or reorder the kernels from different applications [1], [4]–[12]. These techniques are defined as OS-level scheduling methods in this work. Conversely, researchers have proposed techniques [14]–[18] to dynamically partition GPU resources to provide QoS support among concurrent applications in a spatial-multiplexed manner. These works focus on either sharing the device resources at a streaming multiprocessor (SMP) granularity or co-running multiple kernels in one single SMP. Their performance varies according to the on-chip resource partitioning strategies and the interference

The associate editor coordinating the review of this manuscript and approving it for publication was Nizar Zorba.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

among the concurrent applications [22]. These techniques are defined as device-level scheduling methods in this work.

Although these efforts may improve the QoS support for multitasking GPUs to some extent, they are subject to certain limitations and may lose efficacy in special cases. First, OS-level scheduling methods partition GPU time among the concurrent applications at the granularity of kernel execution. These techniques consider the GPU as a black box, which does cause additional manipulation of the kernel resource partition. Since the modern commodity GPUs do not support explicit preemption scheduling among the running kernels, whether the applications' QoS can be guaranteed is heavily dependent on the order that kernels are dispatched to the GPU and the kernels' execution length. This finding causes a high probability of problems for the occurrence of priority inversion, resource underutilization, etc. Second, device-level scheduling methods share the GPU hardware resources among concurrent applications in a spatial-multiplexing manner. These techniques may lose efficiency when the QoS violation is caused by factors beyond the GPU hardware (PCI-e bandwidth contention, etc.). To overcome the deficiency of applying a single scheduling method and explore the possibility of better QoS support for multitasking GPUs, we believe that a cooperative strategy of the two scheduling methods is required.

In this work, we propose a cooperative scheduling scheme (C-QoS) for the OS-level scheduling method and device-level scheduling method. This novel scheme can predict the application's slowdown due to co-location and then apply an ϵ-greedy-based algorithm. This novel algorithm exploits the merits of the two scheduling methods, and it can guarantee the concurrent GPGPU application's QoS while maximizing the system utilization. To validate the efficiency of C-QoS, we use gem5-GPU [23] as the experiment platform and select the workloads for evaluation from the Rodinia [24] and Parboil [25] benchmark sets. The results show that the proposed C-QoS achieves QoS goals 23.33% more often than the uncooperative schemes and achieves 17.27% higher system utilization.

In general, this work makes the following major contributions:

- We propose a cooperative scheduling scheme (C-QoS) using the OS-level and device-level methods, which can provide thorough QoS support for multitasking GPUs and to improve the overall system utilization.
- We propose a novel algorithm, which can make decisions on how to cooperate the two scheduling methods. These decisions are driven by the concurrent GPGPU applications' characteristics and the runtime status of the overall system.

II. BACKGROUND

In this section, we describe the technical background of the modern GPU using CUDA terminology. We introduce some of the GPU techniques according to the descriptions in prior work [26].

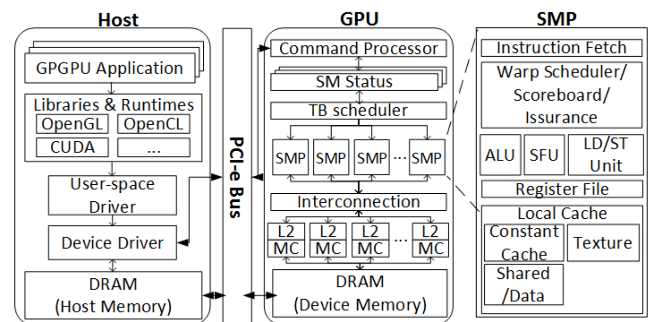
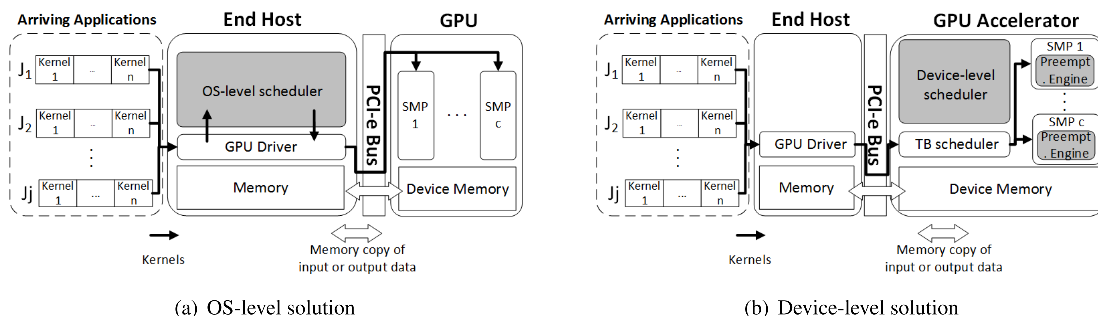


FIGURE 1. System overview.

Figure 1 shows an overview of the system provided with a GPU accelerator. Typically, GPU applications access accelerator resources using several routine steps. These steps consist of the GPU device memory initialization, copy of input data from the host memory to the GPU device memory, computation of the input data by launching some CUDA kernels, and write-back of the computation result from the GPU device memory to the host memory. CUDA kernel codes are written following a single-instruction-multiple-thread (SIMT) model. The SIMT model is an execution model that is employed in parallel computing, where single-instruction-multiple-data (SIMD) are combined with multithreading. Dependency among CUDA kernels may exist. These steps are achieved by sending GPU commands through the PCI-e bus, which are usually grouped to form GPU command groups and submitted by the GPU device driver.

The discrete GPU is connected to the host through a PCI-e bus (theoretical peak bandwidth of 16x PCI-e 3.0 bus used in the NVIDIA GPU K40 is 15,800 MB/s, and the effective bandwidth is 12,1600 MB/s). The GPU consists of thousands of CUDA cores (ALUs). Each streaming multiprocessor (SMP) contains numerous computing resources, such as CUDA cores, LD/ST units, SFUs, registers files, scratchpad memory/shared memory, and L1 cache. The number of threads that an SMP can concurrently execute (2048 in the NVIDIA Kepler architecture) is limited. A GDDR5 memory is shared among the SMPs through an interconnection network as the GPU device memory. Memory requests are distributed to the memory controller according to the addresses. A unified L2 cache is shared by all SMPs. A thread block (TB) scheduler is responsible for determining how many TBs of each kernel is allocated to a certain SMP.

A thread is the basic unit of one kernel, and threads in one kernel are hierarchically grouped into thread blocks (TB). The total TB count of one kernel, as well as the total thread count of one TB of this kernel, is specified by the programmer using variables such as GridDim and BlockDim in the code. A TB is the basic scheduling unit of the GPU hardware. The resource requirement of each kernel can be captured by the compiler before it is launched to the device. Kernels



(a) OS-level solution (b) Device-level solution

FIGURE 2. Different QoS support mechanisms in multitasking GPUs.

from different GPGPU applications consume different types of GPU on-chip resources [27]. A TB scheduler determines how many TBs from the same kernel can be dispatched to a certain SMP. One SMP can keep accepting TBs from different kernels until any type of computational resource reaches the corresponding hardware limit, and then, the SMP will leave the remaining resource unused.

For example, the total thread count of all TBs assigned to one SMP on the NVIDIA TX2 is limited to 2048 per SMP. If the TB size of the certain kernel is 384, then a maximum of 5 TBs of this kernel are scheduled and 128 threads are left idle. TBs that exceed the resource limit of the hardware will be kept waiting until the executing TBs on the SMP finish and release. When one TB is dispatched to an SMP, every 32 threads in one TB will be further grouped into a warp. A warp is the basic execution unit on the SMP. The SIMD width of the GPU hardware is 32. Warp schedulers within one SMP will select the instructions from any ready warps to execute following some kind of scheduling policy (typically greedy then oldest). This selection may greatly impact the performance of the executing kernels and indirectly affect the waiting kernels.

The only GPU architecture that claims to support preemption is NVIDIA Pascal; this technique has not been observed in its subsequent architecture, such as Volta and Turing [28]–[30]. However, no publicly available information shows the availability of software-level preemption control, which introduces extra difficulty for researchers in designing novel strategies to exploit this feature. Substantial overhead caused by context switching and context saving is a serious issue. Thus, efficient preemption support in GPU architecture remains an open problem to be solved. Recently, many works have been proposed to provide both hardware solutions and software solutions for preemptive scheduling in sharing GPUs at different levels of granularity [31], [33]–[37].

The first GPU architecture that claims to support QoS among concurrent applications is NVIDIA Volta due to the support of Volta MPS; this feature is kept in its following architecture Turing [28]–[30]. The drawback of pre-Volta MPS on QoS support has been analyzed in [7]. Volta MPS, which is introduced in the Volta and Turing architectures, enables explicit GPU computational resource allocation. This allocation renders the full spatial multitasking possible in a modern commodity GPU [30]. However, this technique is static because the resource allocation of a particular process cannot change until its end, which limits the MPS technique to handle complicated cases such as dynamic arriving kernels. Furthermore, MPS focuses on resource allocation within the GPU, which means that this technique is unable to handle the QoS violation caused by factors beyond the GPU computational resources.

III. RELATED WORKS

As shown in Figure 2, two major QoS support mechanisms exist in GPU multitasking. The first type is the OS-level scheduling method, in which the QoS manager and the corresponding data structures reside at the host side, and the arriving tasks are re-ordered and launched on the GPU according to their QoS requirements. Kato *et al.* [4] propose an event-driven real-time scheduler that exploit priority-based policies and resource reservation mechanisms. Elliott *et al.* [38] propose a synchronization-based framework that employs priority-based policies to handle resource requirements from different real-time tasks in a system equipped with multiple GPUs. Lee *et al.* [39] map concurrent applications to different SMPs on the same GPU. Chen *et al.* [7] classify and predict the duration of different GPU tasks and provide QoS support to the concurrent GPU applications by resource reservation. Ukidave *et al.* [40] exploit machine learning to identify the similarities between the arriving kernels and the running kernels and use this technique to avoid QoS violations in a GPU-equipped cluster. Zhang *et al.* [8] propose a runtime system that exploits the newly added spatial multitasking feature in a GPU and raises the accelerator utilization while achieving the latency targets for user-facing services. Zhu *et al.* [9] propose a software runtime that isolates high-priority accelerated machine learning tasks from memory resource interference. The OS-level scheduling methods commonly implement their design on a nonpreemptive accelerator design according to the modern GPU's execution model, which unavoidably makes them ineffective in some cases (priority inversion problem caused by the long-running kernels with lower priority, etc.). The time-multiplexed based design hinders these

TABLE 1. Characteristics of 10 GPGPU applications.

Bench.	*Abbr.*	*Inst* ($\times 10^9$)	*Ker*	*Reg* (*KB*)	*Smem* (*KB*)	*Griddim*	*Blkdim*	*Memcpy* (*MB*)	*KerExec* (*%*)	*Memcpy* (*%*)	*Type*
Rodinia	BFS	2.37	2	16 12	0 0	128 128	512 512	268.75	70	30	COMP.
	BP	0.90	2	11 22	1088 0	1 1	256 256	93.31	27	73	TRANS.
	HS	37.70	1	35	3072	1849	256	600	51	49	TRANS.
	SRAD	1.80	6	11 12 14 23 20 10	0 0 4096 0 0 0	1 1 1 1 1 1	512 512 512 512 512 512	0.36	58	42	TRANS.
	MD	0.30	1	45	7200	1	128	3.52	95	5	COMP.
Parboil	CUTCP	63.93	1	25	4124	121	128	0.60	96	4	COMP.
	SPMV	4.19	1	16	0	765	192	29.45	94	6	COMP.
	HISTO	0.7	4	23 14 21 20	0 0 24576 4096	64 65 84 42	512 498 768 512	11.85	91	9	COMP.
	MRI-Q	6.97	2	12 27	0 0	6 128	512 256	4.95	99	1	COMP.
	LBM	0.61	1	54	0	18000	120	520.75	50	50	TRANS.

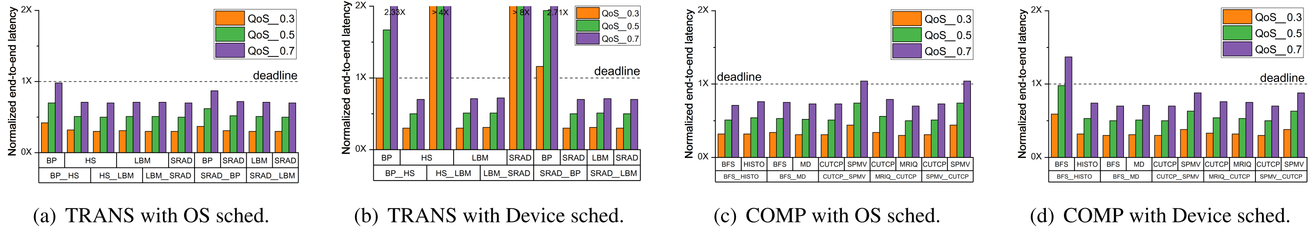
techniques' scheduling performance in the overall system utilization improvement.

The second type is the device-level scheduling method. Aguilera *et al.* [14] propose a runtime technique to dynamically allocate SMPs to the concurrent kernels. Wang *et al.* [15] propose a cycle-level mechanism for fine-grained GPU sharing (sharer kernels can partition the SMPs spatially within the GPU). Park *et al.* [22] propose dynamic resource management to discover the best-performing GPU resource partition, which exploits the works of both [31] and [34]. Song *et al.* [41] propose a framework that enables heterogeneous cores in MPSoC to make a priority-based adaptation to satisfy their diverse QoS targets. Device-level scheduling methods commonly manipulate resource usage among the concurrent GPU kernels at the hardware level. This notion makes these techniques, which exclusively rely on refining intra-GPU resource adjustment to guarantee the applications' performance, become inefficient when the causes of QoS violations surpass the GPU hardware resources.

IV. MOTIVATION

To illustrate the above problem, we perform several evaluations of the modified gem5-GPU using workloads chosen from Rodinia and Parboil benchmark sets. These workloads are chosen based on their characteristics. Table 1 lists the benchmarks that we employed in this paper. Bench indicates the benchmark set to which the particular benchmark belongs. Abbr indicates the abbreviation name of each benchmark. Inst indicates the number of each benchmark's simulated instructions. Ker indicates the number of kernels of the particular benchmark. Reg and Smem indicate the number of the register files and the shared memory consumed by the specific kernel, respectively. Griddim and Blkdim indicate the number of the TBs and the number of threads in each TB of the corresponding kernel of the particular benchmark, respectively. Memcpy indicates the total amount of data migrated between the host and the device during the particular benchmark's lifetime. KerExec and Memcpy indicate the percentage of simulated cycles consumed in the kernel execution and the data migration of each benchmark, respectively. Type indicates the type of the particular benchmark in this study, which will be explained in the following section.

From Table 1, we can observe that different applications demand different resources, and different percentages of duration are spent on kernel execution tasks and memcpy tasks in each benchmark. We define the benchmarks whose percentage of memcpy tasks is not less than 40% as the 'transfer-intensive' applications (TRANS) in this study. We define benchmarks whose percentage of kernel execution tasks exceeds 60% as the 'compute-intensive' application (COMP). Since the original scale of the workloads is small and most of them vary drastically, we repeatedly execute the smaller workloads and rein the number of the simulated instructions of all the benchmarks within two orders of magnitude. Note that no explicit cudaMemcpy exists in the recent GPU programs that adopt the unified memory technique [42]. This technique automatically syncs data and releases users from the data copying. However, memory copies still occur under the hood, and we believe that scheduling towards data transfer between CPU system memory and GPU device memory is still necessary. Furthermore, the complicated page fault mechanism of unified memory may introduce some extra overhead, which could be destructive in QoS guaranteeing. We select and reproduce the algorithms adopted in the previous work (Baymax [7], Spart-QoS [14]) according to their description and exploit them as the 'OS-level scheduling method' and 'device-level scheduling method' in this study. The QoS goal is defined as multiple times of the solo end-to-end latency of each GPGPU application. The concept of the end-to-end latency is derived from [7]. We assume that the scale of the workloads of all the applications is known when they arrive at the system. The performance of the particular application when it runs in isolation is defined as its 100% QoS goal ($QoS_1.0$). In this study, a different level of QoS constraint will be used to simulate the difficulty of guaranteeing the particular application's QoS in different cases. For instance, $QoS_0.5$ indicates that the application's QoS goal equates to the performance when it is running with 50% IPC compared with its isolated run, which obtains $QoS_0.5 = \frac{QoS_1.0}{0.5}$. As described in previous work [43], the differences between each application's isolated performance and its QoS requirement vary vastly due to their variation in scalability, which corresponds to the increase in the thread-level parallelism (TLP). The definition



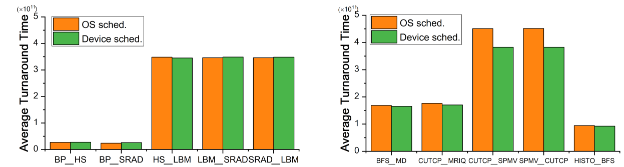
(a) TRANS with OS sched. (b) TRANS with Device sched. (c) COMP with OS sched. (d) COMP with Device sched.

FIGURE 3. **QoS violation with one single scheduling method.**

of the QoS level in this study is similar to that in previous work [14], [15].

Figure 3 shows the QoS violation when two TRANS applications or two COMPs applications are collocated using OS-level scheduling or device-level scheduling method, respectively. We synthetically generate sequences of arriving kernels and inter-arrival times for each experiment. The inter-arrival time differences in each sequence follow an exponential distribution. The bottom row of the x-axis of Figure 3 indicates the combination of the applications of the Rodinia and Parboil benchmark sets, and the left application indicates the first set that arrived at the system in each pair. The y-axis indicates the end-to-end latency of each application normalized to its QoS target (QoS_0.3, QoS_0.5, QoS_0.7).

As shown in Figure 3 (a) and (b), the number of QoS-guaranteed TRANS applications using the OS-level scheduling method (100% on average) outclasses those using the device-level scheduling method (60% on average). When multiple transfer-intensive applications are co-located in the system, a large number of memcpy requests will be generated and cause a serious contention in the GPU's copy engine and PCI-e bandwidth. In this case, hardware resource reallocation provided by device-level scheduling is not capable of handling the problem, and it can only cause a slight difference in the QoS violation. As shown in Figure 3 (c) and (d), the number of QoS-guaranteed COMPs applications using the device-level scheduling method (90% on average) is slightly more than those using the OS-level scheduling method (80% on average). An OS-level scheduling method can obtain information and classify different GPU tasks and then apply specific policies to both the kernel execution tasks and the memcpy tasks. Compared with the scheduling for COMPs applications using the device-level method, this capability mitigates the gap in adaptability between COMPs applications and the OS-level scheduling method. However, since OS-level scheduling cannot operate resource reallocation to the tasks launched on the GPU, its effect on GPU resource utilization improvement is inferior to the device-level scheduling method. For instance, as described in Section VII-C, the device-level scheduler tends to minimize the difference in the dominant resource use in each scheduling period. When a combination of cutcp and spmv is located in the GPU, spmv's kernels have a higher probability of being processed due to its zero shared memory usage compared with cutcp's, even when spmv's kernel arrives at the device later than



(a) TRANS with OS sched and Device sched. (b) COMP with OS sched and Device sched.

FIGURE 4. **Average turnaround time of concurrent GPGPU app pairs with one single scheduling method.**

cutcp's. According to the largest number of TBs that each application can launch in the same SMP (spmv:16, cutcp:2), we can explain the results in Figure 3(c)(d) and Figure 4(b). When the accelerator is shared by spmv's and cutcp's kernels using the OS-level scheduling method, spmv's kernel is very likely blocked by cutcp's, which causes spmv's QoS violation when the QoS constraint becomes strict (0.7). Conversely, adopting the device-level scheduling method helps spmv's average turnaround time experience an obvious acceleration due to faster processing, which also only causes a negligible effect on cutcp's kernels without violating their QoS. Compared with the benchmark pair using the OS-level scheduling method, Figure 4 (b) shows a 11.16% speedup in turnaround time on average using the device-level scheduling method. In this work, we consider turnaround time as the metric for measuring the scheduling method's effect on system utilization. For the same amount of workloads, we believe that the shorter is the turnaround time, the more efficient are the system resources utilized by the concurrent GPGPU applications. Note that QoS for more fine-grained sharing of GPU hardware resources has already been proposed in [15]. Furthermore, a more advanced solution [16] has been proposed to solve the application's slowdown, which is caused by the contention on shared resources. However, these techniques still possess the major limitation of the device-level scheduling method, and we believe that the application of these techniques will not cause a significant change in the results of this evaluation.

As shown in Figure 3 and Figure 4, the single scheduling method is insufficient to provide thorough QoS support to the concurrent GPGPU applications while improving the system utilization. We join the OS-level scheduling method with the device-level scheduling method and expect this combination to help the two scheduling methods be complementary.

Unfortunately, as described in Section VII-C, this simple combination may either enable no improvement or cause performance degradation of the concurrent GPU applications' performance, which is caused by the potential conflict of the two scheduling method's decisions. This finding enlightens us that cooperation is necessary between OS-level scheduling and device-level scheduling. As shown in Section VII-C, the result proves that the cooperative scheduling method is a promising direction for providing better QoS support for multitasking GPUs. However, how to cooperate the two scheduling methods rather than making an exclusive choice between them remains a challenge, which is caused by the complicated interference between the copy engine and the execution engine and the asynchrony of scheduling occurrence. To fully use the merits of the two scheduling methods, we further propose C-QoS to jointly manipulate the two scheduling methods to improve the performance of this cooperative method, and we aim to provide a more thorough QoS support for concurrent GPU applications.

V. C-QoS METHODOLOGY

Two major bottlenecks of the concurrent GPU applications' performance exist in the heterogeneous system equipped with GPUs. These bottlenecks hinder the efficiency of the prior QoS support based on the OS-level or device-level scheduling methods. The first kind of bottleneck is the concurrent application's performance degradation, which is caused by the queuing delay and PCI-e bandwidth contention. Co-located applications contend for the limited PCI-e bandwidth when transferring data between the host and the accelerator. This contention substantially affects the latency-sensitive application's QoS and is analyzed in [7]. Rapid increases in GPU computational capability further shift the bottleneck of GPU applications' performance towards communication. This issue is recently recognized and investigated in areas such as distributed deep learning or parallel deep neural networks [44]. We cast this bottleneck as the OS-level bottleneck in this study. The second kind of bottleneck is the application's slowdown at co-location, which is caused by the reduction in the computational resources' assignment and the contention on the shared GPU resources, such as GPU L2 cache and GPU's device memory bandwidth. As demonstrated in [45], a kernel's scalability, sensitivity to resource contention, and pressure on shared resources strongly affect its slowdown at a co-location. Similarly, we refer to this bottleneck as the device-level bottleneck.

To resolve the two bottlenecks, we propose C-QoS (Figure 5), which is a holistic approach to guarantee the co-located GPU applications' QoS while maximizing the device utilization. This scheme has four design guidelines.

- C-QoS should be able to predict the end-to-end latency of each arriving kernel. In this case, this scheme can quantify the impact of each bottleneck on the particular kernel's turnaround time and determine what scheduling strategy should be operated.

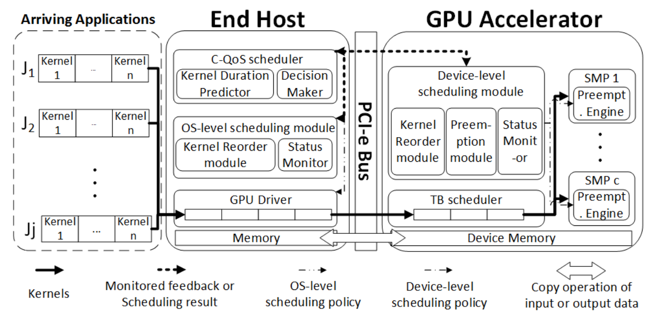


FIGURE 5. Overview of C-QoS.

- C-QoS should be able to manage the kernel's launching order following a specific strategy, instead of handling them in a FIFO manner. In this case, the OS-level scheduling method is operated and the QoS violation caused by queuing delay and PCI-e contention may be alleviated.
- C-QoS should be able to manage the GPU hardware resource partition among the concurrent applications, and C-QoS should be able to operate preemption scheduling at a specific granularity. In this case, the device-level scheduling method is operated and the QoS violation caused by the long-running kernels may be alleviated.
- C-QoS should be able to monitor the runtime status of the GPU's copy engine and the GPU's execution engine. In this case, C-QoS can create a specific strategy to apply in the next scheduling period.

VI. SCHEDULING PROBLEM ANALYSIS

In this section, we first model how the GPGPU applications access the GPU computational resources and the PCI-e bandwidth in GPU-accelerated computing, and we define the metrics to measure the QoS guarantees and the utilization. Then, we state the problem of how to leverage the two scheduling methods to handle the concurrent CUDA kernels.

A. PROBLEM STATEMENT

We assume that the host's resources are sufficient and that the accelerator resources and the PCI-e bandwidth are the major bottlenecks in this study. We consider one host connected with one discrete GPU accelerator using a PCI-e bus as the target platform. We use the set $S = \{s_1, s_2, \cdots, s_{|S|}\}$ to denote the SMPs within one GPU. The GPGPU applications may arrive at the platform dynamically and multiple CUDA kernels of different applications may exist to simultaneously access the accelerator resources. We use a to denote one GPGPU application and the set $A = \{a_1, a_2, \cdots, a_{|A|}\}$ to denote a sequence of applications with a negligible inter-arrival time. Furthermore, we employ k_i to denote one CUDA kernel of application i and the set $K_i = \{k_{i1}, k_{i2}, \cdots, k_{i|K_i|}\}$ to denote the application i's all kernels. Similarly, we utilize tb_{ij} to denote one TB of kernel j and the set

$TB_{ij} = \{tb_{ij1}, tb_{ij2}, \cdots, tb_{ij|TB_{ij}|}\}$ to denote kernel j's all TBs. Because the QoS requirement of each GPGPU application is different, we use d_i to denote the desired deadline of application i. As described in Table 1, the characteristics vary among different kernels even in the same application. Thus, for kernel j of application i, we use $gdim_{ij}$ and $bdim_{ij}$ to denote its Griddim and Blkdim, respectively, and we use reg_{ij} and $smem_{ij}$ to denote its requirements of registers and shared memory, respectively. We employ in_{ij} and out_{ij} to denote the size of data of this kernel migrated between the host and the GPU device.

In any scheduling period, any kernels' use of GPU hardware resources and the PCI-e bandwidth should not exceed the hardware constraints of the accelerator and the PCI-e bus. The definition of scheduling period is described in Section VI-B. We assume a total of P scheduling periods and that data migration caused by the memcpy tasks is transferred from/to pinned memory, which means that the GPU's copy engine will be accessible to one kernel exclusively in any scheduling period. Furthermore, the total usage of both the GPU computational resources and the PCI-e bandwidth in all the scheduling periods should not exceed the particular kernel's requirement. These assumption are formulated in (1).

$$\begin{cases} \sum_{i=1}^{|A|}\sum_{j=1}^{|K_i|} r_{ijp} \leq R \quad \forall p \in \{1, \cdots, P\}, \\ bw_{ijp} \leq BW \quad \forall p \in \{1, \cdots, P\}, \\ \sum_{i=1}^{|A|}\sum_{j=1}^{|K_i|}\sum_{p=1}^{P} r_{ijp} \leq r_{ij}, \\ \sum_{i=1}^{|A|}\sum_{j=1}^{|K_i|}\sum_{p=1}^{P} data_{ijp} \leq data_{ij}. \end{cases} \tag{1}$$

r_{ijp} denotes the allocation of one particular type of resource of kernel j of application i in the p-th scheduling period. In theory, r is able to be generalized to any type of GPU hardware resources (e.g., local memory per thread). As described in Section V, we aim to predict the end-to-end latency of each arriving kernel before their execution to facilitate further scheduling. Thus, we define $r \in \{gdim, bdim, reg, smem\}$ because these four kinds of resources are allocated during TB dispatch of a kernel [7], and this information is the only data that we can obtain before the kernel execution [34]. R denotes the maximum of the corresponding type of hardware resource in one GPU. Similarly, bw_{ijp} denotes the effective PCI-e bandwidth of kernel j of application i in the p-th scheduling period. BW denotes the theoretical peak bandwidth of the PCI-e bus. r_{ij} denotes the actual requirement of one of the previously mentioned resources of kernel j of application i. Furthermore, $data_{ijp}$ denotes the size of the migrated data of kernel j of application i in the p-th scheduling period in one particular direction, and $data_{ij}$ denotes the total size of the data of the corresponding migration. We define $data \in \{in, out\}$.

Note that modeling concurrent kernel execution is challenging because the execution details are complicated, and the interaction of interference of different resources further exacerbates the difficulty. To describe the time cost of each kernel with OS-level scheduling and device-level scheduling in multitasking GPUs, we extend the concurrent task execution model described in [51] and use this model to predict the completion time of each arriving kernel. The end-to-end latency of one CUDA kernel is composed of two parts: the duration of its computational tasks and the duration of its memcpy tasks from the host to the accelerator device or from the accelerator device to the host. We use t_{ij} to denote the entire time to concurrently finish kernel j of application i, and t_{ij} can be calculated as (2). t_exc_{ij} represents the completion time of kernel j's computational tasks, and t_cpy_{ij} represents the completion time of kernel j's memcpy tasks. t_ol_{ij} represents the duration when the computational tasks and memcpy tasks are running an overlapping pattern.

$$t_{ij} = t_exc_{ij} + t_cpy_{ij} - t_ol_{ij}. \tag{2}$$

We first extend the model in [51] with the awareness of a scheduling period. The completion time for kernel j's computational tasks on non-preemptive GPUs, which are denoted by t_exc_{ij}, can be calculated in (3). In the equation, t_s_{ij} represents the duration of the kernel j submitted by the application i to the task queue in the TB scheduler from the GPU device driver (as shown in Figure 5). t_q_{ij} represents the queueing delay of kernel j waiting to arrive at the head of the task queue, and t_w_{ij} represents the duration of kernel j waiting for free SMP at the head of the task queue. As described in [51], t_s_{ij} and t_q_{ij} are collected directly via profiling, and t_w_{ij} is calculated by keeping track of when SMPs are available. t_r_{ij} represents the duration of kernel j running on the accelerator device, which can be further calculated as the sum of the running time of kernel j of each scheduling period in (4). t_sk_{ij} represents the running time of kernel j collected from a single kernel execution profile. R_{tb} and R_{bw} represent the maximum number of TB supported on one SMP and the peak bandwidth supported on the PCI-e bus. TB_{ijp} represents the number of the TBs of kernel j that arrived at the head of the task queue in the p-th scheduling period. occ_{ijp} represents the occupancy of kernel j in the p-th scheduling period, which is the ratio of active TBs to the maximum number of TBs supported on one SMP. occ_{ijp} is limited by the four kinds of resources ($gdim$, $bdim$, reg, and $smem$) allocated during TB dispatch of kernel j. The number of ready TBs and the kernel's occupancy may differ as time elapses. Following the same parameters in all the scheduling periods may cause an underestimation or overestimation of the kernel's running time.

$$t_exc_{ij} = t_s_{ij} + t_q_{ij} + t_w_{ij} + t_r_{ij}. \tag{3}$$

$$t_r_{ij} = \sum_{p=1}^{P}(t_sk_{ij} \times \left\lceil \frac{TB_{ijp}}{R_{tb} \times occ_{ijp} \times |S|} \right\rceil / \min\{1, \frac{R_{bw}}{bw_{ijp} \times \min\{1, \left\lceil \frac{TB_{ijp}}{R_{tb} \times occ_{ijp} \times |S|} \right\rceil\} \times |S|}\}). \tag{4}$$

When device-level scheduling methods become available, the preemption mechanism is supported to handle problems such as priority inversion and improve the system throughput. We choose SM-draining [31] as the preemption technique, and the preemption overhead is defined as the duration of one TB of the preempted kernel. Thus, (5) and (6) calculate the new completion time of the kernel that causes preemption and the completion time that is being preempted respectively, as denoted by t_excp_{ij} and t_excbp_{ij}. When preemption occurs, the TBs of the preempted kernel will not be scheduled for the SMPs. When the running TB is finished, the SMP will be freed and occupied by the new kernel. In this case, the waiting time of the preempting kernel will be 0. Assuming $|K'|$ preempted kernels in one preemption, and t_pi represents the preemption overhead that corresponds to different preempted kernels. The preempted kernel will be restored as soon as the preempting kernel is finished, and then, the completion time of the preempted kernel will be calculated as the sum of its self-completion time and the preempting kernel's completion time.

$$t_excp_{ij} = t_s_{ij} + t_q_{ij} + t_r_{ij} + \min\{t_p1, t_p2, \cdots, t_p\,|K'|\}. \quad (5)$$

$$t_excbp_{ij} = t_s_{ij} + t_q_{ij} + t_w_{ij} + t_r_{ij} + t_excp. \quad (6)$$

Conversely, the duration of each memcpy task from kernel j can be calculated as (7). We assume that each kernel consumes all the effective bandwidth when they transfer data through the PCI-e bus. Thus, the bandwidth cannot be shared among the concurrent memcpy tasks, and each of them has the same priority. t_ol_{ij} is determined by keeping track of both SMP and the PCI-e bus. When these two types of resources are available and ready tasks exist at the head of the task queue, a computational task or memcpy task will be launched; these launches are independent.

$$t_cpy_{ij} = t_s_{ij} + t_q_{ij} + t_w_{ij} + \sum_{p=1}^{P} \frac{data_{ijp}}{bw_{ijp}} \quad \forall data \in \{in, out\}. \quad (7)$$

Now, we discuss the QoS support in multitasking GPUs. We define the set $X_i = \{x_{i1}, x_{i2}, \cdots, x_{i|K_i|}\}$ to denote the decision for the GPGPU application i to arrange GPU computational resources and PCI-e bandwidth to the CUDA kernel set K_i in multitasking GPUs. We use $Q_p(X_i)$ to denote how well the QoS goal of applications i is guaranteed when a scheduling decision set X_i is applied, and $Q_p(X_i)$ can be calculated as (8). As described in Section IV, the QoS goal is defined as multiple times of the solo end-to-end latency of each GPGPU application. We define the value $n \in \{0, 1\}$ to denote whether the application i's QoS goal is guaranteed when the scheduling decision X_i is applied. n and the predicted entire time t_{ij} may vary according to different X_i. t_{i0} represents the time when the first kernel of application i is submitted to the GPU driver. Assume that two different scheduling decisions with the same n value exist in one scheduling period. In this case, we calculate the time headroom of applying X_i as $\sum_{j=1}^{|K_i|}(1 - \frac{t_{ij}(X_i)}{d_i - t_{i0}})$ and use this value to differentiate the two different decisions with the aim of maximizing this metric to allow for more GPU time shared among different concurrent kernels. The range of $Q_p(X_i)$ is [0,2).

$$Q_p(X_i) = n_p(X_i) + \frac{n_p(X_i)}{|A| \cdot |K_i|} \times \sum_{j=1}^{|K_i|}(1 - \frac{t_{ij}(X_i)}{d_i - t_{i0}}). \quad (8)$$

We discuss the utilization of accelerator hardware in multitasking GPUs. As the amount of computational resources that GPU incorporates increases, it becomes increasingly difficult for CUDA kernels to fully utilize the vast GPU resources, which always causes a resource underutilization problem. Throughout this paper, we focus on the hardware resources utilization within the SMP, whose improvement may cause an increase in TLP and higher GPU throughput. We use $U_p(X_i)$ to denote the hardware resource utilization of all the applications in sequence A when X_i is applied. We use dr_{sp} to denote the dominant resource share of the s-th SMP in the p-th scheduling period, and DR_{sp} represents the maximum of the corresponding hardware resource. The dominant resource is originally proposed for job scheduling in clusters [32]. The intuition of this metric is that multiresource allocation should be determined by the maximum share that a CUDA kernel requires of any resource. In consecutive scheduling periods, we calculate $1 - \frac{dr_{sp} - dr_{s(p-1)}}{DR}$ to minimize the difference between the minimum and the maximum of the dominant resource share in each SMP. We calculate this value with the aim of dispatching more TBs to the GPU. We assume that the four types of hardware resources within the SMP share the same weight during the kernel dispatch. Furthermore, we use $\frac{1}{4|K_i|}$ and $\frac{1}{|S|}$ to ensure that the range of $U_p(X_i)$ is the same as $Q_p(X_i)$.

$$U_p(X_i) = \frac{1}{4\,|K_i|} \times \sum_{r \in \{gdim, bdim, reg, smem\}} \sum_{j=1}^{|K_i|} \frac{r_{ijp}}{R} + \frac{1}{|S|} \times \sum_{s=1}^{|S|}(1 - \frac{dr_{sp} - dr_{s(p-1)}}{DR}). \quad (9)$$

We use $C(X_i)$ to denote the cost of applying scheduling decision X_i as

$$C_p(X_i) = \frac{1}{4\,|K_i|} \times \sum_{r \in \{gdim, bdim, reg, smem\}} \sum_{j=1}^{|K_i|}(\frac{r_{ijp}}{R} + \frac{bw_{ijp}}{BW}). \quad (10)$$

The range of $C_p(X_i)$ is (0, 2]. We use the function $O_{ip}(X_i)$ to denote the utility of the application i's performance when choosing a strategy (X_i) in the p-th scheduling period as

$$O_p(X_i) = \frac{a \times Q_p(X_i) + b \times U_p(X_i)}{C_p(X_i)}. \quad (11)$$

a and b are the weighted factors to adjust the scheduling performance on both the QoS support and system utilization among the concurrent GPGPU applications. As we consider

QoS support as the first aim in each scheduling, we set a as 10 and b as 1 throughout this paper.

B. C-QoS SCHEDULING STRATEGY

The C-QoS scheduler aims to obtain the optimal cooperative strategy for the two scheduling methods to maximize the number of QoS guaranteed applications, and it also aims to improve the utilization of the overall system as much as possible. Specifically, given the C-QoS scheduling strategy X_i, the C-QoS scheduler can derive the optimal strategy by solving the problem as

$$\max \quad \sum_{i=1}^{|A|} \sum_{p=1}^{P} O_p(X_i)$$
$$s.t. \quad Eq.1. \tag{12}$$

We develop a dispatch algorithm, termed C-QoS scheduling strategy, which is a greedy heuristic of the multidimensional multiple-choice knapsack problem (MMKP). The details of the proposed strategy are shown in algorithms 1 and 2. The 'ready task' in the following description indicates the task whose state is 'ready'. C-QoS scheduling is triggered when 1. a new application arrives; 2. a memcpy task for the host to device direction completes; 3. a kernel execution task completes; 4. a memcpy task in the device to host direction completes; and 5. an application terminates. The gap between two consecutive previously mentioned events is defined as one scheduling period in this study.

According to the definition of MMKP, each SMP in one GPU is considered one knapsack. The hardware limits of 5 types of resources (as described in (1)) are considered 5 different capacities of one knapsack. We take each set of the partitions of the 5 types of resources as one category of items. Each scheduling decision, which consists of one partition set and one launching order of memcpy tasks, is considered one item in the particular category. We consider the numerator of (11) as the value of one item, and we consider the denominator of (11) as its cost. As described in algorithm 1 line 4 and line 20, we calculate O_p as (11) for the assumptions of dispatching each ready kernel to the GPU, and we use these values to allocate computational resources and PCI-e bandwidth. We assume that the performance and the completion time of all the benchmarks can be profiled or predicted in advance.

As assumed in Section VI-A, the PCI-e bus cannot be shared or preempted among the concurrent CUDA kernels. Thus, C-QoS scheduling towards the new arriving kernels, which have not been submitted to the GPU, will not start until the PCI-e bus is idle (line 1). By leveraging the OS-level scheduling methods, we can reorder the arriving kernels regardless of how they are submitted by the concurrent GPGPU applications. Thus, we traverse Kd_p and pop one kernel k_j, termed the candidate kernel, from Kd_p each time (lines 2-3), and assume that this selected kernel will be launched and occupy the PCI-e bus in the p-th scheduling period (line 4). We join Kd_p and Kr_p into K_p (line 5). We traverse K_p and

---

Algorithm 1 C-QoS Scheduling Algorithm Part a

Input:
The set of kernels located in the GPU driver's task queue in the p-th scheduling period: $Kd_p = \{kd_{p1}, kd_{p2}, \cdots kd_{p|Kd_p|}\}$.
The set of kernels waiting or running on the GPU in the p-th scheduling period: $Kr_p = \{kr_{p1}, kr_{p2}, \cdots kr_{p|Kr_p|}\}$.
The candidate kernel: k_j.

Output:
The scheduling decision: X_i.

```
1:  if the PCI-e bus is idle then
2:    for j = 1 → |Kd_p| do
3:      Pop k_j from Kd_p
4:      Assume that k_j is launched
5:      K_p = Kd_p ⋃ Kr_p
6:      for j' = 1 → |Kd_p| + |Kr_p| − 1 do
7:        Calculate O_{p+1}(X_{i'}) according to k_{j'}
8:        if k_j is ready to be submitted to TB scheduler then
9:          Calculate O^{pot}_{p+2}(X_i) according to k_j
10:         O_{p+1}(X_{i'}) += O^{pot}_{p+2}(X_i)
11:       end if
12:       Push O_{p+1}(X_{i'}) into dlist
13:     end for
14:     Push k_j into Kd_p
15:   end for
16: end if
17: for j = 1 → |Kr_p| do
18:   Execute Algorithm2 and update dlist
19: end for
20: Obtain X_i according to dlist
```

---

calculate $O_{p+1}(X_{i'})$ for each kernel in the set, and we calculate this value to determine how the decision of dispatching k_j will impact the QoS support and resource utilization of other concurrent kernels (line 6-7).

Note that the interaction of performance interference caused by the allocation of different resources is complicated. Even if we apply the same device-level scheduling decisions in two distinct experiments with the same set of kernels, the final results may differ if we apply different OS-level scheduling decisions in any of the scheduling periods. Compared with the simple cooperation that separates the two scheduling methods, as described in Section VII-C, this complicated interference should be considered if we aim to leverage them jointly. Thus, we introduce $O_p^{pot}(X_i)$, termed the potential value, in our proposed algorithm. This value is calculated as 11, which is similar to $O_p(X_i)$. The only difference between $O_p(X_i)$ and $O_p^{pot}(X_i)$ is that the potential value is 'a prediction after a prediction'. As described in lines 8-10, if the candidate kernel k_j is ready to be dispatched to the SMPs in the $(p+2)$-th scheduling period, then we will calculate $O_{p+2}^{pot}(X_i)$ for k_j based on the previous assumption of launching k_j in the $(p+1)$-th scheduling period. The aim of calculating $O_{p+2}^{pot}(X_i)$ is to determine how k_j will impact the concurrent kernels' QoS support and resource utilization

Algorithm 2 C-QoS Scheduling Algorithm Part B

Input:

The set of kernels located in the GPU driver's task queue in the p-th scheduling period: $Kd_p = \{kd_{p1}, kd_{p2}, \cdots kd_{p|Kd_p|}\}$.

The set of kernels waiting or running on the GPU in the p-th scheduling period: $Kr_p = \{kr_{p1}, kr_{p2}, \cdots kr_{p|Kr_p|}\}$.

The candidate kernel: k_j.

Output:

The list of scheduling results: *dlist*.

```
1:  Do profile run in 2^|S| possibilities and obtain n
2:  for i = 1 → n do
3:      for j = 1 → |Kr_p| do
4:          Pop k_j from Kr_p
5:          Assume that k_j is launched according to the possibility i
6:          K_p = Kd_p ∪ Kr_p
7:          for j' = 1 → |Kd_p| + |Kr_p| − 1 do
8:              Calculate O_{p+1}(X_i') according to k_j'
9:              if k_j is ready to occupy the PCI-e bus then
10:                 Calculate O^pot_{p+2}(X_i) according to k_j
11:                 O_{p+1}(X_i') += O^pot_{p+2}(X_i)
12:             end if
13:             Push O_{p+1}(X_i') into dlist
14:         end for
15:         Push k_j into Kr_p
16:     end for
17: end for
18: Return dlist
```

after occupying the PCI-e bus, which is assumed to be non-preemptive in this study. Obviously, determining how many new kernels are going to be submitted to the GPU driver in the $(p + 2)$-th scheduling period in advance is not feasible. Thus, we calculate $O^{pot}_{p+2}(X_i)$ based on the assumption that new kernels will not arrive in the $(p+2)$-th scheduling period. We input the results on a list and start the next iteration (line 12).

Lines 17-19 describe the C-QoS scheduling towards the kernels running or waiting on the GPU in the p-th scheduling period. Before describing algorithm 2, we define two states for each SMP in the GPU: *hold* and *changed*. *hold* indicates that the SMP will remain idle or be occupied by the same kernels before scheduling, while *changed* indicates that the SMP will be occupied by k_j. Thus, $2^{|S|}$ possibilities of k_j's dispatching will exist in theory if k_j is sufficiently large to consume all the SMPs. We filter the impractical possibilities by calculating the number of SMPs required by k_j and comparing it with the number of the accessible SMPs. Furthermore, we eliminate the possibilities that may cause redundant scheduling. In most cases, the time complexity of handling all scheduling options remains too high after our filtering. With advances in technology scaling, the growing number of SMPs and emergence of new hardware resources within the SMPs [30] may further deteriorate this problem. Thus, we

TABLE 2. Simulation parameters.

CPU Param.	*Value*	*GPU Param.*	*Value*	*SMP Param.*	*Value*
CPU Clock.	2 GHz	Core Freq.	1.4 GHz	Registers	256 KB
L1 I-Cache (private)	32 KB	Memory Frequency	1.848 GHz	Shared Memory	96 KB
L1 D-Cache (private)	64 KB	# of SMPs	16	Threads	3072
L2 Unified Cache	2 MB	# of MC.	16	TB Limit	16
DRAM	512 MB	Sched. Policy	GTO	Warp Scheduler	4

classify the possibilities into $|S|$ groups at the beginning of the scheduling and calculate an average $O_p(X_i)$ based on a short profile, to 100 possibilities per group.

In algorithm 2, we select the first n groups based on the sorted profile results in descending order and perform the following scheduling on them (lines 1-2). We traverse Kr_p, pop the candidate kernel k_j from Kr_p (lines 3-4), and assume that k_j will be dispatched to the SMPs according to the i-th possibility (line 5). We join Kd_p and Kr_p into K_p (line 6). We traverse K_p and calculate $O_{p+1}(X_{i'})$ for each kernel in the set, as described in algorithm 1 (line 7-8). Similar to algorithm 1, if the candidate kernel k_j is ready to occupy the PCI-e bus in the $(p + 2)$-th scheduling period, then we will calculate $O^{pot}_{p+2}(X_i)$ for k_j based on the previous assumption of launching k_j in the $(p + 1)$-th scheduling period (lines 9-12). We enter the results on a list and return this list to the scheduling described in algorithm 1 (lines 13-18). In C-QoS scheduling, SMPs are shared among the concurrent GPGPU applications, and our proposed algorithm is compatible with the techniques of more fine-grained resource sharing for multitasking GPUs [15], [16].

As described in line 20 in algorithm 1, we obtain *dlist* and sort this list in descending order. We select one scheduling decision according to the definition of ϵ-greedy algorithm. We set $\epsilon = 0.1$, which indicates a 90% chance of selecting the decision with the largest O_p and a 10% chance of making a random selection among all possible scheduling decisions. This finding alleviates the problem that the greedy algorithm falls into a local optimum too early to some extent.

VII. EVALUATION

A. EXPERIMENTAL SETUP

To evaluate our design, we use gem5-GPU [23], which is a heterogeneous CPU-GPU simulator that consists of the latest version of GPGPU-Sim [46] and gem5 [50], and modify the GPGPU-Sim part to support spatial partitioning, preemptive multiprogramming, and the same assumptions and implementation described in previous work [14], [33], [34]. Applications for evaluation are selected from the chosen benchmark sets described in Table 1. We ran 100 M cycles for each collocation of these benchmark workloads. The QoS goal, which is defined as the end-to-end latency in this study, substantially varies among applications, and we need to select

the least common multiple of them, which is 100 M cycles. The results are accurate when the simulation is longer than 1 M cycles [31]. If one application of the collocation ends before 100 M cycles, then it will be executed multiple times to ensure that a long blank does not exist in the application's lifetime. The simulation parameters are listed in Table 2, and these parameters are similar to the previous work [15]. The theoretical peak bandwidth of the PCI-e bus is set to 16 GB/s, which is the same as the configuration of the 16x PCI-e 3.0 bus. We do not evaluate all combinations to conserve time. All the experiments of this study were completed in approximately one week, which means that approximately 18 weeks are needed to traverse all the combinations with the same arrival time configuration. We believe the selected combinations can cover all the cases needed for the evaluation. All the experiments performed in Section VII are designed for the co-location of 2 GPGPU applications. We believe that C-QoS is scalable for more than 2 applications and the co-location of different types of applications (graphics, AI, etc.), which remain topics of our future work.

We reproduce the models described in [51] and extend them to explore preemptive multiprogramming. We employ these extended models to predict the applications' completion time at runtime. In addition, we reproduce the algorithms adopted in two prior QoS support schemes to multitask the GPU on both the OS level (Baymax [7]) and the device level (Spart-QoS [14]). We consider that these two works are the most representative studies of the corresponding type of scheduling method. Some works augment the QoS prediction in Baymax with a more advanced technique [45], and they tune the kernel performance in Spart-QoS with a more fine-grained SMP partition [15]. We believe that these techniques enable performance improvement in any of the single scheduling methods. However, these improvements are not enough for solving the problem of insufficiency due to their noncooperation. Thus, we believe that comparing C-QoS with Baymax and Spart-QoS is reasonable, and it is a more efficient way to show the difference between cooperative scheduling methods and uncooperative scheduling methods. This comparison will not produce a critical error without considering the previously mentioned augmented methods.

B. END-TO-END LATENCY PREDICTION

We perform validation of the application's end-to-end latency prediction with a focus on the kernel execution tasks time and memcpy tasks time. For the selected applications, we determine that the results of the prediction match the application's corresponding performance, which is measured in gem5-GPU.

Figure 6 presents percentage composition comparisons for Rodinia and Parboil benchmarks, which are run in gem5-GPU and the predictor in this study. In all the 10 GPGPU applications, both the duration of the kernel execution in GPU and the data transfer on the PCIe bus is highly correlated to gem5-GPU. The simulation error is 11% in the worst case, and the average case is 3.27%.

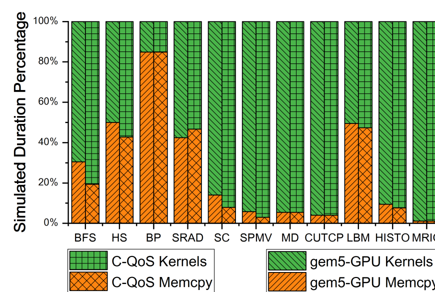


FIGURE 6. Prediction accuracy of C-QoS compared to gem5-GPU on 10 GPGPU applications. The average simulation error is 3.27%.

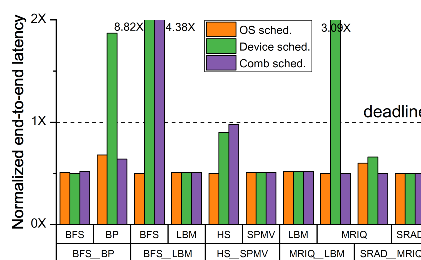
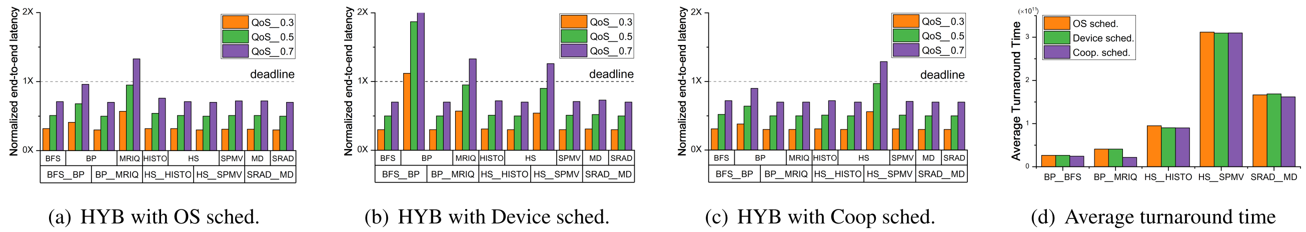


FIGURE 7. QoS-violation with a simple combination of the two scheduling methods with *QoS*_0.5.

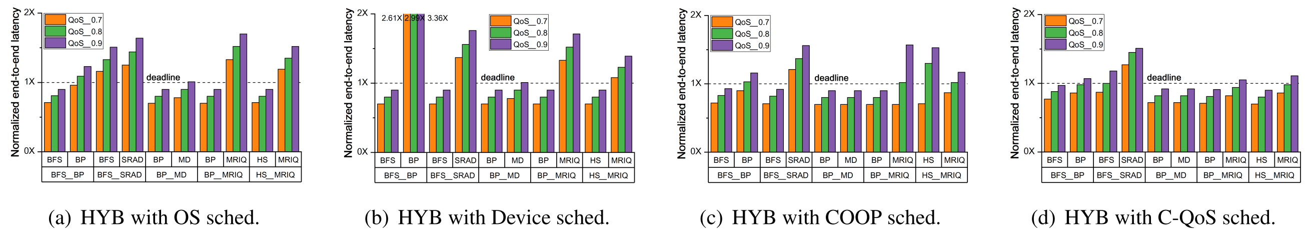
C. QoS VIOLATION WITH SIMPLE COMBINATION AND SIMPLE COOPERATION OF OS-LEVEL SCHED AND DEVICE-LEVEL SCHED

As shown in Figure 7, the QoS guarantee of the simple combination of the two scheduling methods (0.9) is degraded compared with the OS-level scheduling method (1.0). The decision made by two scheduling methods may conflict with each other, which may cause the avoidable QoS violation to occur. According to their description, the OS-level scheduler determines whether a ready kernel is to be launched or delayed depending on its QoS headroom. This concept was defined in prior work. The OS scheduler will decide to launch a kernel if it ensures that this kernel's predicted duration is within its deadline, and this kernel will not cause any QoS violation of the other running kernels. Conversely, the device-level scheduler tends to maximize the resource utilization of the GPU by adopting a dominant-resource-fairness (DRF) metric [34]. When they are combined, the following occurrences are possible: the OS-level scheduler feeds the accelerator with kernels from different applications based on its scheduling criteria and believes that they will share the PCI-e bandwidth and the accelerator's SMPs and satisfy their QoS requirements. Moreover, the device-level scheduler keeps re-allocating the hardware resources to minimize the difference in the amount of the dominant resource in each scheduling period, which may starve the application whose kernels have higher dominant resource occupancy most of the time. This situation may eventually cause the QoS violation of this application. We believe that this combination of the two methods may cause the performance degradation of the concurrent GPU applications' performance. This kind of degradation



(a) HYB with OS sched. (b) HYB with Device sched. (c) HYB with Coop sched. (d) Average turnaround time

FIGURE 8. **QoS-violation with the simple cooperative scheduling method.**



(a) HYB with OS sched. (b) HYB with Device sched. (c) HYB with COOP sched. (d) HYB with C-QoS sched.

FIGURE 9. **Average QoS violation over 100 sequences with *QoS*_0.7, *QoS*_0.8, *QoS*_0.9.**

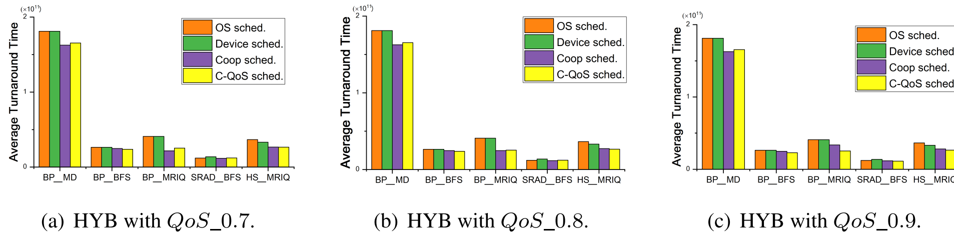
may continue to deteriorate, which may eventually offset the gain from the collocation of the two scheduling methods when the QoS constraint becomes strict.

According to the description in Section II, we suggest that one GPU kernel is composed of three types of tasks: the memcpy tasks in the *host to device* direction: *h2dmemcpy*, the memcpy tasks in the *device to host* direction: *d2hmemcpy*, and the kernel execution tasks of each TB: *ee*. As described in a prior work [51], we know that tasks in the same application are submitted and executed in an FIFO order. Moreover, tasks in different applications can run concurrently on the accelerator if a sufficient amount of computational resources exist because no dependency exists among these tasks. According to these descriptions, we suggest that kernels possess four states in their life span: not ready, ready, being processed, and finished. For example, for a newly arriving GPGPU application, the first *h2dmemcpy* task's state in its stream is set to 'ready' and the following tasks's states are set to 'not ready'. If this task is selected and processed by the copy engine, this task's state will be changed to 'being-processed' and the following tasks' will remain as 'not ready'. If this task is finished at the copy engine, then its state will be set to 'finished', the next task in the stream of the same application will be changed to 'ready', and the following tasks' will remain as 'not ready'. This process also applies to *ee* tasks and *d2hmemcpy* tasks. According to the previously mentioned definition, we propose a simple cooperative method: in each scheduling period, we collect the number of ready tasks whose states are 'ready' or 'being processed' and the number of resources that they occupied in each engine according to their specific category. We calculate a weighted sum: $trans_cnt$ and $comp_cnt$. If $trans_cnt \geq comp_cnt$, then we determine that operating OS-level scheduling is more beneficial than device-level scheduling in the present scheduling period. Otherwise, we select device-level scheduling for the operation. Compared with the use of single scheduling methods (0.9), Figure 8 shows that use of a simple cooperative scheduling method (COOP sched or COOP in the following section) achieves the largest number of QoS-guaranteed applications. The advantage of applying the device-level scheduling method in system utilization improvement is also sustained (4.72% speedup on average in the concurrent pair's turnaround time).

D. QoS VIOLATION COMPARISON WITH DIFFERENT SCHEDULING METHODS IN DIFFERENT CO-LOCATION SCENARIOS

According to the description in Section IV, we combine the selected benchmarks to evaluate 5 pairs according to their types; each pair consists of two different types of workloads (HYB). To share the GPU by the two GPGPU pairs, $5 \times 20 = 100$ sequences are generated. In each sequence, every GPGPU application possesses a different arrival time and both of them have QoS targets. We ran 100 M GPU cycles as described in section VII-A. We use the ratio of the number of QoS guaranteed applications to the total as our metric to compare the scheduling performance of each QoS support scheme: OS-level scheduling method, device-level scheduling method, simple cooperative scheduling method, and C-QoS scheduling method. As explained in Section IV, the QoS level of each GPGPU application is set as a percentage of the end-to-end latency in its isolated run, which ranges from 70% to 90%, with a 10% step size.

Figure 9 shows the QoS violation of co-running two GPGPU applications using different scheduling methods. To average the result of all QoS levels, co-running two applications with OS-level scheduling method has the lowest ratio of QoS guarantee (50%) due to the nonpreemptive design, while C-QoS scheduling method achieves the best result (76.67%). Cooperative scheduling methods (COOP sched



(a) HYB with $QoS_0.7$. (b) HYB with $QoS_0.8$. (c) HYB with $QoS_0.9$.

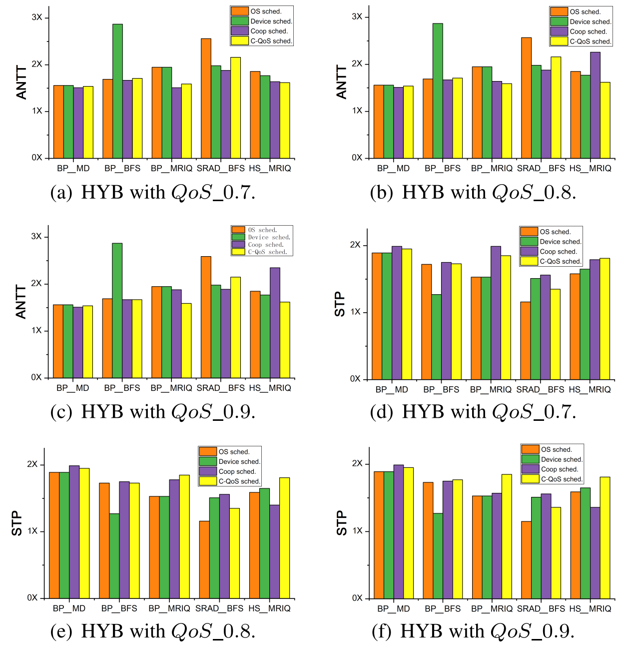
FIGURE 10. Average turnaround time over 100 sequences with ***QoS*_0.7, *QoS*_0.8, *QoS*_0.9.**

and C-QoS sched) are better than uncooperative methods (OS sched and Device sched) in almost all cases, with an average of 16.66% improvement in QoS guarantee. Cooperative methods overcome the major limitation of the uncooperative methods, which are explained in Section IV. Between the two cooperative methods, C-QoS sched achieves better results than COOP sched due to a more sophisticated manipulation of the OS sched and the Device sched, with an average of 13.34% improvement. C-QoS sched helps the concurrent GPGPU pairs reach their QoS goals more often than the uncooperative methods by 23.33%. However, when the QoS constraint becomes extremely strict ($QoS_0.9$), almost all the scheduling methods can only make one of the applications in concurrent pairs reach its QoS goal. In this case, the goal of guaranteeing the concurrent applications' goals exceeds the computational capability of the system in the evaluation.

E. SYSTEM UTILIZATION, ANTT, AND STP

As explained in Section IV, we measure the system utilization by the concurrent GPGPU applications' average turnaround time. We use two other metrics [49]—average normalized turnaround time (ANTT) and system throughput (STP)—to obtain a more comprehensive understanding of the performance of different scheduling methods. ANTT represents the user-perceived response time of the concurrent GPU applications; this metric is calculated as $ANTT = \frac{1}{N} \times \sum_{i=1}^{N} \frac{CPI_i^{MK}}{CPI_i^{SK}}$. N denotes the number of concurrent applications, CPI_i^{MK} denotes the number of cycles per instruction when application i is executed concurrently with different applications' kernels, and CPI_i^{SK} denotes the number of cycles per instruction when application i is executed in isolation. STP represents the system throughput; this metric is calculated as $STP = \sum_{i=1}^{N} \frac{CPI_i^{SK}}{CPI_i^{MK}}$. Note that ANTT is a lower-is-better metric, while STP is a higher-is-better metric.

As shown in Figure 10, cooperative scheduling methods (Coop sched and C-QoS sched) achieve a lower average turnaround time than uncooperative methods (OS sched and Device sched) in all cases by 17.27% on average, which indicates better system utilization and that a reduction in GPU time can be utilized to execute more throughput-oriented kernels. However, as shown in Figure 11, cooperative methods are not always better than uncooperative methods in the evaluation of ANTT and STP. Two reasons can explain these results. The first reason is the difference between the range of



(a) HYB with $QoS_0.7$. (b) HYB with $QoS_0.8$. (c) HYB with $QoS_0.9$. (d) HYB with $QoS_0.7$. (e) HYB with $QoS_0.8$. (f) HYB with $QoS_0.9$.

FIGURE 11. ANTT and STP over 100 sequences with ***QoS*_0.7, *QoS*_0.8, *QoS*_0.9.**

the completion time of the concurrent GPGPU applications. For instance, the completion time of HS in Figure 11, including the duration of kernel execution and data transfer, is 10 times longer than its concurrent benchmark MRIQ. When HS is co-located with MRIQ in the GPU, the C-QoS scheduler may decide to handle the large kernels from HS and postpone MRIQ's dispatching in some of the scheduling periods. These decisions, which are derived from the aim of guaranteeing the QoS goals of both of the applications, may cause a more significant difference between MRIQ's concurrent performance and its isolated performance than that of HS. In the calculation of STP, the value of MRIQ's $\frac{CPI_i^{SK}}{CPI_i^{MK}}$ may decrease and cause a lower final STP. This finding also applies to the evaluation of ANTT. Moreover, QoS support is considered the first aim to accomplish compared with improved system utilization. In some cases, we expect the scheduler to make wise choices to guarantee the concurrent applications' QoS goals, even if these choices may cause minor degradation of the applications' performance. This expectation is linked to the first reason: when the scheduler decides to sacrifice the

performance of one of the concurrent applications, whose completion time is shorter, the value of ANTT and STP may fluctuate and show poor results due to this scheduling decision. However, QoS supports of concurrent GPGPU applications are guaranteed in the end.

F. SCHEDULING OVERHEAD OF C-QoS

The main overhead of C-QoS is derived from the decision-making with the GPU hardware resource partition of the selected task. As described in Algorithm 2, O is positively related to the scheduling overhead of C-QoS, which is a tradeoff for the algorithm's time complexity and scheduling performance. The larger O indicates the higher probability of obtaining the optimal solution among all the scheduling options but a longer scheduling delay. Otherwise, the smaller O significantly accelerates the scheduling but may produce scheduling results that are even inferior to the single scheduling method. The calculation of the resource partition can be overlapped, which means that the scheduling overhead can be compressed. In the evaluation throughout this paper, we set O to a value less than 5. The average scheduling overhead of C-QoS is less than 8 milliseconds.

VIII. CONCLUSION

State-of-the-art QoS support for exclusively multitasking GPUs reside in the OS side or GPU side, and each of them cannot separately mitigate the QoS violations in all cases. In this paper, we prove that the single scheduling method or the simple combination of two scheduling methods is insufficient. We propose C-QoS, a cooperative scheduling scheme that consists of the OS-level scheduling method and the device-level scheduling method. C-QoS enforces the concurrent GPU applications' QoS goals while improving the overall system utilization. Moreover, we propose an algorithm to exploit C-QoS to improve the scheduling performance of QoS support. The evaluation results show that our design enables significant improvement in QoS support and overall system utilization versus uncooperative scheduling methods.

REFERENCES

[1] Amazon. *Amzaon High Performance Computing Cloud Using GPU*. Accessed: Jan. 1, 2020. [Online]. Available: http://aws.amazon.com/hpc/
[2] Oak Ridge National Laboratory. *Oak Ridge National Laboratory's 200 Petaflop Supercomputer*. Accessed: Jan. 1, 2020. [Online]. Available: https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/
[3] P. Carvalho, R. Cruz, L. M. A. Drummond, C. Bentes, E. Clua, E. Cataldo, and L. A. J. Marzulo, "Kernel concurrency opportunities based on GPU benchmarks characterization," *Cluster Comput.*, vol. 23, no. 1, pp. 177–188, Mar. 2020.
[4] S. Kato, K. Lakshmanan, R. Ishikawa, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2011, pp. 17–30.
[5] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged scheduling for fair, protected access to fast computational accelerators," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 301–316, 2014.
[6] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating system abstractions to manage GPUs as compute devices," in *Proc. 23rd ACM Symp. Operating Syst. Princ. (SOSP)*, 2011, pp. 233–248.
[7] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 2, pp. 681–696, Mar. 2016.
[8] W. Zhang, W. Cui, K. Fu, Q. Chen, D. E. Mawhirter, B. Wu, C. Li, and M. Guo, "Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters," in *Proc. ACM Int. Conf. Supercomput.*, 2019, pp. 58–68.
[9] H. Zhu, D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and M. Erez, "Kelp: QoS for accelerated machine learning systems," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 172–184.
[10] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A hierarchical data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous multi-accelerator NUMA nodes," *IEEE Access*, vol. 8, pp. 7861–7876, 2020.
[11] G. Florimbi, H. Fabelo, E. Torti, S. Ortega, M. Marrero-Martin, G. M. Callico, G. Danese, and F. Leporati, "Towards real-time computing of intraoperative hyperspectral imaging for brain cancer detection using multi-GPU platforms," *IEEE Access*, vol. 8, pp. 8485–8501, 2020.
[12] X. Geng, H. Zhang, Z. Zhao, and H. Ma, "Interference-aware parallelization for deep learning workload in GPU cluster," *Cluster Comput.*, Jan. 2020, doi: 10.1007/s10586-019-03037-6.
[13] R. A. Q. Cruz, C. Bentes, B. Breder, E. Vasconcellos, E. Clua, P. M. C. Carvalho, and L. M. A. Drummond, "Maximizing the GPU resource usage by reordering concurrent kernels submission," *Concurrency Comput., Pract. Exper.*, vol. 31, no. 18, Sep. 2019, Art. no. e4409.
[14] P. Aguilera, K. Morrow, and N. S. Kim, "QoS-aware dynamic resource allocation for spatial-multitasking GPUs," in *Proc. 19th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2014, pp. 726–731.
[15] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Quality of service support for fine-grained sharing on GPUs," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 269–281, Jun. 2017.
[16] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou, "Accelerate GPU concurrent kernel execution by mitigating memory pipeline stalls," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 208–220.
[17] J. Fang, Z. Chang, and D. Li, "Exploration on routing configuration of HNoC with intelligent on-chip resource management," *IEEE Access*, vol. 8, pp. 12117–12129, 2020.
[18] S. Najam, J. Ahmed, S. Masood, and C. M. Ahmed, "Run-time resource management controller for power efficiency of GP-GPU architecture," *IEEE Access*, vol. 7, pp. 25493–25505, 2019.
[19] Z. Lin, H. Dai, M. Mantor, and H. Zhou, "Coordinated CTA combination and bandwidth partitioning for GPU concurrent kernel execution," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 3, pp. 1–27, Jun. 2019.
[20] Z. Xu, X. Zhao, Z. Wang, and C. Yang, "Application-aware NoC management in GPUs multitasking," *J. Supercomput.*, vol. 75, no. 8, pp. 4710–4730, Aug. 2019.
[21] Z.-G. Tasoulas and I. Anagnostopoulos, "Performance and aging aware resource allocation for concurrent GPU applications under process variation," *IEEE Trans. Nanotechnol.*, vol. 18, pp. 717–727, 2019.
[22] J. J. K. Park, Y. Park, and S. Mahlke, "Dynamic resource management for efficient utilization of multitasking GPUs," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 527–540, 2017.
[23] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "Gem5-GPU: A heterogeneous CPU-GPU simulator," *IEEE Comput. Archit. Lett.*, vol. 14, no. 1, pp. 34–36, Jan. 2015.
[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.
[25] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssai, D. Geng, W. M. Liu, and W. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," IMPACT Tech. Rep. IMPACT-12-01, 2012, vol. 127, pp. 1–11.
[26] P. De Luca, A. Galletti, and L. Marcellino, "A Gaussian recursive-filter parallel implementation with overlapping," in *Proc. IEEE 15th Int. Conf. Signal Image Technol. Internet Based Syst. (SITIS)*, Nov. 2019, pp. 641–648.
[27] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2017, pp. 104–115.
[28] (2016).*NVIDIA Pascal GPU Architecture*. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
[29] (2017). *NVIDIA Volta GPU Architecture*. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[30] (2018). *NVIDIA Turing GPU Architecture*. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[31] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU spatial multitasking," in *Proc. IEEE Int. Symp. High-Perform. Comp Archit.*, Feb. 2012, pp. 1–12.

[32] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, vol. 11, Mar. 2011, p. 24.

[33] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 193–204.

[34] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 358–369.

[35] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 593–606, Mar. 2015.

[36] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "EffiSha: A software framework for enabling effficient preemptive scheduling of GPU," *ACM SIGPLAN Notices*, vol. 52, no. 8, pp. 3–16, Jan. 2017.

[37] X. Liu, B. Wu, X. Zhou, and C. Jiang, "FLEP: Enabling flexible and efficient preemption on GPUs," *ACM SIGOPS Operating Syst. Rev.*, vol. 51, no. 2, pp. 483–496, 2017.

[38] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *Proc. IEEE 34th Real-Time Syst. Symp.*, Dec. 2013, pp. 33–44.

[39] H. Lee and M. A. A. Faruque, "GPU-EvR: Run-time event based real-time scheduling framework on GPGPU platform," in *Proc. Design, Autom. Test Eur. Conf. Exhibit (DATE)*, Mar. 2014, p. 220.

[40] Y. Ukidave, X. Li, and D. Kaeli, "Mystic: Predictive scheduling for GPU based cloud servers using machine learning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 353–362.

[41] Y. Song, O. Alavoine, and B. Lin, "A self-aware resource management framework for heterogeneous multicore SoCs with diverse QoS targets," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 2, pp. 1–23, Apr. 2019.

[42] M. Knap and P. Czarnul, "Performance evaluation of unified memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and volta GPUs," *J. Supercomput.*, vol. 75, no. 11, pp. 7625–7645, Nov. 2019.

[43] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-slicer: Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 230–242.

[44] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "PipeDream: Fast and efficient pipeline parallel DNN training," 2018, *arXiv:1806.03377*. [Online]. Available: http://arxiv.org/abs/1806.03377

[45] W. Zhao, Q. Chen, H. Lin, J. Zhang, J. Leng, C. Li, W. Zheng, L. Li, and M. Guo, "Themis: Predicting and reining in application-level slowdown on spatial multitasking GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2019, pp. 653–663.

[46] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2009, pp. 163–174.

[47] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," *ACM SIGPLAN Notices*, vol. 48, no. 4, p. 407, Apr. 2013.

[48] (2007). *NVIDIA Compute Unified Device Architecture Programming Guide*. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[49] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, May/Jun. 2008.

[50] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1, Aug. 2011.

[51] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," *ACM SIGOPS Operating Syst. Rev.*, vol. 51, no. 2, pp. 17–32, 2017.

XINJIAN LONG received the B.E. degree from the Beijing University of Posts and Telecommunications, in 2015, where he is currently pursuing the Ph.D. degree with the State Key Laboratory of Networking and Switching Technology. His research interests include GPU computing, autonomic networking in 5G, and artificial intelligence in edge computing.

XIANGYANG GONG received the B.E. and M.E. degrees from Xi'an Jiaotong University, China, in 1992 and 1995, respectively, and the Ph.D. degree from the Beijing University of Posts and Telecommunications, in 2012. He is currently a Professor with the Beijing University of Posts and Telecommunications. His research interests include IP QoS, video communications, novel network architecture, artificial intelligence, and mobile Internet.

YAGUANG LIU received the B.E. degree in electronic information engineering from the Ren'ai College, Tianjin University, in 2018. He is currently pursuing the M.E. degree with the Beijing University of Posts and Telecommunications. He is also a Research Assistant with the Institute of Network Technology, Beijing University of Posts and Telecommunications. His research interests include networking and collaborative computing in 5G, and application of reinforcement learning in intelligent equipments.

XIRONG QUE received the B.E. and M.E. degrees from the Beijing University of Posts and Telecommunications, China, in 1993 and 1998, respectively. She is currently an Associate Professor with the Beijing University of Posts and Telecommunications. Her main research interests include innovation applications, next-generation network architecture, artificial intelligence, and mobile Internet.

WENDONG WANG (Member, IEEE) received the B.E. and M.E. degrees from the Beijing University of Posts and Telecommunications, China, in 1985 and 1991, respectively. He is currently a Full Professor with the Beijing University of Posts and Telecommunications. He has published over 100 articles in various journals and conference proceedings. His current research interests include next generation network architecture, innovation applications, artificial intelligence, and mobile internet. He is a member of the ACM.