RAISE: Efficient GPU Resource Management via Hybrid Scheduling

Yue Weng, Tianao Ge, Xi Zhang, Xianwei Zhang, Yutong Lu
School of Computer Science and Engineering
Sun Yat-sen University
Guangzhou, China
Email: {wengy8, getao3}@mail2.sysu.edu.cn, {zhangx299, zhangxw79, luyutong}@mail.sysu.edu.cn

***Abstract*— As the *de facto* high-throughput accelerators, graphics processing units (GPUs) are now used in a wide spectrum of fields, including artificial intelligence, high performance computing and finance. While with excessive computing and memory resources, GPUs are facing significant challenges to reach high utilization by a monolithic task. Multiple tasks are thus concurrently running to share the GPUs, but they may adversely affect each other, causing performance degradation. As a result, it is extremely critical to manage resources in a reasonable way to strike a balance between utilization and performance. Targeting the issue, this paper proposes an effective resource management design via hybrid task scheduling. Our design continuously tracks the GPU executions and collects the usage statistics, which are then used to direct the task selection and dispatch, including the type, starting time and kernel dimensions. A prototype is developed on off-the-shelf GPUs by moderately refactoring the CUDA source codes. Experimental results show that the design can achieve up to 1.96x performance improvement (1.51x on average), meanwhile effectively boosting resource utilization.**

***Index Terms*—Resource management, GPU, Scheduling**

I. Introduction

Graphics processing units (GPUs) are widely deployed in today's cloud computing centers and supercomputers to meet the increasing demands of modern applications [1], [2]. Representatively, in the November 2021 supercomputer Top500 list, there were 152 machines equipped with GPU or some form of accelerators [3]. With the continuous boost of accelerator computing capability, this proportion has maintained a strong growth momentum in the past 15 years [4]. Accordingly, CPU-GPU heterogeneous systems have received wide attention and become prevalent in cluster environments. By offering massive parallelism and high energy efficiency, GPUs are inherently suitable for various high computing demand scenarios like machine learning, finance, and high-performance computing (HPC). GPUs not only offer the main computing power in the clusters but also have superior power efficiency in terms of Flops-per-Watt compared with CPUs [5], [6]. Popular GPU solutions are NVIDIA Ampere 100 [7], AMD Instinct MI200 [8], Intel Iris Xe Graphics [9] and so on, all featuring hundreds of computing units that process thousands of data items in parallel.

Although providing great potential to reach incredible high performance with rich computation and memory resources, GPUs are frequently suffering from severe underutilization, which further damages the energy efficiency ratio of clusters [10], [11]. The issue arises from the fact that GPUs require a large number of threads to be present to deliver high throughput. However, for many applications, there might be a virtually insufficient amount of data parallelism to fully utilize the GPU horsepower [12]. Targeting the issue, contemporary GPUs contain task queues, e.g., streams [13] or command queues [14], to enable concurrent executions, which can then be exploited by users to batch jobs together. Multiple task queues can be allocated to manage a bunch of kernels, so that independent ones can be simultaneously executed whenever resources are available. In detail, kernels from the same queue are serially executed to guarantee inter-kernel dependencies, but those from different queues can be asynchronously executed by the GPUs, which typically pick up kernels within these queues in a round-robin fashion [15]. Further, at the application level, programmers are allowed to slightly specify the queue priorities [13], [16], which are then statically used by GPU drivers and schedulers. While promising to better utilize GPU components, the concurrency support is operating at a high level and thus agnostic to the underlying resource availability, which may experience severe interference and cause performance degradation [17], [18].

To address the potential issues, a resource-aware scheduler is needed to dispatch non-competitive jobs and adjust task sizes depending on the available resources. Technically, this can be achieved by adding extra fine-grained hardware counters inside GPUs to monitor the resource usages, and forwarding statistics to drivers, which are further required to be augmented to flexibly shuffle the task execution orders. However, this can be extremely challenging due to the hurdles of engineering efforts involved to add hardware support and commonly proprietary driver implementations. Furthermore, it is costly and impractical to replace existing cluster devices extensively to obtain new hardware features. As such, it is desirable to enable the enhanced scheduler on the off-the-shelf GPUs, with information only from the application source codes and existing profiling metrics. Ultimately optimizing multitasking concurrently on a GPU is critical to the clusters, considering that the resource utilization of a single node has a positive correlation to the overall utilization of the clusters.

In this paper, we propose RAISE, a practical task scheduler to provide efficient GPU resource management via hybrid

scheduling. The contributions of this paper are:

- We identify the GPU usage issues of single-task running and uncontrolled task co-execution, and then propose to strengthen the task scheduler to be aware of resource availability.
- Our proposed design allows GPUs to cluster resource-complementary tasks on the basis of tracked resource usages. Even further, the kernel dimensions can be adjusted according to the task types and dynamic resource situations.
- The design can be lightly implemented on off-the-shelf GPUs, with moderate changes on source code and slight effort on profiling, making it feasible to be adopted in practice. Experimental results demonstrate that our proposed design can greatly improve task performance as well as raise resource utilization.

The rest of the paper is organized as follows. Related work is discussed in Section II. Section III introduces GPU backgrounds. Sections IV and V elaborate on the motivation and design, respectively. Section VI presents the experimental methodology and result analysis. The paper is concluded in Section VII.

II. Related Work

As GPUs play an increasingly significant role in clusters, public cloud and supercomputers, GPU resource management is of critical importance and has attracted a lot of attention. In view of software management, Kato et al. designed a new runtime driver Gdev [19] to use GPUs as first-class computing resources for users. However, it still requires experienced developers to handle the resource management in each logical GPU. As to kernel execution, Jiao et al. [20] worked to find the optimal kernel function execution pair to achieve lower performance loss in exchange for resource utilization improvement. Kernel slicing is used to accomplish concurrency, which needs to manually split a large kernel into smaller kernels with fewer threads. Pai et al. [21] proposed elastic-kernel aware concurrency policies that obtained better concurrency. To match the mapping scheme, programmers need to change the loopings over the original kernel code and replace any variables corresponding to physical dimensions. These designs inevitably require burdensome code adjustments, which are especially challenging when porting to new platforms. Our method can be employed expediently with lightweight API replacements.

As the foremost resource units of GPU, streaming multiprocessors (SMs) contain abundant computing cores and memory resources such as register files and shared memory [7]. The partition of SMs has been studied extensively [22], [23] for the purpose of running multi-tasking on a single GPU. To reduce the search space, Zhao et al. [24] dynamically allocated SMs to the workload based on the characteristics of the task, which will bring profiling overhead for determining SMs partitioning strategy in each execution. Xu et al. [25] proposed Warped-Slicer to partition GPU resources among different applications with a short online profile and intra-SM slicing strategy. Sampling the program characteristics is essential to locate the sweet point of the concurrency, which needs longer execution time and more memory resources. For preemptive scheduling, SM draining and context switch are two common design schemes [26], [27]. As the design intention for GPU is chasing high throughput, context switch is supported inadequately and needs to be avoided as possible. While preemption is a guaranteed solution to meet Quality-of-Service [28]–[30], the context saving brought about by preemption is still a challenging problem. Spatial multitasking is an alternative way to share the resource that allows multiple tasks sharing a GPU at the same time to avoid context switching. Aguilera et al. [22] presented a runtime algorithm to predict and adjust the SM allocation while balancing the performance and fairness. Besides, simultaneous multitasking exploits the thread block and warp dispatch mechanism to realize resource sharing among different tasks [23]. Adriaens et al. [31] evaluated several heuristics for partitioning SM and demonstrated the superiority of spatial multitasking compared with cooperative multitasking. The implements of these tasks require additional hardware resources (such as registers and memory) or more fine-grained control of the hardware unit (such as SM and warp scheduler), so the simulator is indispensable to customize the hardware modification. Nevertheless, these modification are challenging to be incorporated or applied with practical GPUs. Differently, our work is solely based on off-the-shelf GPUs and involves moderate changes of source codes, that will be more applicable to the production environment.

III. Background

A. GPU Organization

Fig. 1 shows the high-level organization of a general GPU based on NVIDIA design[1]. The composition of a GPU includes SMs, cache, constant memory and global memory. As the basic unit of computation, each SM contains multiple compute cores such as INT32, FP32, FP64 and tensor to meet different computational accuracy requirements. In terms of the memory hierarchy, an SM contains a private L0 cache and L1 cache for instructions and data respectively. Shared memory enables cooperation between threads and reduces frequent data access from global memory.

With the help of parallel programming models such as CUDA [32], HIP [33], and OpenCL [34], computing modules can be defined as kernel functions to be offloaded onto GPUs. Kernels that consist of large amounts of threads are organized into a grid of thread blocks. Taking CUDA as an example, 32 threads in a block are further grouped into a warp, which is the basic unit for scheduling and execution on SMs by the warp scheduler and dispatch unit. Threads within a warp follow the SIMT (*single instruction, multiple threads*) execution model. An SM can hold tens of warps at most under the limitations of hardware resources.

[1]Other GPU vendors such as AMD adopt similar designs, but with partially different terminologies.

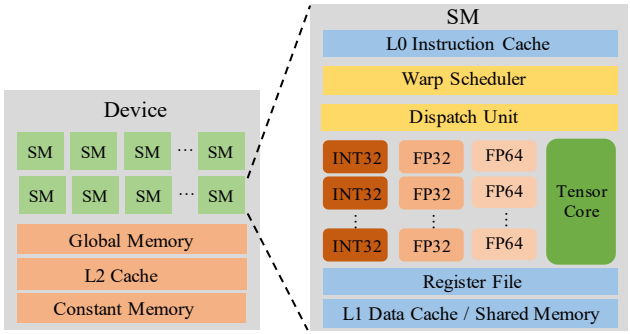

Fig. 1. General purpose graphics processing unit (GPGPU) high-level organization.

B. GPU Multi-tasking

In view of the fact that multitask concurrency may effectively boost utilization [23], GPU manufacturers have proposed a variety of solutions. As a common choice, stream [35] is a sequence of operations that execute in issue order on the GPU. Using multiple streams, kernel execution and data copy between CPU and GPU can thus be overlapped. Since Fermi architecture, Hyper-Q [36] is used to make multiple CPU processes or threads work on the same GPU instantaneously. Furthermore, by utilizing Hyper-Q capabilities, Multi-Process Service (MPS) [37], a binary-compatible implementation of CUDA, is developed to support co-operative multiple process applications, typically MPI jobs. In MPS, the client-server model is adopted and the client runtime built into the CUDA driver is transparently used by applications. Volta MPS clients submit work directly to the GPU without passing through the MPS server. In the latest Ampere architecture, Multi-Instance GPU (MIG) [38] feature allows a GPU to be physically partitioned into up to seven separate GPU Instances for CUDA applications.

While with effective supports of GPU sharing and partitioning, the aforementioned designs are still limited in raising resource utilization. For example, each fine-grained MIG instance still needs to take care of resource management. To this end, our proposed RAISE is seeking to improve resource usages, and it is orthogonal to the existing multi-tasking techniques and cluster scheduling methods.

IV. Motivation

With the continuous expansion of GPU hardware resources, efficient utilization has become an essential goal. Using GPU typically encounters two questions: 1) Can a single task make full use of GPU resources? 2) If not, what scheduling method should be adopted to enhance the usage? Around these two issues, we do the following studies, which inspire us to design a hybrid scheduling mechanism.

A. Resource Idleness of Exclusive Task Execution

To investigate the utilization on different GPU components, we run Breadth-First Search (BFS) from Rodinia benchmark [39] exclusively on the NVIDIA Tesla Volta 100 and collect some metrics via nvprof [40] . The abbreviations of the profiling metrics are described in Tab. II (see section V). Then we normalize the results to the theoretical peak values, as shown in Fig. 2.

We find that the exclusive execution of BFS consumes the GPU resources deficiently. The AOC ($achieved\_occupancy$) is only 6.1% of peak value, and the IPC (ipc) only reaches 2.2%. The SPU ($single\_precision\_fu\_utilization$) and DU ($dram\_utilization$) are both low, which are associated with the execution of computing instructions and memory load/store utilization respectively. Most of the metrics have an enormous gap to the peak values.

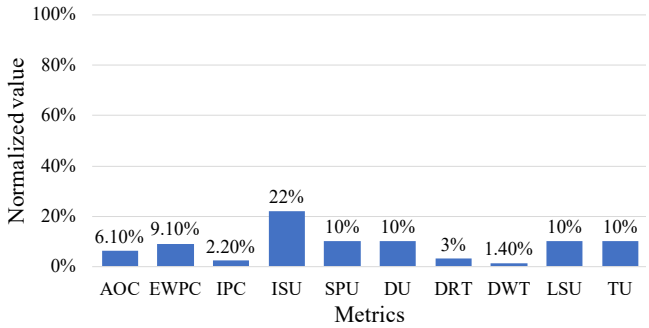

Fig. 2. Profiling results of Breadth-First Search (BFS). All the profiling results are normalized to the peak values. The greater value is, the higher utilization rate will be. These metrics including: 1) AOC: achieved_occupancy; 2) $EWPC$: eligible_warps_per_cycle; 3) IPC: ipc; 4) ISU: issue_slot_utilization; 5) SPU: single_precision_fu_utilization; 6) DU: dram_utilization; 7) DRT: dram_read_throughput; 8) DWT: dram_write_throughput; 9) LSU: ldst_fu_utilization; 10) TU: tex_utilization. Detailed metrics description can be seen in Tab. II.

TABLE I
BFS INSTRUCTION STALLS REASON BREAKDOWN.

Stall reason	Ratio	Stall reason	Ratio
memory_dependency	80.69%	inst_fetch	2.88%
exec_dependency	14.97%	memory_throttle	0.64%
not_selected	0.02%	pipe_busy	0.009%
constant_memory dependency	0.61%	other	0.181%

In Tab. I, we list the instruction stall reasons. It can be found that more than 80% ($memory\_dependency$) of the stalls are attributed to the data retrieval from memory. And, 14.97% ($exec\_dependency$) stalling is occurring because the input required by the instruction is not yet available. As BFS is a latency-sensitive task, most of the computing and memory resources are idle during execution, resulting in serious resource waste. This phenomenon is expected to become more prominent as GPU on-chip resources continue expanding.

B. Utilization Improvement via Hybrid Task Execution

Given that BFS alone cannot fully use the GPU, multi-tasking sharing GPU becomes the preferred option for improving resource utilization. For the illustration purpose, we select two tasks matrix multiplication (MM) and hotspot (HS) from [39], [41] respectively, which are both compute-bound. Then, we run BFS and MM concurrently using CUDA stream,

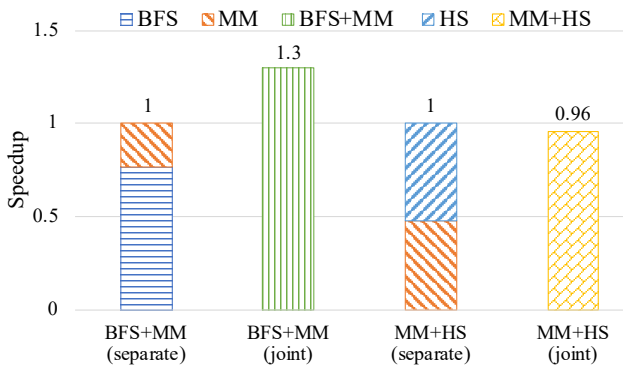

Fig. 3. Comparison of the different running combinations of BFS, MM and HS, where 'separate' means the applications run exclusively, and 'joint' means the applications run concurrently with the help of CUDA stream. The speedups are normalized to the 'separate' setting.

and recollect the performance values. Compared with the separation execution, we find that the joint execution brings a 1.3x speedup as shown in Fig. 3. This is because BFS is a latency-bound task ($IPC = 0.31/4.0(max)$, $DU = 1.0/10.0(max)$). The primary instruction stall reason is due to memory access latency ($memory\_dependency = 80.69\%$, as shown in Tab. I) and the computing units are idling for most of the time. Differently, MM is a compute-bound task ($IPC = 2.27/4.0$) and thus mainly occupies computing units. Therefore, when MM and BFS are executed concurrently, higher resource utilization can be achieved.

Further, we run MM and HS concurrently, but obtain worse performance compared to separation. As HS is also a compute-bound task ($IPC = 3.11/4.0$), the reason for this negative result can be blamed on the repetitive type of tasks and computational resource competition, which worsens the execution performance.

These experiments indicate that not all co-executions have satisfactory harvests. More attention should be paid to the co-executing strategy, as well as fine-grained control measures to balance resource utilization and efficiency

V. Design

A. Application Characterization

Application characteristics are of critical importance to direct resource usage optimization. We use profile tool to capture the execution statistics and select several representative metrics for analysis.

We take IPC and DRAM utilization (DU) as the primary metrics to distinguish computational and memory tasks. Tasks consume more computing resources (i.e., higher IPC and single-precision floating-point units SPU), while using limited memory resources will be classified as **compute-bound (COM)**. On the contrary, tasks restricted to memory access will be classified as **memory-bound (MEM)** based on DU. The **latency-bound (LAT)** tasks use less computing and memory resources and are limited by instruction fetching, memory access, or insufficient computing units, resulting in low utilization of overall resources. These understanding and division of task execution characteristics will serve the resource management and scheduling.

B. GPU Resource Management

Collection of GPU resource utilization. Since there are few APIs and complex closed-source GPU drivers, we could obtain limited information in user space [32], [33]. We seek to obtain GPU resource statistics with moderate efforts and overheads. As the profiling result represents the characteristics and resource usage during task execution, we can dynamically collect the GPU resource statistics by leveraging profiling information and tracking task status. Moreover, profiling results can be obtained in advance through profile tools [40], which are beneficial to make faster and more accurate decisions. As to the online scenarios, the profiling results can be collected by the transitory online profiling with limited overhead.

We set up a global resource vector $M_g^i\ (i \in [1, N])$ corresponding to the metrics in Tab. II. These metrics can depict the behaviors of computing and memory access reliably and are distinguishable among different tasks. Detailed descriptions of these metrics can be found in [40]. They can also be used to indirectly infer the GPU global resource capacity. When there is no task running, the values of these resources are maximized (i.e., 10, all values are scaled to 0-10). When a task t is selected and dispatched, it consumes a certain amount of resources vector $m_t^i\ (i \in [1, N])$, which are gained from the profiling results of the exclusive run. For each task t, we scale the profiled performance data m_t^i to a range of 0-10. The collection of different utilization helps us understand the current usage of various GPU resources and further provides valuable guidance for hybrid scheduling.

Dynamic kernel dimension. In order to fully use the fragmented resources and improve the overall resource utilization of the GPU, we dynamically adjust the dimensions of the kernel functions. For each kernel function, we set three different scales (i.e., Min, Mid, Max) of $gridsize$ and $blocksize$ in advance on the basis of original computational logic. The Max dimension setting considers both input size and the compute capability of the hardware [42]. The dimension settings of Mid and Min are demultiplicationed. For example, the Max $blocksize$ setting of MM is (32, 32), and the Mid and Min are (16, 16) and (8, 8), respectively.

By setting different $blocksize$, the resource demands and characteristics of the tasks are diverse. Smaller $blocksize$ could use fragmented resources more flexibly, while large $blocksize$ can take advantage of high parallelism when resources are sufficient. Before dispatching a task, we dynamically determine the kernel scale dimension based on the collected global resource usage information to adapt to changing resource situations.

Type scores. In addition to task type division, we also need to have a more comprehensive grasp of the overall characteristics of task execution. Inspired by the application clustering algorithm [43], we propose a concise but effective method to handle this problem – type scores. First of all, we define $\mathrm{Idx}(x)$ to represent the index on metric x. Computing

TABLE II
ABBREVIATIONS AND DESCRIPTIONS OF PRIMARY PROFILING METRICS.

Metric	Abbr.	Descriptions
achieved_occupancy	AOC	Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessors.
sm_efficiency	SME	The percentage of time at least one warp is active on a multiprocessor over all multiprocessors on the GPU.
ipc	IPC	Instructions executed per cycle.
issue_slot_utilization	ISU	Percentage of issue slots that issued at least one instruction, averaged across all cycles.
single_precision fu_utilization	SPU	The utilization level of the multiprocessor function units that execute single-precision floating-point instructions.
dram_utilization	DU	The utilization level of the device memory relative to the peak utilization.
ldst_fu_utilization	LSU	The utilization level of the multiprocessor function units that execute global, local and shared memory instructions.
tex_utilization	TU	The utilization level of the texture cache relative to the peak utilization.
dram_write(read) throughput	DW(R)T	Device memory write(read) throughput.
gld(gst)_throughput	GL(S)T	Global memory load(store) throughput.
eligible_warps_per_cycle	EWPC	Average number of warps that are eligible to issue per active cycle.
shared_utilization	SU	The utilization level of the shared memory relative to peak utilization.

score S_t^{com} is calculated by Eq. 1. Regarded as the most important computing metric, IPC's weight is set to 0.5, and the average weight of the remaining metrics is 0.5. Heavier weights reflect greater attention to the characteristics. The S_t^{com} can focus on both computing features and memory usage meanwhile. Similarly, the memory score is calculated by Eq. 2. For latency tasks, since there is no significant resource usage, we calculate the average of all metrics in Eq. 3[2]. Type scores can not only be used to distinguish tasks between different categories, but also grade the application characteristics within the same category, such as a sparse memory access task and an intensive memory access task. By grading the degree of application, we are able to have more precise control over the use of resources. A task with high type scores will be launched when the resources are sufficient and low type scores will be more appropriate when facing limited remaining resources.

$$S_t^{com} = 0.5 * m_t^{\mathrm{Idx}(IPC)} + 0.5 * \frac{1}{N-1}\sum_i^N m_t^i \quad (i \neq \mathrm{Idx}(IPC)) \tag{1}$$

$$S_t^{mem} = 0.5 * m_t^{\mathrm{Idx}(DU)} + 0.5 * \frac{1}{N-1}\sum_i^N m_t^i \quad (i \neq \mathrm{Idx}(DU)) \tag{2}$$

$$S_t^{lat} = \frac{1}{N}\sum_i^N m_t^i \tag{3}$$

We can also use the type scores to measure the characteristics of the global GPU resources (i.e., S_g^{com} and S_g^{mem}) when we replace m_t with M_g in Eq.1-3. Global scores will be used in scheduling to select the task and kernel dimension.

[2]The definition of type scores we use is straightforward and effective. There might be other efficacious methods to define the type scores and achieve similar results, these investigations belong to future work.

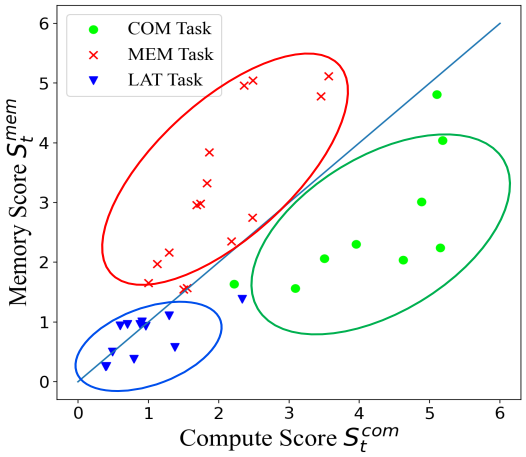

Fig. 4. Classification of tasks with different kernels and dimensions based on the compute and memory scores.

To examine the rationality of our type scores, we calculate both compute and memory scores of different kernels with varied dimensions (i.e., different $blocksize$ and $gridsize$) from the selected applications (see Section VI), and the results are reported in Fig. 4. It is clear that the type scores intuitively distinguish different tasks while preserving various categories of tasks. Our method is not strict to the effect of clustering algorithms, which only need to roughly classify the task types.

Hybrid scheduling. Based on the collection of GPU resource utilization and type scores, we implement GPU multitask hybrid scheduling with CUDA stream. 'Hybrid' means our scheduling mechanism emphasizes the type's complementarity and launched time of co-execution tasks, paying more attention to the balance of the global resource utilization. Fig. 5 is the complete scheduling flow chart.

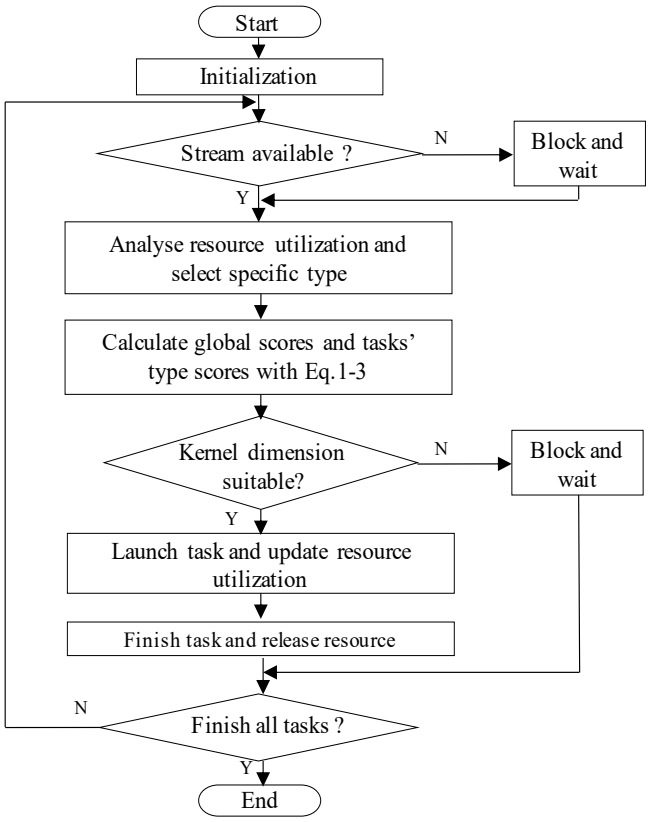

Fig. 5. The flow chart of hybrid scheduling design.

The scheduling process is as follows: 1) During initialization, we set the GPU resource vector M_g to the maximum value and create N_{str} free streams. 2) Each task will occupy a stream exclusively and tasks in different streams may be executed concurrently. If no stream is available, the main program will be blocked and wait until a task finishes and releases a stream. 3) Then we track and analyze the remaining GPU resources and attempt to select a task with a specific type. For computing resources, we set a threshold α for IPC, when $M_g^{\mathrm{Idx}(IPC)} > \alpha$, the GPU currently has enough resources to start new COM tasks. Similarly, we set β for DU. When IPC is insufficient, MEM tasks will be preferred. If DU is also deficient, LAT tasks will be the last attempt. 4) After the task type is determined, there are various qualified tasks and kernel dimensions for further selection. For example, GPU now could launch a COM task. We calculate global scores S_g^{com} with M_g according to Eq. 1-3 and compare it with S_t^{com} where task t belongs to a COM task. 5) To higher resource utilization, we select a task t with suitable kernel dimensions that can consume the most remaining resources, i.e., $\mathrm{Max}(\{S_t^{com} | S_t^{com} < S_g^{com}\})$. If there is no suitable task or kernel, which means the GPU is now fully occupied, the launch of new tasks will be blocked until resources are released. 6) If there is a kernel that meets the requirements, it will be executed and GPU resource usage will be updated simultaneously. 7) This process repeats until all tasks are completed.

Design implementation. CUDA stream and CPU thread are used to enable concurrent task execution on GPU. To utilize RAISE, we need to fine-tune the source code of workloads. The purpose is to implement that the scheduler can assign multiple streams to different tasks and execute them concurrently. Since all workloads implicitly use the default stream, we need two transformation steps. 1) Replace the memory management APIs (e.g., $malloc()$, $free()$, $cudaMemcpy()$) and kernel execution APIs (e.g., $kernel <<< gridsize, blocksize >>> ()$) to the APIs that support asynchronous execution of the specified stream (e.g., $cudaMalloc()$, $cudaFree()$, $cudaMemcpyAsync()$, $kernel <<< gridsize, blocksize, streamId >>> ()$, $cudaStreamSynchronize()$); 2) Convert the entry of the workload (e.g., the $main$ function) into an API that can be called externally, such as, $taskX.run(streamId)$. The workload now can be activated dynamically with a specific stream denoted as $streamId$.

VI. Evaluation

A. Experimental Setup

The hardware and software configurations in our experiments are exhibited in Tab. III. We selected 12 representative tasks from Rodinia [39] and CUDA SDK [41] to verify our method. The abbreviations, domain, input sizes and task types are specified in Tab. IV. For each task, we set up kernel dimensions with three different scales (Min, Mid, Max) in advance under lightweight code adjustment, and then conduct profiling. In practical scenario, the kernel settings and profiling results could be collected automatically based on historic job data. Kernel dimensions are described in Tab. IV. It should be noted that most of the tasks have only one kernel. As to the tasks with more multiple kernels, we only list the dimension of the dominant kernel of these tasks due to the space limitation. The kernel scales are classified based on the $blocksize$.

TABLE III
Hardware and software configurations.

	Settings
Hardware	CPU: Intel Xeon Silver 4208 @ 2.10GHz GPU: NVIDIA Tesla Volta 100
Software	OS: Ubuntu 18.04 x86_64 with kernel 4.15.0-123 GPU driver: 418.87 CUDA version: 10.1 GCC version: 7.5.0

The profiling results based on the Max dimensions are illustrated in Fig. 6. For the task with more than one kernel, we perform a weighted summation of the metrics results based on the execution time of kernels. For a task, we classify its type based on the IPC and DU metrics on Max dimension subjectively. We observe that the classification results according to Max are equally applicable to Mid and Min dimensions. Based on the profiling results and classification, we can find that the characteristics of different types of tasks are distinct and reasonable. The COM tasks (such as HS and VA) have

TABLE IV
WORKLOADS DESCRIPTION.

Workload	Abbr.	Domain	Type	$gridsize$ / $blocksize$		
				Min	Mid	Max
Hotspot [39]	HS	Physics Simulation	COM	(128, 128) / (8, 8)	(43, 43) / (16, 16)	(19, 19) / (32, 32)
Vector Add [41]	VA	Linear Algebra	COM	(32, 32) / (8, 8)	(128, 128) / (16, 16)	(512, 512) / (32, 32)
Matmul [41]	MM	Linear Algebra	COM	(320, 160) / (8, 8)	(160, 80) / (16, 16)	(80, 40) / (32, 32)
SRAD [39]	SRAD	Image Processing	MEM	(256, 256) / (4, 4)	(128, 128) / (8, 8)	(64, 64) / (16, 16)
Computational Fluid Dynamics [39]	CFD	Fluid Dynamics	MEM	(1520, 1) / (64, 1)	(760, 1) / (128, 1)	(380, 1) / (256, 1)
Gaussian Elimination [39]	GE	Linear Algebra	MEM	(128, 1) / (8, 1)	(64, 1) / (16, 1)	(32, 1) / (32, 1)
Hotspot3D [39]	HS3D	Physics Simulation	MEM	(64, 128) / (8, 4)	(32, 128) / (16, 4)	(16, 128) / (32, 4)
K-Means [39]	KM	Data Mining	MEM	(176, 176) / (16, 1)	(88, 88) / (64, 1)	(44, 44) / (256, 1)
Breadth-First Search [39]	BFS	Graph Algorithms	LAT	(65535, 1) / (32, 1)	(32768, 1) / (64, 1)	(16384, 1) / (128, 1)
CUFFT [41]	CU	Image Processing	LAT	(8, 1) / (64, 1)	(16, 1) / (128, 1)	(32, 1) / (156, 1)
Pathfinder [39]	PF	Grid Traversal	LAT	(417, 1) / (64, 1)	(114, 1) / (128, 1)	(47, 1) / (256, 1)
Back Propagation [39]	BP	Pattern Recognition	LAT	(1, 2560) / (8, 8)	(1, 1280) / (16, 16)	(1, 640) / (32, 32)

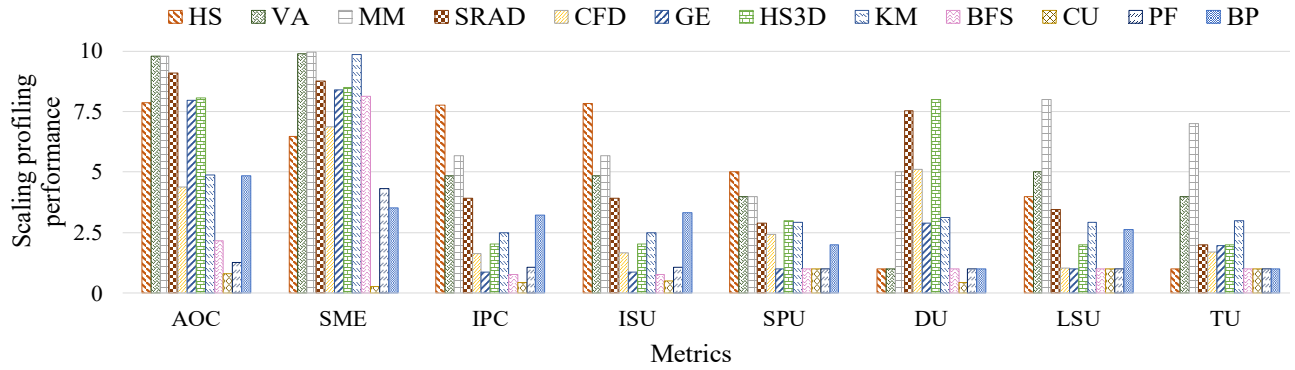

Fig. 6. Workloads profiling results on NVIDIA Tesla Volta 100. The resources described by the metrics include global (AOC, SME), computing (IPC, ISU, SPU) and memory access (DU, LSU, TU). The performance values have been scaled to 0 to 10 for intuitive display and further computation.

higher IPC, ISU and SPU which are the computing related metrics. The $SRAD$ and $HS3D$ are representative of MEM tasks equipped with prominent DU metric. Those tasks neither computing nor memory intensive are LAT tasks like the BFS and CU.

In our experiments, the maximum value of various resources is 10. The resource vector dimension N is 8, since the top eight metrics in Tab. II is sufficient to describe a task comprehensively. Given the averages of IPC and DU of all the tasks are about 2.5, we loosen the restriction on dispatch conditions and set the threshold α and β to 3, which could meet the resource requirements of most tasks. The experimental validations demonstrate that our settings are reasonable and effective. More detailed investigations of these settings are carried out in sensitivity studies.

Although co-scheduling of GPU kernels has been studied extensively [22], [23], [25], [26], [31], almost all studies are designed and evaluated on simulation. With the restriction of CUDA driver, it is scarcely possible to implement these works in the real environment and make fair comparisons. Without loss of generality, we choose three baselines that are close to the scenarios we use GPU for comparison. The first one is sequential execution (SQ), which completes all tasks of the same type and then launches other types sequentially. The second is round-robin (RR) with the order of COM, MEM and LAT circularly. The third is random (RD), where tasks are dispatched randomly. All the baselines support concurrent execution through streams. The kernel dimensions are also randomly selected for these baselines.

The evaluation criteria are the execution time to finish all tasks and the utilization efficiency of different GPU resources. We repeatedly run each experiment 10 times and then calculate the average and standard deviation. For the methods containing random numbers, different random seeds will be examined.

B. Performance and Utilization

To demonstrate the hybrid scheduling capability of RAISE under different load pressure, we design the total number of tasks to be processed from 12 to 120 times for simulating the task queue in the cluster environment. Each workload described in Tab. IV will be repeatedly executed 1 to 10 times,

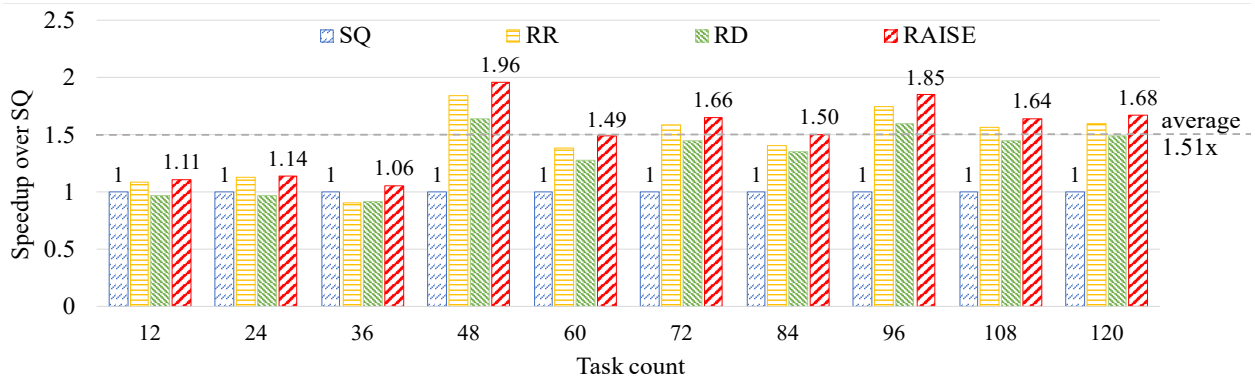

Fig. 7. Comparison with sequential baseline SQ, round-robin baseline RR and random baseline RD on the execution time of various numbers of tasks. The speedup is normalized to SQ.

respectively. We use $N_{str} = 4$ CUDA streams for concurrency. The speedups of the execution time are presented in Fig. 7.

The result reports that RAISE achieves great performance improvement compared to the non-controlled hybrid scheduling. The average speedup over SQ is 1.51x and up to 1.96x in the case of 48 tasks. Compared with RR and RD, RAISE also has boosted improvement. As the task count increases, RAISE delivers remarkable superiority compared to baselines. The average speedup over 72 tasks are 1.67x, which manifests the outstanding scalability. Besides, we find that RAISE has a much smaller standard deviation (average 0.29) compared to SQ (3.89), RR (1.07) and RD (4.12), indicating that our scheduling mechanism is more stable.

We have further analyzed the reasons for the performance improvement. The entire scheduling process will be profiled and compared with RD. As shown in Fig. 8, the performances of global[3] and computing metrics are given. Most metrics have significant improvements except occupancy AOC. The 16% drop of AOC indicates that the average active warps per active cycle are reduced and is beneficial to alleviate resource competition. RAISE prevents abundant warps competing for the same resources simultaneously and disorderly. Eligible warps $EWPC$ is logically increased as fewer warps be executed at the same time and more warps are eligible to be issued. The improvement in SM efficiency SME is slight. This is because that baselines and RAISE will use SMs whenever possible and most of the time at least one warp is active on an SM. Because of the loosen collecting rule of SME, the gap between baseline and RAISE is not obvious. In terms of computing resources utilization, the promotions are more remarkable. Issue slot utilization ISU has a 14% increase, which means RAISE can achieve more efficient instruction execution. With the help of hybrid scheduling mechanism and more elaborate consideration of resources usage, the issue of instructions IPC and single-precision floating-point computing units SPU are boosted by 13% and 22%, respectively.

Fig. 9 lists the metrics related to memory access. Compared with RD, DRAM utilization DU dropped by 25%, which is closely related to the decline of occupancy AOC. Fewer active warps per cycle result in a degradation of memory usage and access competition, yielding more memory resources such as texture memory TU and shared memory SU to be adequately utilized. Although the DU has decreased, the global memory throughput GLT and GST still have 13% and 53% improvement respectively. In the aggregate, the lightweight and real-time statistics collection leveraged by RAISE allows different resources to be employed proportionally. Less resource competition and higher utilization bring more efficient task execution.

[3] AOC, $EWPC$ and SME belong to the global metrics that represent the overall resource usage, such as average SMs efficiency and eligible warps per cycle.

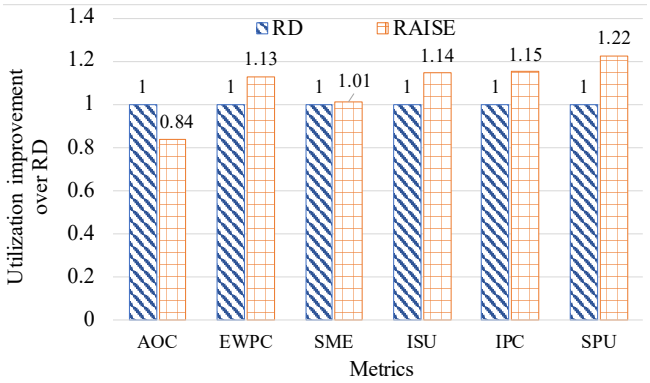

Fig. 8. Utilization improvement over RD on global and computing metrics.

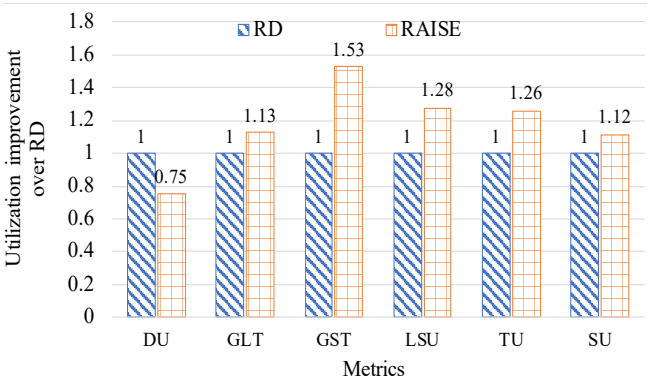

Fig. 9. Utilization improvement over RD on memory metrics.

C. Sensitivity Studies

We explore the impact of α and β as manifest in Fig. 10. By fixing one of the parameters and setting another one from

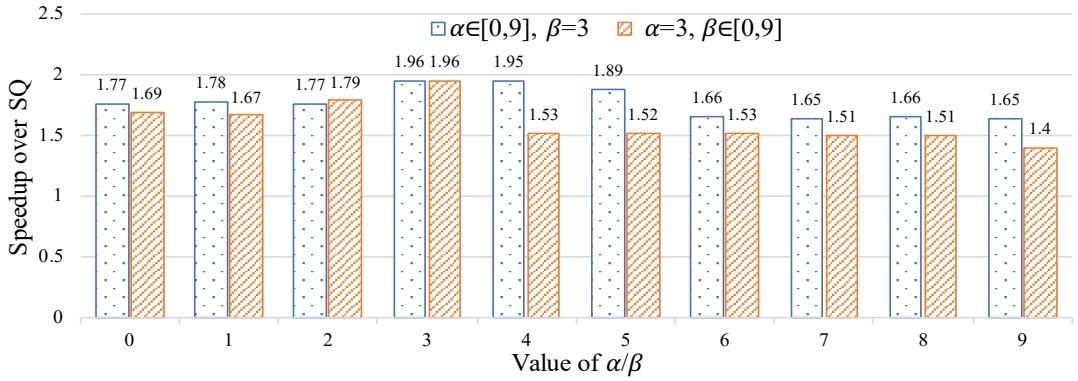

Fig. 10. Different settings of α and β. We change one parameter from 0 to 9 by fixing the other one.

0 to 9, we can compare the performance caused by different parameter settings. RAISE with smaller α will allow more compute-bound tasks to be launched. β for the memory-bound task works in the same way. We can observe that there is a sweet interval from 2 to 4, where the settings have a balance between hardware resource constraints and tasks resource consumption. In general, the influence of α and β on performance is limited which implies that RAISE is equipped with great robustness.

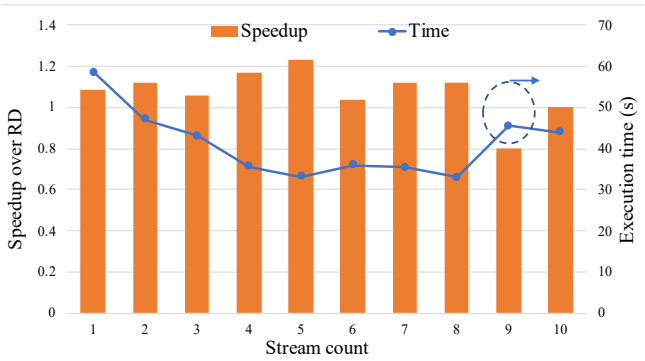

Fig. 11. The execution time and speedup of different stream count.

We further study the influence of stream count N_{str} on 36 tasks compared to RD in Fig. 11. As the increase of stream count, the absolute execution time of RAISE decreases first and then goes up. The speedup normalized to the random baseline yields optimal performance with moderate stream count.

When the stream count is restricted, a handful of tasks cannot drain the GPU, and account for restricted resource utilization. The scheduling results are close to serial or pair execution. On the contrary, as the number of streams increases, the speedups do not improve accordingly and even deliver a negative impact, which indicates that excess streams have little correlation to higher performance. Superfluous streams will bring heavy scheduling pressure, yielding inevitable resource competition and terribly time-consuming. A moderate number such as 4 or 5 streams can maximize GPU's capabilities by balancing resource competition and execution efficiency. As GPU resources expand, the optimal stream number needs slightly fine-tuning.

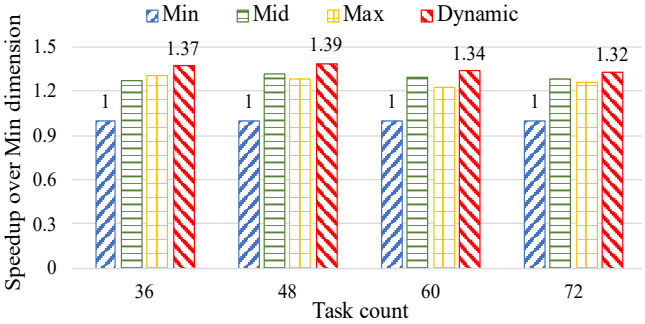

Fig. 12. The impact of dimension over RAISE.

We also study the impact of kernel dimensions as illustrated in Fig. 12. As the kernel scales are proportional to the resource occupancy, keeping Min or Max dimensions will result in insufficient or excessive resource usage, respectively. While Mid dimension has a relatively balanced performance. These analyses demonstrate the importance of kernel dimension adjustment under various resource conditions. Our RAISE with dynamic kernel dimensions can achieve up to 1.39x improvement compared to these prim variants. Dynamic-dimension decision has the ability to harmonize the kernel requirement with the remaining resources. By dynamically selecting the kernel dimension based on current resource conditions, resource competition, as well as execution efficiency can be coordinated commendably.

VII. Conclusion

To address the problem of underutilized resources, we propose RAISE, a lightweight GPU resource management via hybrid scheduling. The design is implemented as a resource-aware scheduler to direct the concurrent running of tasks based on resource availability. Experimental results show that RAISE can significantly improve GPU utilization in both computing and memory metrics. As higher resource utilization is beneficial to the execution time, RAISE can achieve an average 1.51x speedup and up to 1.96x compared to baselines. The sensitivity studies indicate that RAISE has great robustness. Moreover, RAISE can be completely implemented on the off-the-shelf GPUs and is thus more feasible to put into practice. In future work, we will seek to merge RAISE into an open-

source driver, such as AMD ROCm, aiming to pursue more precise and fine-grained control. Moreover, we will explore how to apply the hybrid scheduling design for larger scale tasks to further raise the concurrency level.

ACKNOWLEDGMENT

This research was supported by the National Natural Science Foundation of China-#62102465, #U1811461, #61872392, the Major Program of Guangdong Basic and Applied Research-#2019B030302002, and the Guangdong Natural Science Foundation-#2018B030312002.

REFERENCES

[1] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE micro*, vol. 31, no. 5, pp. 7–17, 2011.

[2] R. R. Expósito, G. L. Taboada, S. Ramos, J. Tourino, and R. Doallo, "General-purpose computation on gpus for high performance cloud computing," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1628–1642, 2013.

[3] Top500, "Top500 supercomputer list," https://www.top500.org/lists/top500/2021/11/ Accessed 11/2021.

[4] T. P. Morgan, "Top500 supercomputers: Hungry for the exascale feast," https://www.nextplatform.com/2021/11/15/top500-warm-leftovers-while-we-await-the-exascale-feast/ Accessed 11/2021.

[5] X. Mei, Q. Wang, and X. Chu, "A survey and measurement study of gpu dvfs on energy conservation," *Digital Communications and Networks*, vol. 3, no. 2, pp. 89–100, 2017.

[6] R. A. Bridges, N. Imam, and T. M. Mintz, "Understanding gpu power: A survey of profiling, modeling, and simulation methods," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, pp. 1–27, 2016.

[7] NVIDIA, "Nvidia ampere architecture," https://www.nvidia.com/en-us/data-center/ampere-architecture/ Accessed 11/2021.

[8] AMD, "Amd instinct mi200," https://www.amd.com/system/files/documents/amd-instinct-mi200-datasheet.pdf Accessed 11/2021.

[9] Intel, "Intel iris xe gpu," https://www.intel.com/content/www/us/en/products/discrete-gpus/iris-xe-aic.html Accessed 11/2021.

[10] J. Enos, C. Steffen, J. Fullop, M. Showerman, G. Shi, K. Esler, V. Kindratenko, J. E. Stone, and J. C. Phillips, "Quantifying the impact of gpus on performance and energy efficiency in hpc clusters," in *International Conference on Green Computing*. IEEE, 2010, pp. 317–324.

[11] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving gpu energy efficiency," *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, pp. 1–23, 2014.

[12] Q. Chen, H. Lee, H. Y. Yeom, and Y. Son, "Flexgpu: A flexible and efficient scheduler for gpu sharing systems," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 300–309.

[13] NVIDIA, "CUDA streams," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams Accessed 11/2021.

[14] S. Hodes, "Leveraging asynchronous queues for concurrent execution," https://gpuopen.com/learn/concurrent-execution-asynchronous-queues/ Accessed 11/2021.

[15] S. Puthoor, X. Tang, J. Gross, and B. M. Beckmann, "Oversubscribed command queues in gpus," in *Proceedings of the 11th Workshop on General Purpose GPUs*, 2018, pp. 50–60.

[16] AMD, "Amd polaris gpu architecture white paper," https://www.amd.com/system/files/documents/polaris-whitepaper.pdf Accessed 11/2021.

[17] X. Xu, N. Zhang, M. Cui, M. He, and R. Surana, "Characterization and prediction of performance interference on mediated passthrough gpus for interference-aware scheduler," in *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[18] X. Geng, H. Zhang, Z. Zhao, and H. Ma, "Interference-aware parallelization for deep learning workload in gpu cluster," *Cluster Computing*, vol. 23, no. 4, pp. 2689–2702, 2020.

[19] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class gpu resource management in the operating system," in *2012 USENIX Annual Technical Conference (ATC)*, 2012, pp. 401–412.

[20] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra, "Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 1–11.

[21] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 407–418, 2013.

[22] P. Aguilera, K. Morrow, and N. S. Kim, "Fair share: Allocation of gpu resources for both performance and fairness," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. IEEE, 2014, pp. 440–447.

[23] 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 *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 358–369.

[24] X. Zhao, Z. Wang, and L. Eeckhout, "Classification-driven search for effective sm partitioning in multitasking gpus," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, pp. 65–75.

[25] 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 *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 230–242.

[26] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 193–204, 2014.

[27] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared gpu," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 593–606, 2015.

[28] W. Han, D. Mawhirter, B. Wu, L. Ma, and C. Tian, "Flare: Flexibly sharing commodity gpus to enforce qos and improve utilization," in *The 32nd Workshop on Languages and Compilers for Parallel Computing*, 2019.

[29] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "Effisha: A software framework for enabling effficient preemptive scheduling of gpu," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 3–16.

[30] B. Wu, X. Liu, X. Zhou, and C. Jiang, "Flep: Enabling flexible and efficient preemption on gpus," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 483–496, 2017.

[31] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for gpgpu spatial multitasking," in *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 2012, pp. 1–12.

[32] NVIDIA, "Nvidia cuda toolkit," https://developer.nvidia.com/zh-cn/cuda-toolkit Accessed 11/2021.

[33] AMD, "Amd hip guide," https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html Accessed 11/2021.

[34] Khronos, "Opencl," https://www.khronos.org/opencl/ Accessed 11/2021.

[35] NVIDIA, "stream," https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html Accessed 11/2021.

[36] NVIDIA, "Hyper-q," https://developer.download.nvidia.cn/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf Accessed 11/2021.

[37] NVIDIA, "Mps," https://docs.nvidia.com/deploy/mps/index.html Accessed 11/2021.

[38] NVIDIA, "Mig," https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html Accessed 11/2021.

[39] 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 *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[40] , "nvprof," https://docs.nvidia.com/cuda/profiler-users-guide/index.html Accessed 11/2021.

[41] NVIDIA, "Nvidia cuda sdk," https://developer.nvidia.com/cuda-code-samples Accessed 11/2021.

[42] NVIDIA, "Compute capability," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications Accessed 11/2021.

[43] J. H. Ryoo, S. J. Quirem, M. Lebeane, R. Panda, S. Song, and L. K. John, "Gpgpu benchmark suites: How well do they sample the performance spectrum?" in *2015 44th International Conference on Parallel Processing*. IEEE, 2015, pp. 320–329.