Cluster Computing (2022) 25:597–617
https://doi.org/10.1007/s10586-021-03429-7

K-Scheduler: dynamic intra-SM multitasking management with execution profiles on GPUs

Sejin Kim[1] · Yoonhee Kim[1]

Received: 12 April 2021 / Revised: 23 July 2021 / Accepted: 24 September 2021 / Published online: 12 October 2021

Abstract
Data centers and cloud environments have recently started providing graphic processing unit (GPU)-based infrastructure services. Actual general purpose GPU (GPGPU) applications have low GPU utilization, unlike GPU-friendly applications. To improve the resource utilization of GPUs, there is the need for the concurrent execution of different applications while sharing resources in a streaming multiprocessor (SM). However, it is difficult to predict the execution performance of applications because resource contention can be caused by intra-SM multitasking. Furthermore, it is crucial to find the best resource partitioning and an execution set of applications that show the best performance among many applications. To address this, the current paper proposes K-Scheduler, a multitasking placement scheduler based on the intra-SM resource-use characteristics of applications. First, the resource-use and multitasking characteristics of applications are analyzed according to their classification and their individual execution characteristics. Rules for concurrent execution are derived according to each observation, and scheduling is performed according to the corresponding rules. The results verified that the total workload execution performance of K-Scheduler improved by 18% compared to previous studies, and individual execution performance improved by 32%.

Keywords GPU applications · Interference · Co-execution · Co-ScheML scheduler · Resource contention · GPU utilization

1 Introduction

General purpose graphics processing units (GPGPUs) perform fast parallel calculations using GPUs in various fields, including deep learning (DL) and high-performance computing (HPC). According to a recent report by TOP500 [1], which publishes rankings on the fastest supercomputers in the world twice a year, 141 of the 500 supercomputers available for ranking adopted GPU as their accelerator. There is also increased utilization of GPUs by researchers and private data centers (e.g., Google [2]); various large shared clouds (e.g., Amazon EC2 [3], Nimbix [4], Peer1 Hosting, and Microsoft Azure [5]) have begun to provide GPU-based infrastructure services to support GPU clouds.

More and more resources are being integrated with GPUs; however, the current system software cannot fully utilize the resources provided by single applications. For example, Summit, which currently ranks second on the TOP500 list [1], achieves an improvement of 65% of the best performance achieved in the GPU node when executing the GPU-friendly LINPACK [6] benchmark. By contrast, it only achieves 1.5% of the best performance for the High Performance Conjugate Gradients (HPCG) [7] benchmark, and shows a much lower GPU usage rate for actual applications [8]. Therefore, the concurrent execution of multiple applications has been proposed as a measure to improve resource utilization.

Many vendors have proposed GPU multitasking to improve the GPU utilization, such as NVIDIA's multiple process service (MPS) [9] and AMD's queue-based multi-programming [10]. However, multitasking using a simple

Sejin Kim and Yoonhee Kim have contributed equally to the work.

✉ Yoonhee Kim
yulan@sookmyung.ac.kr

Sejin Kim
wonder960702@gmail.com

[1] Department of Computer Science, Sookmyung Women's University, Seoul, South Korea

left-over strategy cannot minimize resource utilization, and shows a low performance compared to the single execution of individual applications. As the GPU cloud market size continues to increase, there are an increasing number of applications that need to be allocated to each host machine of a data center. Consequently, it has become more important to select a combination of applications with the best performance not only from the perspective of finding the best resource partitioning among the combinations of multiple applications, but also in terms of overall application execution management. In other words, the challenge is to find a combination of kernels that exhibits the best performance among multiple combinations of kernels. However, the larger the number of applications, the more complex it becomes to find the combination of kernels having the best performance.

The research to support multitasking is becoming more active. Spatial multitasking, which divides resources into subsets of streaming multiprocessors (SMs), was first studied to share GPU resources among multiple kernels [11–13]. However, they could not solve the under-utilization problem of intra-SM resources. A recent research subject is intra-SM sharing, which involves sharing intra-SM resources among multiple kernels [14–16]. Warped slicer [16] and SMK [15] presented the best resource partitioning methodology. However, because they both predicted the concurrent execution performance using only the single execution performances of all applications, the theoretical performance may differ from the real performance. Hongwen et al. [14] took note of this, and introduced the memory request and memory instruction limit to solve interference among kernels which can lead to sever performance degradation due to resource contention. To control memory request and memory instruction, the study required additional hardware and hardware changes; it cannot be implemented with existing hardware. In order to maximize intra-SM utilization, we need to predict multitasking performance based on resource usage characteristics and implement a scheduler supported on real hardware.

To solve this problem, the present study proposes K-Scheduler, a multitasking placement scheduler based on intra-SM resource-use characteristics for general-purpose applications. The basic principle of this scheduler is not only to satisfy the global goal of improving the performance of the total workload, but also to satisfy the expected performance of each single client. To this end, we need a scheduling technique that minimizes resource contention by reflecting a complex cache access pattern, execution pattern, and data dependence as well as the amount of static resources used that are determined at compile time.

This study uses the following approach. First, the resource-use characteristics of each application are analyzed according to the classification and individual execution characteristics of all the applications. It is generally known that the performance of compute-intensive applications improves as more resources are allocated, and the performance of memory-intensive applications is saturated before all resources are allocated [12, 13]. However, even among applications that belong to the same category, the execution performance may differ depending on the individual execution characteristics and the amount of resource allocation. Second, the concurrent execution characteristics are analyzed according to the classification of applications. Previous studies have proven the excellent performance of multitasking between applications having different characteristics. However, the present study observed that there may be a performance gain owing to multitasking even between applications having the same characteristics. Third, rules for concurrent execution are derived according to each observation, and scheduling is performed according to the corresponding rules.

The main contributions of this study are as follows.

- The performance change according to the amount of resource allocation of an application varies by the characteristics of the application. Hence, applications are classified according to their individual execution characteristics and the classification method. In addition, the performance change is analyzed according to the allocated SM and the number of thread blocks.
- The state-of-the-art intra-SM sharing shows that the theoretical performance and real performance may differ if only the single execution characteristics of an application is considered. To address this, the multitasking characteristics according to the classification of applications are observed.
- K-Scheduler, which is a scheduler that guarantees individual performance and improves the execution performance of all applications, is proposed. This avoids resource contention, which may occur when sharing resources according to the rules derived from the observation of multitasking characteristics.

The remainder of this paper is organized as follows. Section 2 discusses the background behind our study. Section 3 explains the motivation for the study, and Section 4 analyzes the characteristics of the intra-SM execution pattern. Then, Section 5 discusses the framework with K-Scheduler. Section 6 describes the experiments and presents an analysis of the results. Section 7 discusses related works, and Sect. 8 concludes the paper.

In the rest of the paper, abbreviations introduced in Table 1 will be used .

Table 1 Intra-SM resources and runtime stall for each benchmark application

Abbreviation	Meaning
LM	LavaMD [17]
BS	BlackScholes [18]
CUTCP	CUTCP [19]
STENCIL	STENCIL [19]
SPMV	SPMV [19]
LBM	LBM [19]
FT	FDTD3D [18]
QS	QuasiRandom Generator [18]
NW	Needleman-Wunsch [17]
HS	Hotspot3D [17]
DX	DXTC [18]
BO	Binomial Options [18]
CP	CP [19]
SG	SGEMM [19]
RD	Reduction [20]
COV	Covariance [21]
SY	Syr2k [21]
CONV	Convolution-3D [21]
SM	Streaming Multiprocessor
TB	Thread Block
ANTT	Average Normalized Turn-around Time
EPC	Eligible Warp Per Cycle
GFLOPS	Giga Floating point Operations Per Second
BW	Bandwidth

2 Background

The execution unit of GPU applications is the kernel function. The kernel produces a large number of threads, which are grouped into a thread block (TB). The NVIDIA GPU hardware has another thread group called a "warp." In recently developed GPUs, a warp is composed of 32 threads [22]. The number of warps (or TBs) that simultaneously operate in the GPU is limited by the GPU resources, such as the number of registers, the size of the sharing memory, and the maximum number of TBs.

The GPU consists of the SM, L2 cache, and GPU dynamic random access memory (DRAM). The SM shares the device memory through an interconnected network, and can switch from a warp to another warp without context switch overhead. Consequently, the warp scheduler can mask the delay of the warp by switching to a different warp when one warp is stopped owing to memory work or other reasons. These warps, which are managed simultaneously by one SM, are called "active warps." The number of active warps is determined by the resource use amount of the kernel function and hardware constraints. However, not all active warps can issue the next command. If an active warp cannot issue a command owing to a barrier or because it must wait for the result of the previous command, it is called a "stall warp." In contrast, if a command can be issued, the warp is called an "eligible warp." In other words, the number of active warps is the sum of the number of "stalled warps" and "eligible warps."

State-of-the-art GPUs can concurrently execute multiple GPU kernels in a single GPU hardware, thus enabling spatial multitasking for GPU hardware. NVIDIA's Hyper-Q technology [23] and AMD's queue-based multi-programming [10] are examples of such GPUs. NVIDIA's Hyper-Q technology cannot perform concurrent execution for kernels of other applications; consequently, NVIDIA enabled the concurrent execution of multiple applications by providing MPS [9]. However, MPS uses the left-over strategy, which allocates as many resources as possible to one kernel and the remaining resources to another kernel. Therefore, if the front kernel uses many resources and takes much time, the rear kernel is blocked.

3 Motivation

3.1 Heterogeneity of Kernel

Because the required resources of each kernel differ, intra-SM multitasking is required to improve utilization, and its performance is influenced by the combination of kernels. Table 2 shows the intra-SM resource usage of various applications and the reason for stall, which occurs at runtime in the environment of NVIDIA TITAN XP GPU and i7-5820K CPU. We can observe that the static intra-SM resource usage of each kernel is heterogeneous. For example, the QS application shows the largest usage of registers; it uses approximately 23% of registers in the entire SM, but it does not use any sharing memory. By contrast, the CUTCP uses approximately 10% of the total sharing memory, but less than 5% of the total registers. Therefore, QS has a limited maximum number of active TBs owing to the hardware restriction on registers. It can be seen from this that the intra-SM resources that are depleted by each kernel are heterogeneous, and the execution of a single kernel can decrease the utilization of intra-SM resources.

The runtime behavior of the kernel also has heterogeneity. In the case of BlackScholes application and SPMV application, most stalls are caused by memory dependency; however, for QS, the memory dependency stall is only 0.15%, and most stalls are caused by the delay of execution dependency and other reasons. Figure 1a shows the kernel time result graph when applications SPMV, QS, and BS in

Table 2 Intra-SM resources and runtime stall for each benchmark application

Application	Static resource		Runtime stall reason (%)		
	Registers/block	SMem/block (byte)	Exec dep	Memory dep	Others
LM	7168	7200	94.3%	0.15%	4.54%
BS	2944	0	6.36%	73.81%	3.88%
CUTCP	3328	4019	18.23%	0.91%	52.74%
STENCIL	4096	1000	10.45%	59.08%	7.88%
SPMV	5148	0	15.59%	61.31%	15.48%
LBM	4800	0	2.41%	14.9%	46.26%
FT	5120	1500	13.86%	26.25%	38.6%
QS	15360	0	22.43%	0.87%	43.26%
NW	656	2180	29.17%	37.16%	6.43%
HS	8192	0	11.86%	78.18%	13.29%
DX	4032	2048	46.11%	11.08%	3.90%
BO	4096	516	13.03%	0.00%	27.94%
CP	4224	0	35.96%	0.02%	33.42%
SG	5376	512	53.14%	13.38%	4.94%
RD	4608	2000	31.12%	35.69%	3.06%
COV	5632	0	2.10%	96.56%	0.68%
SY	6144	0	0.35%	71.45%	2.12%
CONV	5120	0	20.12%	16.55%	30.17%

Exec dep execution dependency, *Memory dep* memory dependency

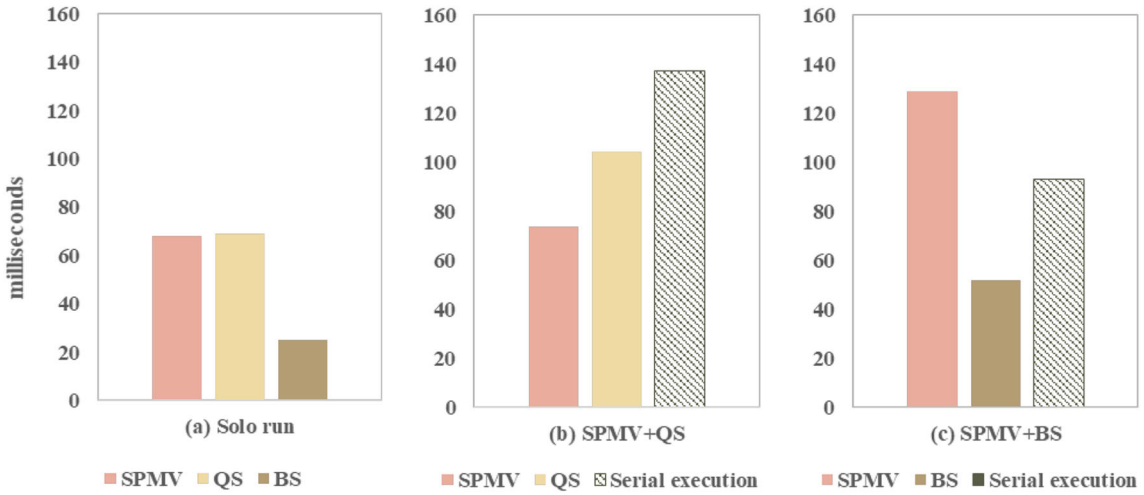

Fig. 1 Variations in the multitasking performance according to the concurrent execution kernel

Table 5 were executed alone. Figure 1b shows the multitasking result of SPMV and QS, and Fig. 1c shows the multitasking result of SPMV and BS. The rightmost bar shows the execution time when the applications are executed sequentially. It can be seen that even if the same SPMV application is executed, the concurrent execution performance varies depending on the application with which it was executed concurrently. When SPMV and QS were executed concurrently, the total execution time improved by approximately 30% compared to the sequential execution time. However, when SPMV and BS, which generate the same stall, are executed concurrently, the performance worsened as the concurrent execution time increased by 30% compared to the sequential execution time. Thus, the intra-SM resource utilization can be improved by concurrently executing applications using different resources or applications that stop for different reasons.

3.2 Limitation of existing multitasking placement technique

Warped-slicer [16] is an intra-SM sharing system, and it uses the water-filling algorithm to determine the TB partitioning between concurrent kernels. It determines the TB partition inside the SM based on the scalability curve that represents the performance of active TBs. Here, the TB partitioning that minimizes performance degradation is identified as the sweet point. Figure 2 explains that although excellent resource partitioning can be found, there is a limitation when considering the performance according to the resource allocation when an application is executed alone.

Figure 2a shows the scalability curves of STENCIL and SPMV. Substituting each application in the scalability curve of the Warped-slicer, we can identify the sweet spot that can maximize the performance of concurrent execution while satisfying the constraints of hardware resources. When the optimal number of combinations of active TBs, represented by the sweet spot, is found, STENCIL and SPMV have 12 and 3 active TBs, respectively. At this time, the performance normalized against the sequential execution time, the expected performance, is 1.93. However, as shown in Fig. 2b, the real performance is 0.94. Because the performance of concurrent execution was predicted based only on the result of the single execution of each kernel, the resource contention that can occur in the application that was executed concurrently was not reflected. As a result, the loss in the real performance was larger than that of the expected performance. Therefore, it is crucial not only to consider the resource distribution method between applications, but also to find the combination of applications that can maximize the real performance by predicting the contention of resources and the degree of contention that can occur in the concurrent execution of applications.

4 Analysis of characteristics of Kernel's intra-SM execution pattern

For the efficient multitasking of applications, a combination of kernels that can effectively divide the intra-SM resources and maximize the performance of concurrent execution must be identified. It is impossible to explore all combinations of kernels because the number of target applications and the number of resources in the GPU increase with time. Therefore, this study classifies kernels by employing the classification method presented in [24] using possible reasons for which a stall can occur during the execution of the kernel. We extract observations after analyzing the time at which the performance is saturated and the characteristics of concurrent executions. Table 3 shows the classification result according to the eligible warps per cycle (EPC) and kernel classification method [24] of the benchmark in Table 2. For example, LM and CUTCP are both "compute," and hence, they are compute-intensive kernels. They involve many integer and floating point computations. The EPC of LM is 0.17. When the number of eligible warps in each cycle is less than 1, it

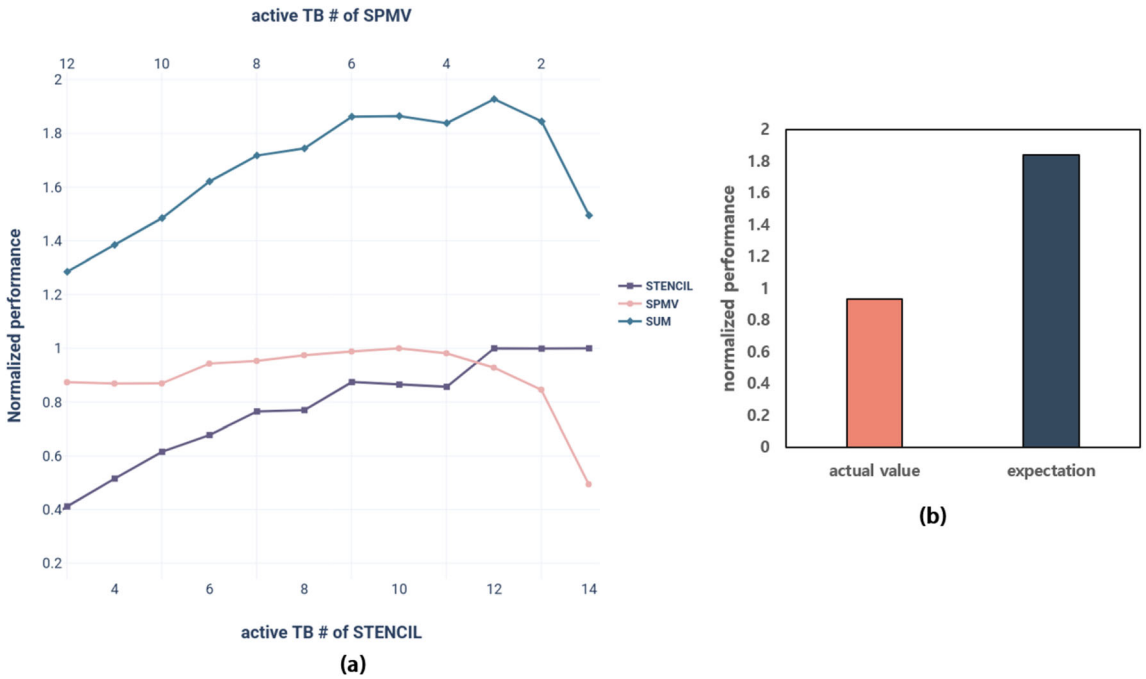

Fig. 2 **a** Scalability curve of Warped-slicer and **b** performance difference between actual and expected performance

Table 3 Intra-SM resources and runtime stall for each benchmark application

Application	Runtime stall reason (%)						Eligible warps/cycle	Type
	Execution dependency	Intruction fetch	Memory dependency	Others	Synch-ronization	Texture cache		
LM	94.30%	0.53%	0.15%	4.54%	0.46%	0%	0.17	Compute
BS	6.36%	1.89%	73.81%	3.88%	0%	0.20%	4.04	Memory
CUTCP	18.23%	13.83%	0.91%	52.74%	7.31%	0%	5.67	Compute
STENCIL	10.45%	1.99%	59.08%	7.88%	12.31%	0%	5.68	Memory
SPMV	15.59%	0.92%	61.31%	15.48%	0%	5.01%	0.72	Memory
LBM	2.41%	1.03%	14.90%	46.26%	0%	46.26%	0.59	L1 Cache
FT	13.86%	3.69%	26.25%	38.60%	17.30%	0%	0.55	Compute
QS	22.43%	2.12%	0.87%	43.26%	0%	0%	10.06	Compute
NW	29.17%	4.94%	37.16%	6.43%	20.82%	0.23%	0.31	Memory
HS	11.86%	1.30%	78.18%	13.29%	0.00%	0.01%	2.32	Memory
DX	46.11%	7.43%	11.08%	3.90%	20.27%	0%	3.75	Compute
BO	13.03%	9.25%	0.00%	27.94%	43.05%	0%	5.93	Compute
CP	35.96%	27.29%	0.02%	33.42%	0.00%	0%	3.41	Compute
SG	53.14%	3.46%	13.38%	4.94%	20.21%	0%	4.02	Compute
RD	31.12%	4.97%	35.69%	3.06%	12.89%	0%	0.53	Memory
COV	2.10%	0.57%	96.56%	0.68%	0%	0%	0.08	Memory
SY	0.35%	0.24%	71.45%	2.12%	0%	25.79%	0.12	Memory
CONV	20.12%	1.34%	16.55%	30.17%	0%	31.64%	0.86	L1 Cache

implies that many stalls occur. Meanwhile, the EPC of CUTCP is 5.67; thus, it is an application with relatively few stalls.

4.1 Characteristics of application execution according to static resource allocation

The LM and CUTCP described in the previous section are both computationally intensive kernels; however, they have different EPCs. The experiment result for this is shown in Fig. 3. This experiment shows that even the same computationally intensive kernels have different performance saturation points depending on their resource allocation. Figure 3a shows a graph of the kernel execution time by the active SM/TB of LM. It can be seen that the LM with 0.17 EPC does not have a performance gain owing to stalls even if many active TBs are allocated. As shown in this figure, the kernel execution time is 64 ms when the resource of 30SM-1TBs is allocated, and 58 ms when 30SM-2TBs are allocated. Thus, the performance improves when two TBs are activated. However, if 30SM-4TBs are allocated, the kernel execution time is 58 ms. This shows that the performance does not improve even if more resources are allocated. By contrast, as shown in Fig. 3b, the kernel execution time of CUTCP having 5.67 EPC with 30 active SMs is approximately 237 ms when one TB is allocated, approximately 123.974 ms when two TBs are allocated, and approximately 69 ms when four TBs are allocated. Thus, the performance improves almost linearly. When 10 TBs are allocated, the performance improves to approximately 46 ms. However, when 12 TBs are allocated, the performance decreases to 57 ms. This indicates that when 10 TBs are activated in CUTCP, the performance is saturated. Because the EPC is large, even if the number of active TBs increases, the performance is not affected by stalls; the performance improves when more resources are allocated.

Observation 1 The performance of a kernel with a low number of eligible warps per cycle (EPC) does not improve because of stalls, even if many resources are allocated.

4.2 Characteristics of multitasking of applications according to system performance requirement

The performance supply and requirement of each application for data transfer and computation varies according to the environment in which applications are executed. A GPU has a very fast memory transfer speed compared to the memory performance of a CPU. However, each

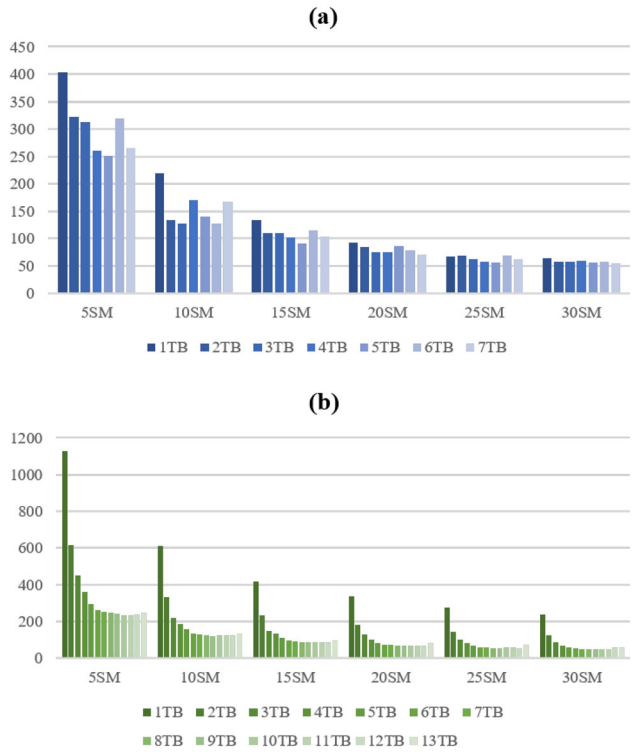

Fig. 3 Number of active SMs of **a** LM and **b** CUTCP. The kernel execution time changes according to the number of TBs launched per SM

hardware has a different available memory bandwidth, and if this is exceeded, the memory access time becomes a bottleneck for application execution. The float point operations per second (FLOPS) is an index used to represent the GPU performance. Each hardware has a different computation ability, and if this is exceeded, the computing performance becomes a bottleneck and increases the kernel execution time. The memory bandwidth of TITAN XP, which is the experimental environment of this study, is 547.6 GB/s; single-precision and double-precision are supported up to 12.15 and 379.7 Gflops, respectively.

One application usually does not maximize the use of the entire supplied performance. However, when two or three kernel instances are executed simultaneously, the available bandwidth and amount of computation are insufficient to satisfy the demand. Figure 4 shows the performance when the number of instances was increased for the BS kernel and LavaMD, which uses a 266-GB/s memory bandwidth per instance and requires 200.26 GFLOPS per instance, respectively. As shown in Fig. 4a, when two instances of BS are executed concurrently, the sum of the required bandwidth of each instance does not exceed the supply performance. Thus, there is no significant difference between this execution time and the execution time of one instance. However, if three instances are executed concurrently and the bandwidth demand exceeds the available bandwidth, the concurrent execution performance decreases rapidly. In Fig. 4b, which shows the variation in the number of instances of LavaMD, when floating-point operation below the supply performance was performed by concurrently executing two instances, the execution time is not much different from the performance time of one instance; the performance is almost double the performance of sequential execution. However, when three instances are executed concurrently and the supplied performance is exceeded, the floating point operation becomes a bottleneck, and the kernel execution time increases. This indicates that if the hardware cannot satisfy the bandwidth and computation requirements of applications, the resources are saturated and contention for resources occurs, resulting in lower performance of concurrent execution.

Observation 2 The cumulative DRAM bandwidth usage and computation amount (GFLOPS) of each application exceeds the supplied hardware performance, and the performance of concurrent execution is not improved.

4.3 Characteristics of concurrent execution according to classification of applications

The characteristics of concurrent execution according to the classification of applications are analyzed based on the pairing of applications in each group and the result of the concurrent execution of applications. In each graph in this section, the application in front of plus ("+") belongs to the first group, and the application after the plus belongs to the second group. This result shows the performance speed-up obtained by normalizing the performance time of two applications executed sequentially with respect to the concurrent execution time. In other words, if the speed up is larger than 1, there is a concurrent execution compared to sequential execution; otherwise, there is no performance gain, or the performance decreases.

4.3.1 Multitasking of computationally intensive application and memory-intensive application

The concurrent execution time of computationally intensive applications and memory-intensive applications was shortened by approximately 27% on average compared to the sequential execution time. Previous studies have found that multitasking between kernels that belong to different categories results in good performance [24]. The result of this experiment shows that the performance improves the most when multitasking is performed for applications with many stalls. Because these applications have a low EPC, their performance does not improve owing to stalls even if many resources are provided. However, it can be seen that

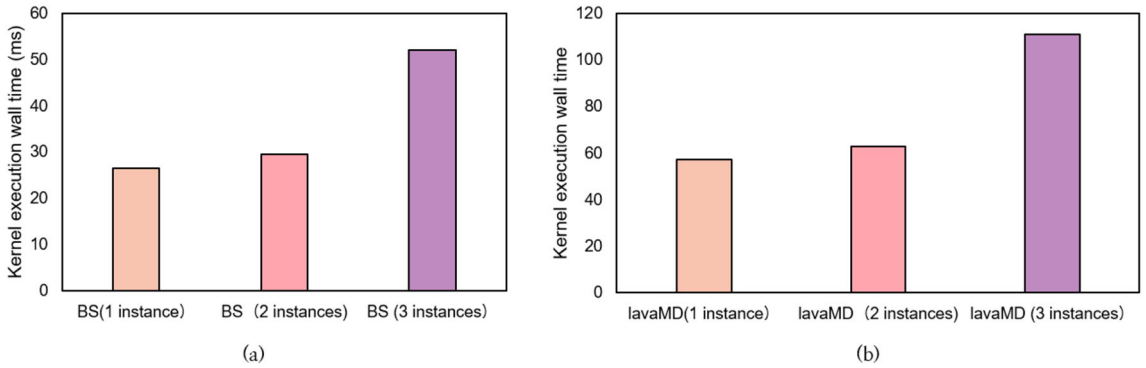

Fig. 4 Kernel performance time according to the number of instances of BS and LavaMD: **a** Kernel performance time according to the number of instances of BS. **b** Kernel performance time according to the number of instances of LM

when applications using difference resources are grouped together, their performance can be improved by each application complementing each other's stalls.

4.3.2 Multitasking between L1 cache-intensive application and applications of other categories

As shown in Fig. 5 [24], the concurrent execution of L1 cache-intensive applications with other applications shows a performance that is mainly similar to or worse than the performance of sequential execution. This is because the L1 cache is saturated. In the case of QS, the L1 cache transaction is close to 0, thus showing a performance improvement of approximately 11% by concurrent execution. This indicates that the performance gain of L1 cache-intensive applications can be expected only by concurrent execution with applications that use less L1 cache. This is also observed in a study using a simulator [16], which showed that the hardware also operated in the same way as the simulator.

Observation 3 The L1 cache-intensive application has no performance improvement of multitasking by concurrent execution with applications for which the number of L1 cache transactions exceeds the baseline.

4.3.3 Multitasking between memory-intensive applications

Table 4 shows the DRAM throughputs of memory-intensive kernel pairs. Figure 6 shows a graph for the multitasking result of each pair. The sum of the DRAM throughputs of the pairs in the table does not exceed the bandwidth of the system, but they all exhibit performances that are the same as or lower than the sequential execution performance. For example, the sum of DRAM throughputs of the NW+HS pair is 441 GB/s, showing the best multitasking performance, which improved by approximately 3% compared to sequential execution. The HS+RD pair that showed the worst performance is 376.951, which decreased by approximately 9% compared to the sequential performance. This result shows that the sum of DRAM throughputs of every pair does not exceed the bandwidth of the system, but there was no performance gain resulting from concurrent execution because the performance was similar to or lower than that of sequential execution.

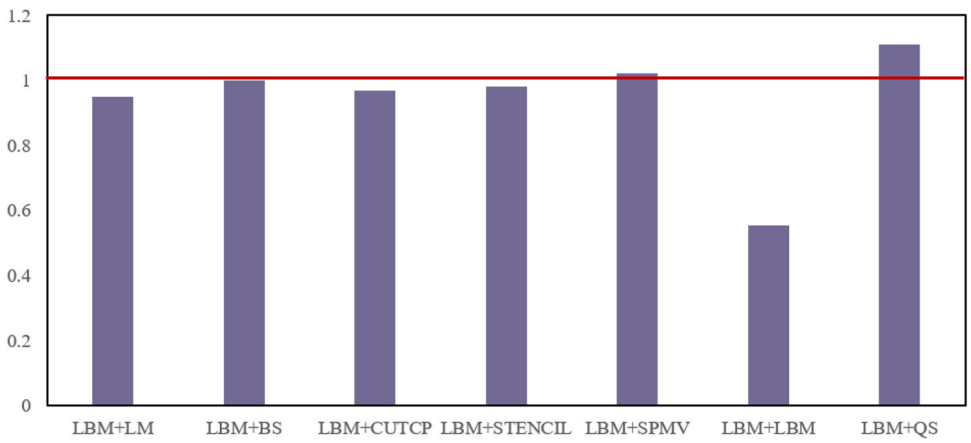

Fig. 5 Multitasking performance between L1 cache-intensive application and applications of other categories [24]

Table 4 Sum of DRAM throughputs for each multitasking pair

HS+RD	RD+NW	SY+STENCIL	SPMV+NW	NW+HS
376.951	96.991	390.922	429.715	441.12

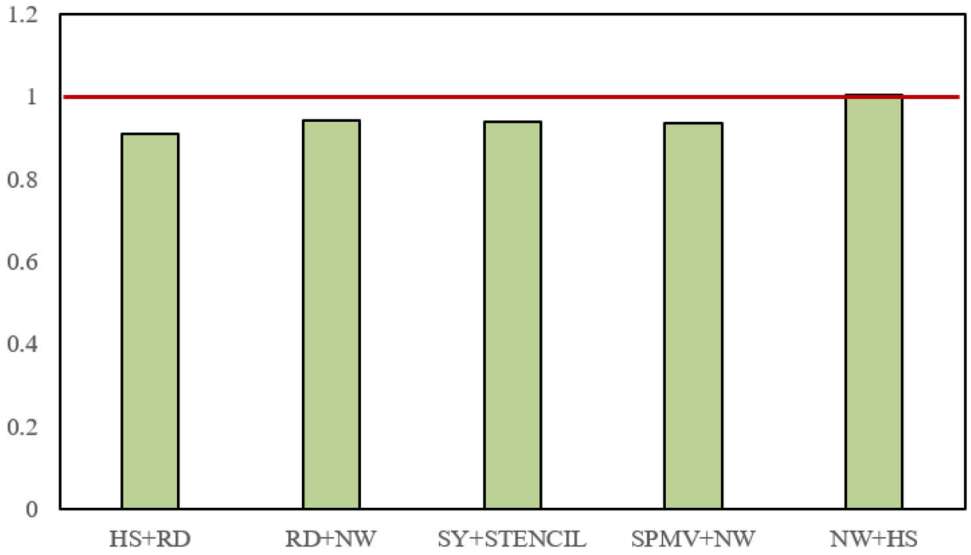

Fig. 6 Multitasking performance between memory-intensive applications

Furthermore, these results indicate that it is insufficient to consider only the sum of DRAM throughputs when determining the performance. It can be inferred that this is due to the DRAM bandwidth as well as the fact that there is no contention for resources among different memory systems, such as cache and sharing memories.

Observation 4 Multitasking between memory-intensive applications shows a lower performance than the sequential execution of each application.

4.3.4 Multitasking between computationally-intensive applications

Figure 7 shows the result of the concurrent execution of computationally intensive applications. The result of this experiment indicates that the concurrent execution of computationally intensive applications may result in performance gain. However, not all pairs show performance improvement compared to sequential execution. Figure 7a shows the multitasking performance between CUTCP, which is a computationally intensive application, and other computationally intensive applications. It can be seen that except for the LavaMD and finite-difference time domain (FDTD) pair, the applications exhibit a performance that is similar to or lower than that of the sequential execution. This is because CUTCP is an application with the highest EPC, and almost no stall occurs when CUTCP is executed alone. Hence, there are not many stalls to hide by concurrent execution. Therefore, the performance of CUTCP, which has an EPC smaller than 1, improved only by multitasking with LavaMD and FDTD. Figure 7b shows the multitasking performance of FDTD, whose EPC is less than 1, and other computationally intensive applications. It can be seen that the performance improved because the stalls occurring in FDTD can be hidden through concurrent execution with other applications. Therefore, it can be observed that if two applications both have large EPCs, there is no performance gain because there are not many stalls to hide through concurrent execution. However, it was observed that if any one application has a small EPC, there is a stall to be hidden, and performance improvement can be expected by performing concurrent execution.

Observation 5 If multiple computationally intensive applications whose EPC is larger than they are executed concurrently, there is no performance improvement through concurrent execution.

5 Framework architecture with K-Scheduler

This section introduces a resource sharing execution framework to solve low utilization problem of intra-SM resources. A kernel scheduler proposed in this study is named K-Scheduler. After describing the overall structure design with K-Scheduler, we define rules based on the

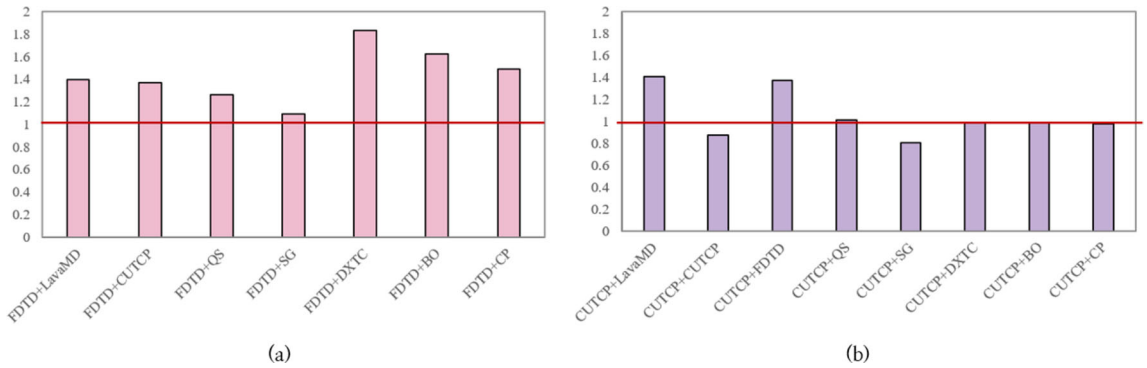

Fig. 7 **a** Multitasking performance of FDTD and computationally intensive application; **b** multitasking performance of CUTCP and computationally intensive applications

observation in Section 4.3 and K-Scheduler is introduced based on this.

5.1 K-Scheduler-based system structure

Application has dominant intra-SM resources; the concurrent execution of applications with different dominant resources can obtain a performance gain if applications with different dominant resources are placed together. The static profiling information, including the intra-SM resources used by the kernel, is obtained from Static Profiler using the NVIDIA CUDA Compiler (NVCC) [25] at compile time at the kernel level. The GPGPU application shares resources such as cache and global memory bandwidth, and must perform dynamic profiling as well as static profiling because stalls occur for different reasons. Thus, K-Scheduler, which classifies applications using the acquired profiling information, and which performs scheduling based on the classified applications, was developed. To this end, kernels are classified into predefined categories using static and dynamic profiling information. For example, when K1 kernel is submitted to the structure shown in Fig. 8, it is transformed to K_{C1}, including the kernel classification information through the kernel classifier. K-Scheduler performs rule-based scheduling using the kernel classification information and profiling information. As shown in Fig. 8, when the submitted kernel K_{C1} passes through K-Scheduler and submitted to the GPU, it is transformed to kernel K'_{C1} after receiving the amount of allocated resources and information regarding the SM placement to be performed. In this way, concurrent execution is carried out with minimized resource contention. SmCompactor [26] is used to maximize resource utilization while sharing intra-SM resources. It is a TB-based scheduling framework that enables intra-SM sharing in the hardware. SmCompactor enables the intra-SM multitasking of applications by allowing the submission of each TB to the desired SM. As shown in Fig. 8, each TB of the kernel that received the amount of resources and placement information is mapped to the task and waits in the hardware queue of the GPU through smCompactor. The hardware queue submits the TB to the corresponding SM according to the SM placement information. A sequence diagram to describe the interaction among components of K-Scheduler framework is shown in Fig. 9. When K-Scheduler requests profiling information, Kernel classifier acquires static and dynamic profiling information from Profiler. After Kernel classifier classifies kernels using profiling information, it returns kernel information which includes classification and profiling information to K-Scheduler. K-Scheduler decides placement of kernels according to scheduling algorithm using the information and let smCompactor runtime launch kernels. smCompactor runtime transforms kernels for intra-SM sharing.

5.2 Performance saturation point identification according to resource allocation

This section introduces a method of finding the resource allocation for each application when multiple applications are executed in SM. When the amount of allocated resources is increased for the GPGPU application, its performance may not improve or degrade further when a certain point is reached even if more resources are allocated [24]. Furthermore, the performance saturation point for the resource allocation varied according to the characteristics of the application. To maximize the benefit of concurrent execution in multitasking, the performance loss of each application must be limited to the permissible level, and as many applications as possible should be executed concurrently in one SM. To do this, it is important to identify the performance saturation point where the performance is not improved or improved only insignificantly even if the resource allocation is increased. In this study, if the performance is not improved at a certain rate when the

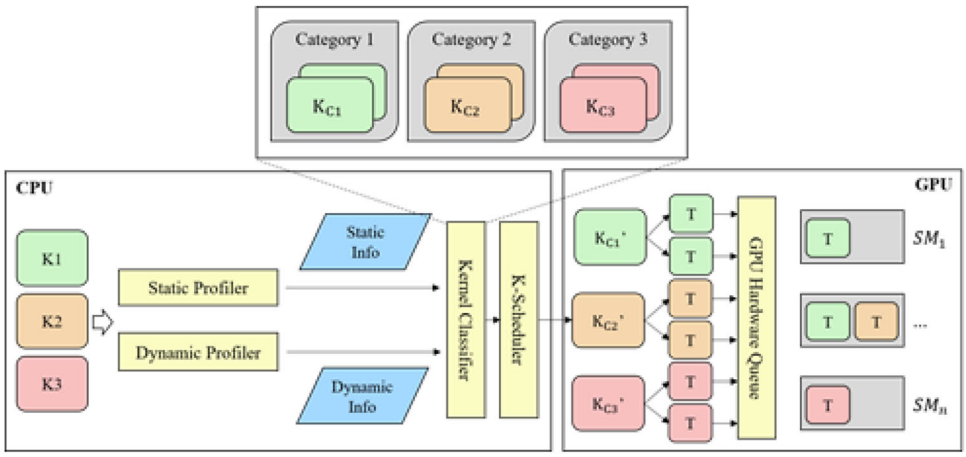

Fig. 8 System structure based on K-Scheduler

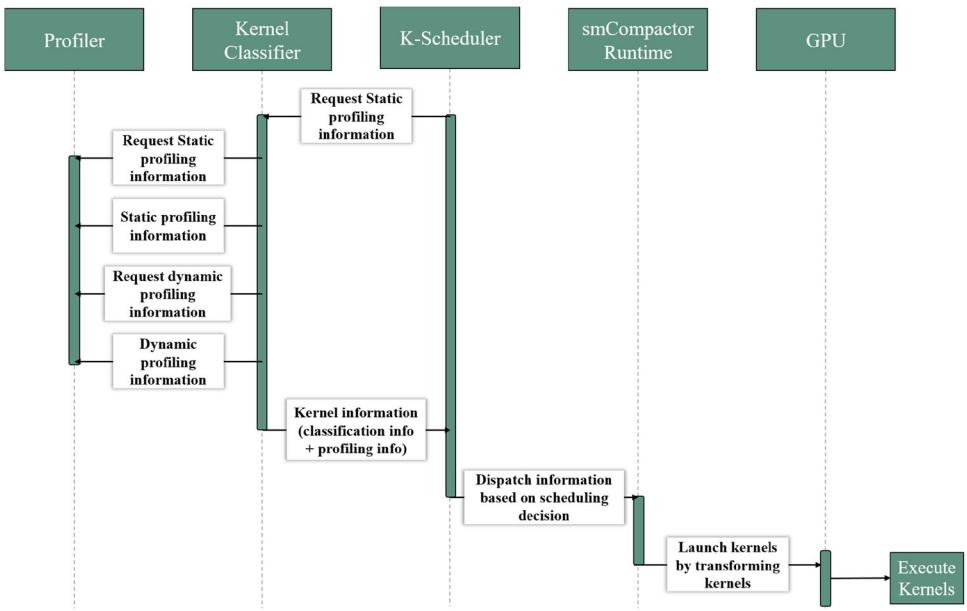

Fig. 9 Sequence diagram of K-Scheduler framework

amount of resources allocated to each application is increased, it is determined that the performance saturation point was reached and no more resources are allocated. The vector *Perf* stores the performance according to the number of active SMs and TBs of each application. $Perf_{i,j}$ indicates the performance when i active SMs and j active TBs are allocated. $Rate_{tb}$ is the rate at which the point of saturated performance is assessed, and it determines the allowable level of performance loss. A smaller value means that a smaller performance loss is allowed. w denotes the window size. In the case of a specific application, the performance may not increase much when one active TB is added; however, the performance gain may be large when two or three TBs are added. Hence, this is considered for the window size. When the following equation is satisfied, the number of TBs where the performance is saturated for i active SMs is determined as j.

$$Perf_{i,j} * (1 + Rate_{tb})^{w} \geq Perf_{i,j+w} \quad w = 1, 2, ..., win\_size \tag{1}$$

The value of w is increased from 1 to the window size (*win_size*). This equation determines whether the performance is improved for a certain ratio (*Rate_tb*) when the number of active TBs increases from j to j+w for i active SMs. If this equation is satisfied for every w value, it means that the performance did not improve for a certain ratio in the given window. Thus, it is determined that it is meaningless to give more resources, and the number of TBs that saturate the performance is determined as j. To find the saturated point by applying this algorithm, the performance value based on the number of active SMs and the number of active TMs is required. To measure performance according to the resource allocation, the profiling method proposed in [16] can be used to obtain the performance value with a small profiling overhead.

Algorithm 1 K-Scheduler algorithm

Input: $Workload = \{K^1, K^2, \ldots, K^k\}$
Output: CK_List
▷ $CK\_List = \{CK_1 = \{SK^1, \ldots, SK^m\}, \ldots, CK_n\}$
1: $SK\_List \leftarrow sort\_kernels(Workload)$
▷ $SK\_List = \{SK^1, SK^2, \ldots, SK^k\}$
2: **while** $SK\_List \neq \emptyset$ **do**
3: $CK \leftarrow \emptyset$
4: **for** $SK^i \in SK\_List$ **do**
5: **if** rule #1 and rule #2 and rule #3,4,5 is satisfied **then**
6: move SK^i to CK
7: **end if**
8: **end for**
9: add CK to CK_List
10: **end while**

5.3 K-Scheduler

This section explains the K-Scheduler algorithm, which places multiple heterogeneous kernels in one SM by considering the static resource usage and runtime execution pattern to improve the performance of concurrent execution. In addition, the rules that are required to apply this algorithm are defined.

5.3.1 Algorithm

The algorithm for K-Scheduler is shown in Algorithm 1. The input of this algorithm is workload, which is a set of kernels for which the identification and classification of performance saturation points are completed. The input kernels are sorted, where the L1 cache-intensive application has the highest priority, the memory-intensive application has the next priority, and the computationally intensive application has the lowest priority. If applications of the same category have the same priority, they are sorted with the longest job as the first (line 2). The set of applications for concurrent execution are found by repeating this process until no kernel remains in the *SK_List* (line 3). The combination of applications to be found in this repeated process is called CK, which is initialized to an empty set (line 4). To find the kernel to be included in CK, the elements of *SK_List*, SK^i are checked to determine whether the kernel satisfies rules #1 to 5. If all the rules are satisfied and it is determined as a kernel for concurrent execution, SK^i is moved from *SK_List* to *CK* (lines 5–9). Once the For loop is completed, a *CK* is composed of kernels for multitasking and is added to *CK_List* (line: 10). The output of the algorithm is *CK_List*, which is a list of the combinations of kernels for concurrent execution; multitasking is performed according to the sequence and combinations of this list. =To analyze its time complexity, sorting kernels(line: 1) takes $O(NlogN)$. The worst case in line 2-10 is executing all the kernels exclusively and its time complexity is $O(N^2)$. Therefore, the overall time complexity of Algorithm 1 is $O(N^2)$.

Figure 10 shows the scheduling method of K-scheduler when the kernels of $K_0, \ldots, K_5$ are sequentially submitted, and the results are compared with the conventional scheduling method. Figure 10a shows the single execution method supported by the conventional hardware queue, which does not support multitasking and the sharing of intra-SM resources without the assistance of special skills. Therefore, every kernel exclusively occupies SM and is executed sequentially. Figure 10b shows the Warped-slicer method. The number of concurrent execution kernels of Warped-slicer is always set to K. In this figure, the scheduling layout shows the case where K is set to 3. Three kernels form a combination according to the order of submission, and the best resource partitioning is found according to the mechanism of single profiling and Warped-slicer for each kernel. Figure 10c shows the method employed by K-Scheduler. The result of $CK\_List = \{\{K_0, K_3, K_5\}, \{K_2, K_4\}, \{K_1\}\}$ is obtained by applying the scheduler algorithm to the submitted kernels, and the TBs of each kernel are dispatched to the SM according to the scheduling information. K-Scheduler may not dispatch TBs to every SM dynamically according to the kernel profiling information, and the number of kernels allocated to the SM is also not fixed. As shown in the figure, for both Warped-slicer and K-Scheduler, all

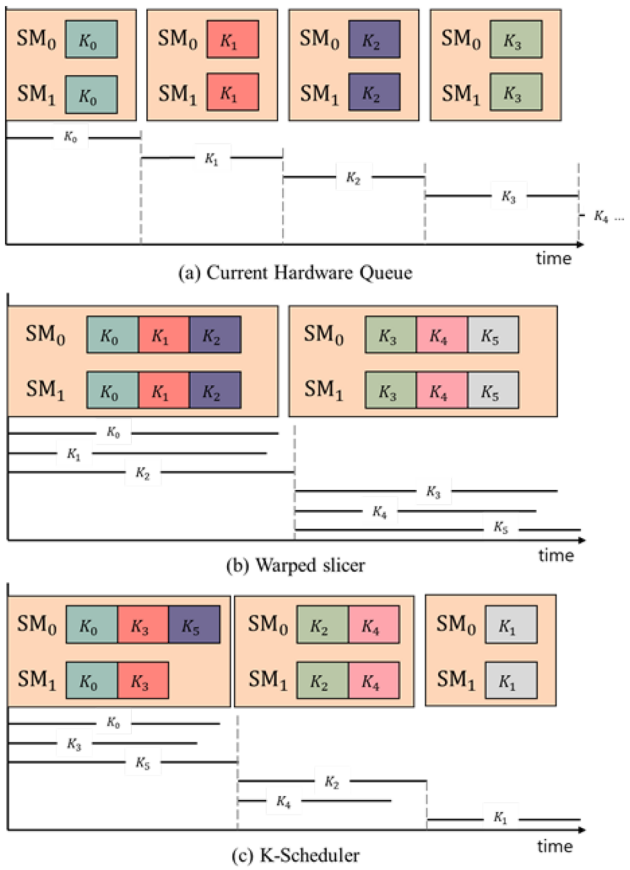

Fig. 10 Comparison of scheduling method for **a** hardware queue, **b** Warped-slicer, and **c** K-Scheduler

applications in one application set must be completed before the next application set can be started. A dynamic SM sharing method using preoccupation has been proposed [27], but it is not supported by the current hardware, and the effects were proven through simulations. Dynamic scheduling in hardware is a major issue to be addressed in the future.

5.3.2 Static Rules

Rule # 1 *A kernel cannot use static resources of the GPU that exceed the provided amount.*

Rule #1 checks whether there are sufficient static resources in the GPU considering static profiling information. As described in Sect. 2, the maximum number of TBs that can be activated in one SM is determined by the static resource constraints of the GPU (number of registers, size of sharing memory, and limited maximum number of TBs). Therefore, in Rule #1, the set of kernels selected up to the present for concurrent execution for scheduling K^i is $CK = \{CK^1, CK^2, \ldots, CK^m\}$. Here, m is the number of kernels that are selected up to the present. Whether or not K^i can be added as an element of the CK set based on the static profiling information is determined as follows:

$$\sum_{j=1}^{m} CK^j_{reg} + K^i_{reg} < MAX\_REG \tag{2}$$

$$\sum_{j=1}^{m} CK^j_{smem} + K^i_{smem} < MAX\_SMEM \tag{3}$$

$$\sum_{j=1}^{m} CK^j_{tb} + K^i_{tb} < MAX\_TB \tag{4}$$

The first equation determines whether the sum of the total number of registers requested by the selected kernels until now and K^i_{reg}, which is the number of registers requested by K^i, does not exceed *MAX_REG*, which is the maximum number of registers that can be provided by the hardware at present. The second equation determines whether the sum of the total sharing memory size of selected kernels and the K^i_{smem}, which is the sharing memory size used by K^i, is not smaller than *MAX_SMEM*, the maximum size of the sharing memory. The third equation determines whether the sum of the total number of allocated TBs of the kernels, which are elements of the CK set, and the number of allocated TBs of K^i, is smaller than *MAX_TB*, the maximum number of TBs. If these three equations are all satisfied, Rule #1 is satisfied, and this means that concurrent execution with the kernels of CK is possible when only the static profiling information of K^i is considered.

Rule # 2 *A kernel performance cannot exceed the total system performance.*

Rule #2 checks to determine whether there is sufficient available bandwidth and FLOPS (floating point operations per second) in the GPU considering the system performance requirement according to Observation #2. As described earlier, the set of kernels selected until now is CK. When scheduling for K^i is performed, the following equations are used to verify whether the supply performance of the theoretical DRAM bandwidth and GFLOPS (Giga FLOPS) is not exceeded when the kernels are scheduled:

$$\sum_{j=1}^{m} CK^j_{BW} + K^i_{BW} < MAX\_BW \tag{5}$$

$$\sum_{j=1}^{m} CK^j_{GFLOPS} + K^i_{GFLOPS} < MAX\_GFLOPS \tag{6}$$

The first equation checks whether the sum of the bandwidth requirements of the selected kernels until now and the bandwidth requirements required for performing K^i exceeds the DRAM bandwidth provided by the hardware. In addition, the second equation schedules concurrent execution when the calculated performance of the requirements of the current and selected kernels is lower than the supply of GFLOPS provided by the system. The issue of whether or not to perform concurrent execution is determined by comparing the system performance requirements of kernels and the amount of supply for the system.

5.3.3 Dynamic rules

Based on Observations #3, 4, and 5, Rules #3, 4, and 5 are defined as follows:

Rule # 3 *A L1 cache-intensive kernel does not multitask with a kernel exceeding the reference point based on the number of L1 cache transactions.*

Rule # 4 *Multitasking among memory-intensive kernels is not desirable.*

Rule # 5 *A computation intensive kernel for which the EPC exceeds the reference point (max threshold) cannot multitask with a kernel for which the EPC is higher than the reference point (base threshold).*

The method of applying Rules #3, 4, and 5 is described using the algorithm in Algorithm 2.

The set of kernels selected until now is CK. The scheduling for K^i is performed according to the categories of K^i (line 2). First, if K^i is a L1 cache-intensive kernel, whether a L1 cache-intensive kernel exists in CK is

checked using the $L1\_in\_CK()$ function. If $L1\_in\_CK()$ is true, L1 cache-intensive kernels cannot be placed together, and the algorithm returns false (lines 3–7). If K^i is not a L1 cache-intensive kernel, we first check whether the L1 cache-intensive kernel exists in CK. If there is an L1 cache-intensive kernel, $Can\_placed\_with\_l1\ (K^i)$, which tests whether it is possible to place the cache transaction and L1 cache-intensive kernel together, is called. If multitasking is impossible because the L1 cache transaction exceeds the base line, the algorithm returns false according to Rule #3 (lines 9–10, 16–17). Consequently, the L1 cache-intensive kernel, whose performance in concurrent execution may be the worst of the kernels, will be scheduled with the highest rank. If K^i is a memory-intensive kernel, the $M\_in\_CK()$ function is used to determine whether a memory-intensive kernel exists in *CK*. If this function returns true, the concurrent execution of memory-intensive kernels is not allowed according to Rule #4, and the algorithm returns false (lines 11–13). If the category of K^i is Compute, $Over\_max\_in\_CK()$ is used to verify whether a kernel whose EPC is higher than the max threshold exists in CK. When this condition is satisfied, false is returned if K^i_{epc} exceeds the *BASE* according to Rule #5 (line 18–19). If there is a kernel exceeding the base threshold, whether K^i_{epc} exceeds *MAX* is tested according to Rule #5, and false is returned if it exceeds *MAX* (line 20–21). If the aforementioned detailed conditions are satisfied, K^i can obtain the gain of concurrent execution while minimizing resource contention with the kernels of *CK*, and true is returned. The time complexity of the algorithm is $O(1)$ for judging whether K^j is included in *CK*.

Algorithm 2 Algorithm for applying rules #3, 4, and 5

Input: K^j, CK
1: **switch** K^i_{cat} **do**
2: **case** L1 cache
3: **if** $L1\_in\_CK()$ **then**
4: **return** false ▷ Rule #3
5: **end if**
6: **case** Memory
7: **if** $L1\_in\_CK()$ and not $Can\_placed\_with\_l1(K^i)$ **then**
8: **return** false ▷ Rule #3
9: **else if** $M\_in\_CK()$ **then**
10: **return** false ▷ Rule #4
11: **end if**
12: **case** Compute
13: **if** $L1\_in\_CK()$ and not $Can\_placed\_with\_l1(K^i)$ **then**
14: **return** false ▷ Rule #3
15: **else if** $Over\_max\_in\_CK()$ and $K^i_{epc} > BASE$ **then**
16: **return** false ▷ Rule #5
17: **else if** $Over\_base\_in\_CK()$ and $K^i_{epc} > MAX$ **then**
18: **return** false ▷ Rule #5
19: **end if**
20: **return** true

6 Experiments and analysis

6.1 Experiment method

6.1.1 Experimental setup

This experiment was performed using NVIDIA CUDA version 10.0 in an Ubuntu 16.04 environment equipped with a NVIDIA Titan XP GPU having 12 GB of memory. The static resources of the GPU are listed in Table 5. The K-Scheduler is implemented on the scheduling framework based on the TB of smCompactor [26].

6.1.2 Experimental workload

The applications used in this experiment are listed in Table 2, which are the benchmarks of NVIDIA CUDA Sample [18], Rodinia GPU benchmark suite [17], Parboil benchmark [19], Polybench [21], and SHOC benchmark [20]. All applications were executed using the standard data input set.

Nine workloads were selected for the corresponding benchmarks. The characteristics of each workload are summarized in Table 6.

6.1.3 Evaluation metrics

- Weighted speedup [16]: The total execution time normalized by the sequential execution time. A higher value indicates a better performance.
- Average normalized turnaround time (ANTT) [14]: The average of each application's turnaround time. A lower value indicates a better performance.
- Fairness [15]: (Min speedup)/(Max speedup) This indicates the difference between the minimum and maximum speedups. It has a value between 0 and 1. A value closer to 1 means higher fairness.

6.1.4 Baseline scheduler

The compared schedulers are even partitioning scheduler, which allocates the same amount of intra-SM resources to

Table 5 Static resources of GPU

GPU memory	11.91 GB	Warps per SM	64
GPU speed	1582 MHz	Thread blocks per SM	32
GPU architecture	Pascal	Shared Memory per SM	96 KB
PCIe bandwidth	32 GB/s	Threads per SM	2048

Table 6 Intra-SM resources and runtime stall for each benchmark application

Workload sequence	Characteristics of workload
W1	Compute intensive applications
W2	Memory intensive applications
W3	Compute intensive applications + Memory intensive applications (1 : 1)
W4	Compute intensive applications + Memory intensive applications (2 : 1)
W5	Applications with large number of EPC (EPC>1)
W6	Applications with small number of EPC (EPC<1)
W7	Applications with large number of EPC + Applications with small number of EPC (1:1)
W8	Applications with large number of EPC + Applications with small number of EPC (1:2)
W9	Applications with large number of EPC + Applications with small number of EPC (2:1)

each application, and Warped-slicer [16], which was described in Sect. 3.2.

6.2 Scheduling performance

6.2.1 Comparison of scheduling performance

6.2.1.1 Comparison of weighted speedup Figures 11a and 12 show the weighted speedup of the baseline scheduler compared with K-Scheduler; the sequential execution time is expressed as 1 (the higher, the better). As shown in Fig. 11a, K-Scheduler, Warped-slicer, and even partitioning increases the performance by 32%, 16%, and 9%, respectively, on average compared to sequential execution. First, even partitioning could increase the resource utilization compared to the conventional hardware queue method, which is executed sequentially. Warped-slicer exhibited an improved performance compared to even partitioning by recognizing the performance loss according to the resource allocation of each kernel for scheduling. However, it showed a real performance that is different from the theory because it did not consider resource contention among the kernels, and it shows that there is a limit to the performance improvement. Meanwhile, K-Scheduler, which minimizes resource contention between the performance and kernels according to the resource allocation of each kernel, showed the best performance. For in-depth analysis, Fig. 12 shows the weighted speedup for each workload. Among all the workloads, W1 is composed of computationally intensive applications. With respect to W1, there is little difference in performance between even partitioning and Warped-slicer. This indicates that it is more important for multitasking to determine with which application to perform concurrent execution rather than the resource allocation method. The scheduler proposed in this study could maximize performance under the given workload because it performs scheduling while recognizing the potential for resource contention owing to concurrent execution, although concurrent execution between computationally intensive applications can realize a performance gain. The workload W2 is composed only of memory-intensive applications, and shows that multitasking between memory-intensive applications has no gain of concurrent execution. Because K-Scheduler does not perform concurrent execution between memory-intensive applications, it shows the same performance as that of sequential execution. The workload W7 is composed of applications with a low EPC and applications with a high EPC at the ratio of 1:1. Thus, it does not exhibit much difference in performance between schedulers. There is a gain of concurrent execution when applications that have a high EPC are executed concurrently with applications that have a low EPC. Meanwhile, because the workload has a balanced mixture of different applications in the same proportions, K-Scheduler does not have many opportunities to improve performance compared to Warped-slicer and even partitioning.

6.2.1.2 Comparison of ANTT Figures 11b and 13 show the ANTT of each scheduler. A value closer to 1 means that individual performance is preserved. As shown in Fig. 11b, K-Scheduler, Warped-slicer, and even partitioning have average ANTT values of 1.31, 2.23, and 2.57, respectively. Even partitioning exhibited a significantly delayed response time because it was slowed down by a factor of 2.56 times in terms of individual application because it performed scheduling by evenly distributing only static resources. Warped-slicer did not show a significant improvement in ANTT compared to even partitioning, thus showing that the service response time is not guaranteed. Meanwhile, K-Scheduler guarantees the quality of service (QoS) for individual applications. In particular, in the case of W7, the weighted speedup of K-Scheduler only improved by approximately 0.3% and

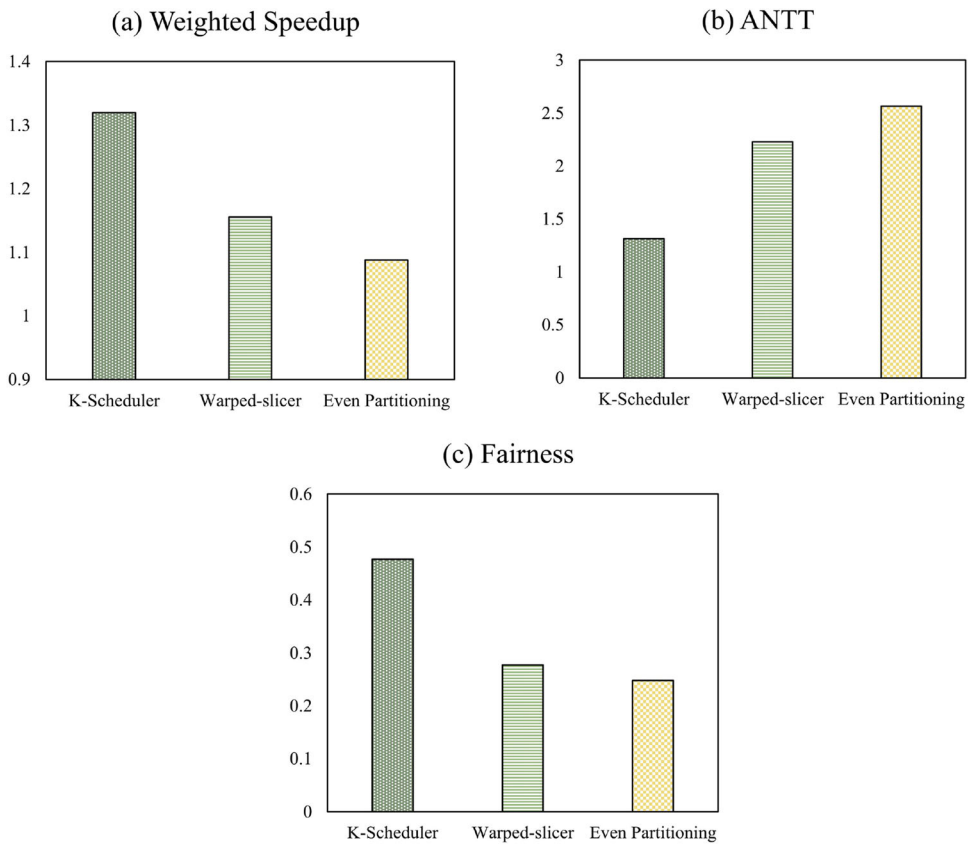

Fig. 11 Scheduling performances of K-Scheduler, warped-slicer, and even partitioning based on **a** weighted speedup, **b** ANTT, and **c** fairness

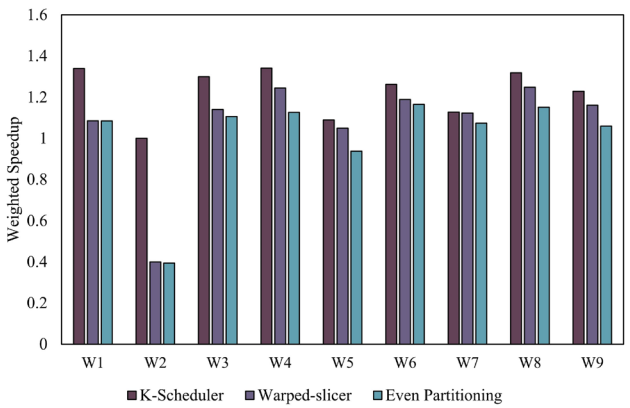

Fig. 12 Comparison of weighted speedup for each workload

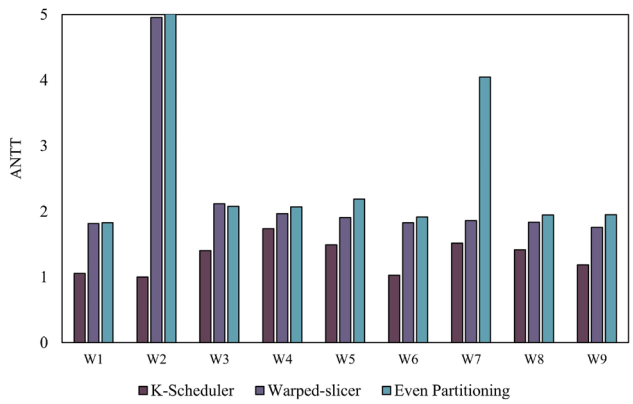

Fig. 13 Comparison of ANTT for each workload

5% compared to Warped-slicer and even partitioning as shown in Fig. 13. Thus, the performance improvement of the total workload was not large. However, the ANTTs improved by 22% and 166%, respectively, showing the excellence of K-Scheduler. In the case of W2, the ANTTs of the baseline scheduler were 4.95 and 0.06, respectively. Thus, the multitasking of the memory-intensive application shows a decrease not only in the total workload performance, but also in the performance of the individual kernel. The workload W6 is only composed of kernels with a low EPC, showing the smallest values of ANTT with 1.02, 1.8, and 1.9 for K-Scheduler, Warped-slicer, and even partitioning, respectively. The degradation of individual performance is the smallest because many stalls occur for the kernels with a low EPC, and these are hidden owing to multitasking.

6.2.1.3 Comparison of fairness Figures 11c and 14 show the average fairness and the fairness of each scheduler. A fairness value closer to 1 means that the difference between the speedups of each application is not large. As shown in Fig. 11c, the fairness of K-Scheduler is 0.48, whereas the

fairness values of Warped-slicer and even partitioning are 0.28 and 0.25, respectively. This indicates that K-Scheduler can perform more fair resource sharing between applications than Warped-slicer and even partitioning. In W7, the fairness was low because there is a pair comprising an application with a long execution time and an application with a short execution time. However, excluding this workload, K-Scheduler showed the fairest results under every workload.

6.2.2 Evaluation and analysis of K-Scheduler design

6.2.2.1 Variable number of concurrent execution kernels

This section describes how the variable number of concurrent execution kernels of K-Scheduler affects the scheduler's performance. First, as mentioned in Section 5.3.1, Warped-slicer, which is the baseline scheduler, performs scheduling with a fixed number of concurrent execution kernels, K. This scheduler has a trade-off depending on the K value in terms of the performance of the total workload and the performance of individual kernels. As shown in Fig. 15a, from the perspective of the weighted speedup, it is 1.02 when K is set to 2, and 1.09 when K is set to 3. When the K value is increased, multiple kernels share resources with low utilization. As a result, the performance of the total workload improved, and this increased the weighted speedup. However, as shown in Fig. 15b, the ANTT is 1.4 at K = 3 and 1.73 at K = 2. Thus, when the K value increases, the performance of the individual kernel decreases. This is because when the number of kernels for concurrent execution increases, limited resources must be shared by multiple kernels; the performance of individual kernels decreases as a result. However, K-Scheduler solved this problem by dynamically changing the K value instead of fixing it. The weighted speedup and ANTT of K-Scheduler were 1.2 and 1.3, respectively. Thus, the weighted speedup improved compared to warped slicer at K = 3 and K = 2, respectively. K-Scheduler increases the number of concurrent execution kernels for the combination of kernels with low resource contention, and decreases the number of kernels for a combination of kernels with a high resource contention. This variable number of K results in good performance in terms of both the total workload and the workload of individual kernels.

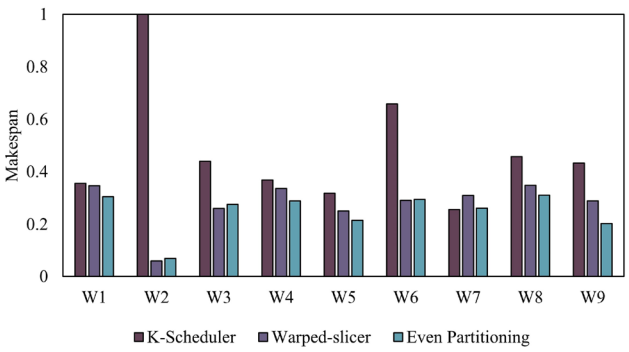

Fig. 14 Comparison of fairness for each workload

6.2.2.2 Hierarchical scheduling with rules

This section describes the change of performance according to the K-Scheduler's scheduling rules (Fig. 16). First, K-Scheduler (rule 1) only considers static resources. The weighted speedup is 0.55, which is a performance that is two times worse than the sequential performance. In terms of single kernel, it has an ANTT of 2.88, showing a significant degradation of the performance of individual applications. Multitasking, which only considers static resources, exhibits severe performance degradation owing to contention among resources used by each application. In particular, K-Scheduler does not set the number of multitasking kernels, and kernels using static resources perform multitasking with multiple kernels using static resources. Consequently, this study performed multitasking with up to seven kernels, thus showing a significant performance degradation. Thus, scheduling that considers static resources only can result in a significant performance degradation. The scheduler (Rules 1 and 2) that considers the system performance supply and static resources shows a speedup that is approximately 8% higher than sequential execution; individual applications show a 54% lower performance on average. This is better than the performance of K-Scheduler (Rule 1), which does not consider dynamic resources; however, there is a limit to the improvement in the performance in terms of both the total workload performance and individual kernels. Complex contention occurs not only for the supply of global memory bandwidth and computing resources, but also in other resources, and scheduling needs to consider this. K-Scheduler can achieve a weighted speedup of 1.2 and an ANTT of 1.3. Therefore, K-Scheduler can achieve a good performance through scheduling that considers static and dynamic resources, the classification of applications, and runtime characteristics.

7 Related studies

As NVIDIA introduced the Hyper-Q technology that supports spatial diversification [25], there have been ongoing studies on developing a method that effectively performs spatial diversification using it. Spatial diversification techniques are classified into methods that divide and use subsets of SM [11–13], and ones that do the space inside SM [14–16].

CD-Search [13] classifies applications into computationally intensive and memory-intensive applications to

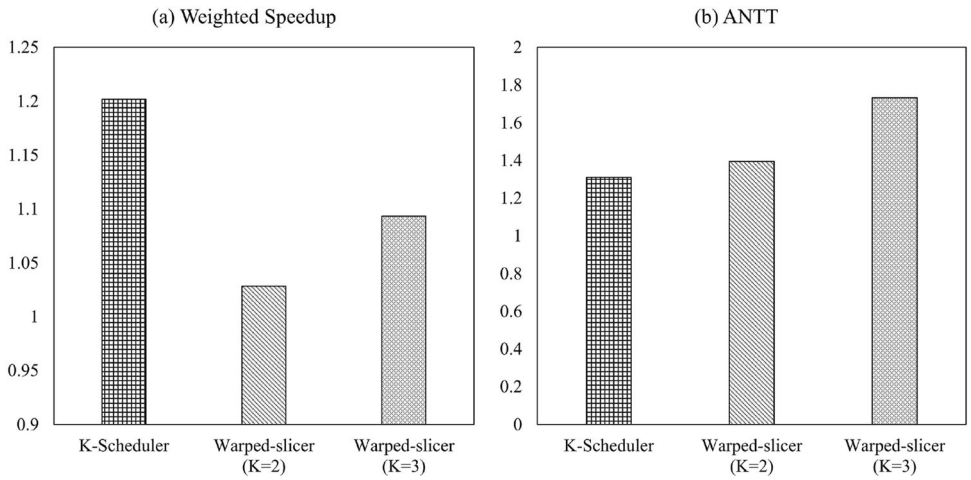

Fig. 15 Comparison of fairness for each workload based on **a** weighted speedup and **b** ANTT

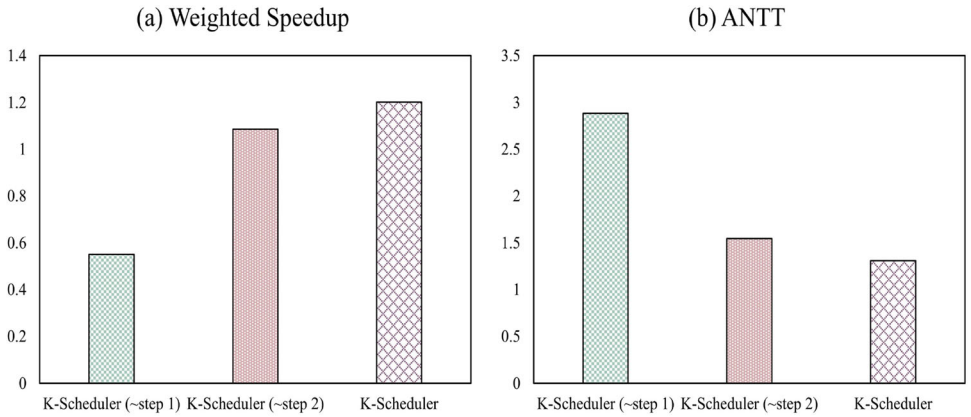

Fig. 16 Comparison of scheduling by rule application step based on **a** weighted speedup and **b** ANTT

efficiently divide the subsets of SM. Although previous studies classify applications based only on the DRAM throughput, this study classified applications by considering the off-SM bandwidth as well. Furthermore, a power mode that can save power was proposed by considering the fact that the memory-intensive application can achieve a similar performance, even when a small number of SMs are allocated. Thomas et.al. [28] utilizes a roofline model to find resource bottleneck, which allows for more efficient resource allocation, compared to CD-Search.

Themis [29] proposes a four-layer neural network for predicting slowdown in multitasking. It limits the number of parameters to reduce over heads. However, it performs hundreds of floating-point operation to predict single value. HSM [12] creates a performance model by combining white box and black box methods, and predicts the performance when the SM subsets are divided and executed simultaneously, compared to the independent execution of each application. This method predicts the utilization of DRAM bandwidth, and they propose a multitasking scheduler that achieves fairness of QoS based on it. Laius [11] predicts the work performance to achieve the QoS of queries for users, and allocates resources that recognize the contention of resources accordingly. Furthermore, the progress of jobs is monitored continuously. If there is a delay in the progress of a task relative to the predicted value, the allocated amount of computation resources is increased to compensate for the delay. To this end, they introduce an online method as well as an offline method to build a runtime system. For the multitasking method that divides SM subsets, there is a limit to performance improvement in terms of resource utilization and throughput compared to the method of sharing intra-SM resources [15].

Warped-slicer [16] focuses on the method of efficiently sharing resources in the SM. Applications are classified using L2 miss per kilo warp instruction. The most efficient allocation method is found when sharing resources by two applications using the water-filling algorithm. [15] converts the context in the TB unit for resource sharing in the SM; they propose a scheduling algorithm using static resource usage and a dynamic computing cycle. However, the dynamic resource allocation method using the computing cycle assumes a linear increase when estimating the execution time. The observations of this paper indicate that the performance does not increase linearly with the resource

Table 7 Comparison of fine-grained scheduler

	K-Scheduler	smCompactor [26]	Warped-slicer [16]	HSM [12]	Hongwen et al. [14]
Sharing method	Intra-SM sharing	Intra-SM sharing	Intra-SM sharing	Spatial multitasking	Intra-SM sharing
Evaluation benchmarks	Rodinia, Parboil, polybench, CUDA SDK, SHOC	Rodinia, CUDA SDK	Rodinia, Parboil, ISPASS, CUDA SDK	Rodinia, Parboil, PolyBench, Mars, CUDA SDK	Rodinia, Parboil, CUDA SDK
Evaluation tools	Experiment on real GPU using Persistent thread model	Experiment on real GPU using Persistent thread model	Simulation	Simulation	Simulation
Advantage	Minimize interference by predicting multitasking performance on real GPU	Make intra-SM sharing possible on real GPU	Propose Intra-SM sharing method	Predict multitasking performance	Propose methods for minimizing interference
Disadavantage		Not include scheduling mechanism for intra-SM sharing	May be different between actual performance and theoretical performance	Show low internal utilization of SM	Need hardware modification
Performance metrics	ANTT, Speed up, Fairness	Execution time	IPC, ANTT, Fairness	Fairness, ANTT, STP	System Throughput, ANTT

allocation, and there is a section where performance is saturated. This indicates the need for a resource allocation method that recognizes these sections. In Refs. [15, 16], the authors introduce a methodology for allocating intra-SM resources, but does not consider interference that occur between jobs that are placed together. Hence, the concurrent execution performance degrades owing to contention for resources. Dai el.al. [14] points out that theoretical and real performances differs because the interference between concurrent kernels cannot be solved using only the method that allocates resources for sharing intra-SM resources. To solve the interference issue, they proposed the balanced submission of memory requests and the restriction of memory commands executed in individual kernels. This is impossible to implement in actual hardware because additional hardware and hardware changes are required to control the memory request and memory commands. However, the method proposed in our study attempted to minimize interference while working in actual hardware. Alizadeh et. al. [30] predicts interference which may be caused by resource contention in fine-grained sharing of SMs using machine learning. It shows up to 91.7% accuracy. It only classifies applications and is not about scheduling mechanism.

Refs. [14–16, 11–13] all conducted experiments in simulators, and the study results may differ from the actual hardware experiment results. Furthermore, a dynamic SM sharing method using preoccupation has been proposed [27]. However, its effect has been demonstrated only through simulations; it is not currently supported by hardware. smCompactor [26] proposed a scheduling framework that is based on TB to share SMs and intra-SM resources. This study improved the utilization of intra-SM resources by enabling the sharing of intra-SM resources in actual hardware, but it did not consider the placement method that recognizes the resource contention of each kernel. Furthermore, studies on various fine-grained scheduling methods are compared with this study in Table 7.

8 Conclusion

Existing studies have limitation with respect to achieving performance improvements through scheduling when many requests from clients are received because they focused on a partition technique of intra-SM resources. This study proposed K-Scheduler, which is a multitasking placement scheduler. The characteristics of resource use and the concurrent execution of applications were analyzed according to the individual execution characteristics of applications, and a scheduling method was introduced by inferring rules according to each observation. Experiment results demonstrated that K-Scheduler improved the total workload execution performance by 18% compared to previous studies. In addition, it improved the performance

of individual execution by 32%. Thus, the performance did not decrease significantly when it was executed independently. Using this scheduler, each application could receive fast responses, and the total workload throughput was also improved. For future research, we will conduct experiments by subdividing the application characteristics to generalize the proposed scheduling framework.

Funding This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1A2C1003379).

Data availability The datasets generated during and/or analysed during the current studyare available from the corresponding author on reasonable request.

Declarations

Informed consent Written informed consent for publication of this paper was obtainedfrom Sookmyung Women's University and all authors.

References

1. TOP-500. https://www.top500.org/
2. Dean, J., Barroso, L.A.: The tail at scale. Commun. ACM **56**(2), 74–80 (2013)
3. EC2 ELASTIC GPUS, A. (2017). https://aws.amazon.com/ec2/Elastic-GPUs/
4. NIMBIX. https://www.nimbix.net/cloud-computing-nvidia/
5. MICROSOFT-AZURE. https://docs.microsoft.com/en-au/azure/virtual-machines/windows/sizes-gpu
6. Dongarra, J.J., Luszczek, P., Petitet, A.: The linpack benchmark: past, present and future. Concurr. Comput.: Pract. Exp. **15**(9), 803–820 (2003)
7. Dongarra, J., Heroux, M.A.: Toward a new metric for ranking high performance computing systems. Sandia report, SAND2013-4744 312, 150 (2013)
8. Allen, T., Feng, X., Ge, R.: Slate: enabling workload-aware efficient multiprocessing for modern gpgpus. In: 2019 IEEE international parallel and distributed processing symposium (IPDPS), pp. 252–261. IEEE
9. NVIDIA-MULTI-PROCESS-SERVICE. (2020). https://docs.nvidia.com/deploy/pdf/CUDA-Multi-Process-Service-Overview.pdf
10. Schulte, M.J., Ignatowski, M., Loh, G.H., Beckmann, B.M., Brantley, W.C., Gurumurthi, S., Jayasena, N., Paul, I., Reinhardt, S.K., Rodgers, G.: Achieving exascale capabilities through heterogeneous computing. IEEE Micro **35**(4), 26–36 (2015)
11. Zhang, W., Cui, W., Fu, K., Chen, Q., Mawhirter, D.E., Wu, B., Li, C., Guo, M.: Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In: Proceedings of the ACM international conference on supercomputing, pp. 58–68 (2019)
12. Zhao, X., Jahre, M., Eeckhout, L.: Hsm: A hybrid slowdown model for multitasking gpus. In: Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems, pp. 1371–1385 (2020)
13. Zhao, X., Wang, Z., Eeckhout, L.: Classification-driven search for effective sm partitioning in multitasking gpus. In: Proceedings of the 2018 international conference on supercomputing, pp. 65–75 (2018)
14. Dai, H., Lin, Z., Li, C., Zhao, C., Wang, F., Zheng, N., Zhou, H.: Accelerate gpu concurrent kernel execution by mitigating memory pipeline stalls. In: 2018 IEEE international symposium on high performance computer architecture (HPCA), pp. 208–220. IEEE (2018)
15. Wang, Z., Yang, J., Melhem, R., Childers, B., Zhang, Y., Guo, M.: Simultaneous multikernel gpu: multi-tasking throughput processors via fine-grained sharing. In: 2016 IEEE international symposium on high performance computer architecture (HPCA), pp. 358–369. IEEE (2016)
16. Xu, Q., Jeon, H., Kim, K., Ro, W.W., Annavaram, M. Warped-slicer.: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In: 2016 ACM/IEEE 43rd annual international symposium on computer architecture (ISCA) (2016), pp. 230–242. IEEE (2016)
17. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K. Rodinia.: Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE international symposium on workload characterization (IISWC), pp. 44–54. IEEE (2009)
18. NVIDIA-CUDA-SAMPLE. https://docs.nvidia.com/cuda/cuda-samples/index.html
19. Stratton, J.A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G.D., Hwu, W.-M.W.: Parboil: a revised benchmark suite for scientific and commercial throughput computing. Center for Reliable and High-Performance Computing **127**,(2012)
20. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (SHOC) benchmark suite. In: Proceedings of the 3rd workshop on general-purpose computation on graphics processing units, pp. 63–74 (2010)
21. POLYHEDRAL-BENCHMARK-SUITE. http://web.cse.ohio-state.edu/pouchet.2/software/polybench/
22. PROGRAMMING GUIDE, C.-C. (2021). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
23. TUNING GUIDE, K. (2021). https://docs.nvidia.com/cuda/kepler-tuning-guide/index.html
24. Kim, S., Qichen Chen, H.Y., Kim, Y.: Performance analysis of concurrent multitasking for efficient resource utilization of gpus. J. KIISE **48**(6), 604–611 (2021)
25. NVCC. (2021). https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html
26. Chen, Q., Chung, H., Son, Y., Kim, Y., and Yeom, H.Y.: Smcompactor: a workload-aware fine-grained resource management framework for gpgpus. In: Proceedings of the 36th annual ACM symposium on applied computing, SAC '21, pp. 1147–1155 (2021)
27. Park, J.J.K., Park, Y., Mahlke, S.: Dynamic resource management for efficient utilization of multitasking gpus. In: Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems, pp. 527–540 (2017)
28. Thomas, W., Toraskar, S., Singh, V.: Dynamic optimizations in gpu using roofline model. In: 2021 IEEE international symposium on circuits and systems (ISCAS), pp. 1–5 (2021)
29. Wei, M., Zhao, W., Chen, Q., Dai, H., Leng, J., Li, C., Zheng, W., Guo, M.: Predicting and reining in application-level slowdown on spatial multitasking gpus. J. Parallel Distrib. Comput. **141**, 99–114 (2020)
30. Alizadeh, N.S., Momtazpour, M.: Machine learning-based interference detection in gpgpu concurrent kernel execution. In: 2020 25th international computer conference, computer society of Iran (CSICC), pp. 1–4. IEEE (2020)

Sejin Kim Sejin Kim is currently a Master degree in the Department of Computer Sci- ence, Sookmyung Women's University. She received her B.S. degree from Sook- myung Women's University in 2017. She is also researcher at the Distributed & Cloud Computing Lab of Sookmyung Women's University. Her research interests include management in cloud computing and heterogeneous systems.

Yoonhee Kim Yoonhee Kim she is the professor of Computer Science Department at SookmyungWomen's University. She received her Bachelors degree from Sookmyung Women's University in 1991, her Master degree and Ph.D. from Syracuse University in 1996 and 2001, respectively. She was a Research Staff Member at the Electronics and Telecommunication Research Institute during 1991 and 1994. Before joining the faculty of Sookmyung Women's University in 2001, she was the faculty of Computer Engineering dept. at Rochester Institute of Technology in NY, USA. Her research interests span many aspects of runtime support and management in distributed computing systems. She is a member of IEEE and OGF, and she has served on variety of program committees, advisory boards, and editorial boards.