Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

*Digital Object Identifier 10.1109/ACCESS.2021.Doi Number*

# Characterizing Fine-Grained Resource Utilization for Multitasking GPGPU in Cloud Systems

**Kyungwoon Cho and Hyokyung Bahn, Member, IEEE**
Department of Computer Engineering, Ewha University, Seoul, 120-750, Republic of Korea

Corresponding author: Hyokyung Bahn (e-mail: bahn@ewha.ac.kr).

**ABSTRACT** Managing GPGPU resources in cloud systems is challenging as workloads with various resource usage patterns coexist. To determine the co-location of workloads, previous studies have shown that run-time performance profiling and dynamic relocation of workloads is necessary due to interference between workloads. However, this makes instant scheduling difficult and also affects the performance of workload executions. In this article, we show that efficient resource sharing in GPGPU is possible without run-time profiling if resource usage characteristics of workloads are analyzed down to a fine-grained unit level. To extract workload characteristics, we do not perform profiling at scheduling time, but separates profiling from scheduling, thereby reducing the run-time complexity of previous approaches. Specifically, we anatomize the characteristics of various GPGPU workloads and present a new scheduling policy that aims at balancing resource utilization by co-locating workloads with complementary resource demands. Simulation experiments under various virtual machine scenarios show that the proposed policy improves the GPGPU throughput by 119.5% on average and up to 191.7%.

**INDEX TERMS** GPGPU, resource utilization, cloud system, multitasking, thread block scheduler.

## I. INTRODUCTION

With the rapid advances in many-core computing technologies, general-purpose GPUs (GPGPUs) have been widely adopted in cloud data centers as well as desktop systems. GPGPUs have a massive number of computing units, which allow thread-level parallelism for various application domains including deep learning, graphic rendering, and genome analysis [1, 2, 3].

Multitasking of heterogeneous workloads has not received much attention from traditional GPU management as GPUs are generally adopted in systems dedicated to specific workloads. However, due to the widespread adoption of cloud systems, heterogeneous workloads are concurrently executed within a GPGPU device, and thus maximizing resource utilization by multitasking in GPGPU has become an important issue [4, 5, 6]. As shown in Figure 1, modern cloud systems are equipped with GPGPU devices along with traditional host resources (CPU, memory, storage, etc.), and various workloads are executed as virtual machines that share resources.

Traditional host resources such as CPU, memory, and I/O devices in cloud systems can be shared among heterogeneous workloads through resource planning in the host operating system and virtualization software [7, 8]. However, this is challenging in GPGPU systems as the principle of resource management in GPGPU is to simplify hardware logic and maximize parallelism for a given workload rather than sharing resources for heterogeneous workloads. For this reason, some resources of the GPGPU may be exhausted whereas others are still under-utilized.

Previous studies have also focused on this issue, and the concurrent execution of multiple applications within GPGPU has been addressed [9, 10]. Specifically, SMK (simultaneous multi-kernel) and Maestro were introduced as software techniques to provide multitasking of heterogeneous workloads within SMs (streaming multiprocessors) [9, 10]. The basic principle of these techniques is to co-locate workloads with compensating resource usage in the same SM to balance the utilization of overall resources. Specifically, pairing memory-intensive and computing-intensive workloads in the same SM can significantly improve the overall utilization of resources.

Unfortunately, the resource under-utilization problem cannot be resolved if the bottleneck resource for all workloads is the same. That is, we cannot make use of the co-location strategy aforementioned if there are only computing-intensive

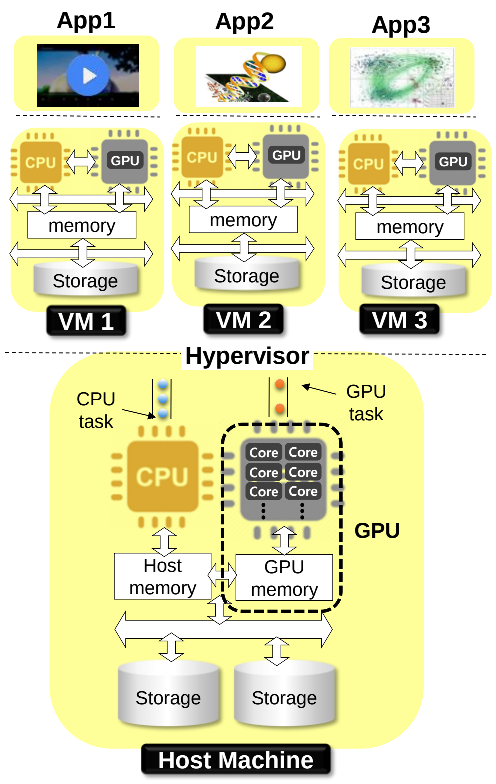

FIGURE 1. **A cloud host machine that equips the GPGPU device and various workloads are executed as virtual machines that share the resources including GPGPU.**

or memory-intensive workloads. Also, previous approaches require run-time performance profiling and dynamic re-arrangement of workloads to cope with interference between workloads. However, this makes instant scheduling difficult and also affects the performance of workload executions.

In this article, we show that efficient resource sharing in GPGPU is possible without run-time profiling if resource usage characteristics of workloads are analyzed down to a fine-grained unit level. Specifically, we analyze fine-grained resource usage patterns for a variety of GPGPU workloads and discuss how this analysis can be utilized in resource planning for multitasking GPGPUs. To extract workload characteristics, we do not perform profiling at scheduling time, but separates profiling from scheduling, thereby reducing the run-time complexity of previous approaches.

Let us consider a workload situation where only computing-intensive workloads exist in the system. Even in this case, our analysis shows that placing two computing-intensive workloads in the same SM is efficient if their bottleneck resources within the SM are different. For example, if one workload performs integer arithmetic and the other performs floating-point arithmetic, they can be assigned to the same SM without conflict as GPGPUs typically have separate units for integer and floating-point arithmetic within an SM.

In order to maximize resource utilization through multitasking, this article classifies GPGPU workloads into computing-bound, memory-bound, and dependency-latency-bound, and then refines the classification based on detailed resources that cause bottlenecks. In particular, we show that bottleneck resources can be different even for workloads with the same classification.

Based on this observation, we present a GPGPU workload placement scheme for cloud systems that assigns workloads with different bottleneck resources on the same SM in order to maximize overall resource utilization. Specifically, we design a thread block scheduling policy, called FRU-RR (Fine-grained Resource Utilization aware Round-Robin) that assigns pending thread blocks to SMs in a Round-Robin order but also considers each SM's fine-grained resource utilization.

Experimental results for various virtual machine scenarios show that FRU-RR improves GPGPU throughput by 119.5% on average and up to 191.7% compared to Round-Robin scheduler, and by 30.1% on average and up to 42.9% compared to previous studies that allow co-locations of workloads within the same SM [9, 10]. Our findings and contributions can be summarized as follows.

- Due to the coexistence of heterogeneous workloads and various resource types to manage, GPGPU scheduling in cloud systems incurs resource under-utilization problems.
- Previous approaches have addressed this issue, but they need run-time performance profiling and dynamic relocation overhead.
- We observe that efficient resource sharing in GPGPU is possible without run-time profiling if resource usage patterns are analyzed down to a fine-grained unit level.
- We separate profiling from scheduling, thereby reducing the run-time complexity of previous approaches, and quantify the resource usage patterns of various GPGPU workloads.
- We propose a scheduling policy that aims to balance resource utilizations by co-locating workloads with complementary resource demands, and validate it through a variety of cloud scenarios.

The remainder of this article is organized as follows. Section II briefly explains the internal structure of GPGPU devices and the CUDA platform architecture. Section III presents the analysis of various GPGPU workloads with respect to resource utilization. In Section IV, we explain the proposed thread block scheduling policy and conduct the validation of the policy by simulation experiments under various virtual machine scenarios. Section V summarizes previous studies related to this article, specially focusing on thread block scheduling. Finally, we conclude this article in Section VI.

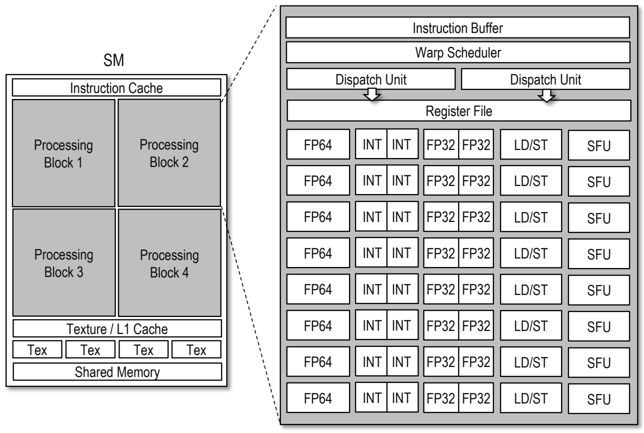

FIGURE 2. **Internal structure of SM (Stream Multiprocessor).**

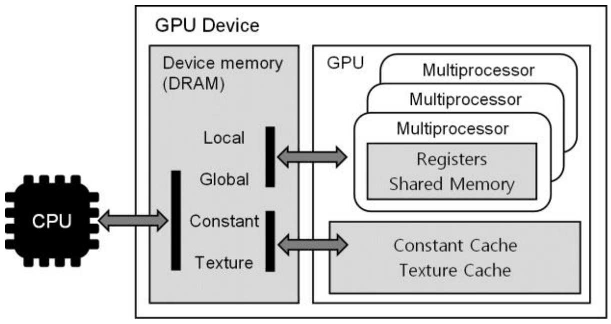

FIGURE 3. **Memory architecture of CUDA.**

## II. THE GPGPU ARCHITECTURE

### *A. INTERNAL STRUCTURE OF GPGPU*

Modern general-purpose GPUs (GPGPUs) have tens of thousands of computing units, which can accelerate the computing performance by executing threads in parallel [11]. In general, a GPGPU application has a large number of concurrent threads and they are grouped into thread blocks, which are basic resource allocation units. The GPGPU hardware consists of tens of stream multiprocessors (SMs), and thread blocks are allocated to the SMs by the thread block scheduler [12, 13, 14]. As SM adopts the SIMT (Single Instruction Multiple Thread) model, it executes the same instruction for multiple threads simultaneously [15]. The maximum number of threads that can be executed per SM is typically 2048. The thread block scheduler manages the number of allowable threads per SM not to exceed this limit while allocating thread blocks to the SM [16].

Meanwhile, a series of threads that should be executed simultaneously within the same hardware unit are called a Warp, which consists of up to 32 threads. Each thread block consists of at least one Warp, and the number of Warps increases by 1 whenever the number of threads within a Warp exceeds 32. Thus, SM may not execute all threads in a thread block simultaneously, but threads within the same Warp are essentially executed simultaneously [17].

Figure 2 shows the internal structure of a typical SM [18]. As shown in the figure, each SM consists of Instruction Cache, two or four Processing Blocks, Texture / L1 Cache, Texture Memory, and Shared Memory. Each Processing Block has various types of computing units such as FP64 cores, INT32 cores, FP32 cores, Load/Store units, and SFUs (special function units). Recently, a new type of computing units called Tensor cores, designed specifically for deep learning matrix operations, are increasingly being added to SMs [18].

Each Processing Block also has Instruction Buffer, Register File, Dispatch Unit, and Warp Scheduler. Dispatch Unit passes instructions to be executed to each core, and Warp Scheduler performs the scheduling of threads in Warp units.

### *B. CUDA PLATFORM AND MEMORY ARCHITECTURE*

CUDA (Compute Unified Device Architecture) is a parallel computing platform designed by NVIDIA [19]. As CUDA provides an application programming interface (API) model that supports industry standard languages like C, software developers can implement parallel processing algorithms in GPGPU efficiently.

Figure 3 shows the memory architecture of CUDA [19]. It consists of on-chip memory (right-hand side) and off-chip memory (left-hand side). On-chip memory consists of Registers, Shared Memory, Constant Cache, and Texture Cache. Registers are statically dedicated to each thread during thread block scheduling, and Shared Memory is used for inter-thread communication within the thread block.

Due to the limited capacity of Registers, the on-chip memory may be exhausted, and then Local Memory, which is one of the off-chip memory, can be used. Off-chip memory also has Global Memory that can be accessed by all threads, Constant Memory that is shared by all threads but read-only, and Texture Memory. For better memory performance, application developers should take full advantage of the memory structure and characteristics of CUDA.

## III. ANALYZING RESOURCE UTILIZATIONS OF GPGPU WORKLOADS

In this section, we analyze the fine-grained resource utilization of various GPGPU workloads in order to manage the GPGPU efficiently and improve resource utilization. As we increase the load on the GPGPU up to its resource limit, either a computing or a memory resource usually becomes the performance bottleneck. Based on this bottleneck resource, we classify GPGPU workloads into computing-bound and memory-bound workloads.

A workload is classified as computing-bound if the utilization of the computing resource is much higher than that of the memory resource, thereby reaching its limit first. In contrast, a memory-bound workload uses high memory bandwidth, which becomes a bottleneck resource. Meanwhile,

TABLE I
GPGPU WORKLOADS ANALYZED IN THIS ARTICLE.

| Workload | Kernel name | Benchmark | Grid size | Block size | Application domain |
|---|---|---|---|---|---|
| Lava MD | kernel_gpu_cuda | Rodinia 3.1 | [41, 1, 1] | [256, 1, 1] | Molecular dynamics |
| Nbody | integrateBodies | NVIDIA CUDA SDK | [16, 1, 1] | [256, 1, 1] | Simulation |
| Back Propagation | bpnn_adjust_weights_cuda | Rodinia 3.1 | [1, 4096, 1] | [16, 16, 1] | Pattern recognition |
| Hotspot | caculate_temp | Rodinia 3.1 | [43, 43, 1] | [16, 16, 1] | Physics simulation |
| Particle Filter | collideD | Rodinia 3.1 | [256, 1, 1] | [64, 1, 1] | Medical imaging |
| Srad | srad | Rodinia 3.1 | [450, 1, 1] | [512, 1, 1] | Image processing |
| Stream Cluster | kernel_compute_cost | Rodinia 3.1 | [128, 1, 1] | [512, 1, 1] | Data mining |
| Black Scholes | BlackScholesGPU | NVIDIA CUDA SDK | [15625, 1, 1] | [128, 1, 1] | Finance |
| Convolution Texture | convolutionColumnskernel | NVIDIA CUDA SDK | [192, 128, 1] | [16, 12, 1] | Image processing |
| HS Optical Flow | JacobiIteration | NVIDIA CUDA SDK | [20, 80, 1] | [32, 6, 1] | Image processing |
| Fluids GL | RealComplex_compute | NVIDIA CUDA SDK | [512, 1, 1] | [256, 1, 1] | Simulation |
| Heartwall | kernel | Rodinia 3.1 | [41, 1, 1] | [512, 1, 1] | Medical imaging |
| NN | euclid | Rodinia 3.1 | [168, 1, 1] | [256, 1, 1] | Data mining |
| B+tree | findK | Rodinia 3.1 | [10000, 1, 1] | [256, 1, 1] | Search |
| Conv2D | scudnn_relu_nn | PyTorch | [160, 1, 1] | [512, 1, 1] | Deep Learning |
| SHA256 | sha256_cuda | Public Domain | [80, 1, 1] | [1024, 1, 1] | Crytocurrency |

there are cases that neither the computing resource nor the memory resource becomes a bottleneck resource although we increase the load of the GPGPU up to its maximum capacity. In this case, resource utilization is limited by the execution dependency of the workload, which we classify as a dependency-latency-bound type.

We investigate the resource utilization of various GPGPU workloads by making use of the NVIDIA's profiling tool. Specifically, we execute 15 workloads consisting of NVIDIA SDK and Rodinia, which are popular benchmarks used for GPGPU performance evaluations [20, 21]. We also perform profiling for Conv2D, a convolutional neural network as a representative benchmark for machine learning, and SHA256, a secure hashing algorithm mainly used in Bitcoin's PoW (Proof of Work). Our execution configurations of the workloads are listed in Table I.

### *A. COMPUTING-BOUND WORKLOADS*

Figure 4 shows the utilization of computing and memory resources for the workloads classified as computing-bound out of the 17 workloads we considered. Low resource utilization here indicates that the resource has been idle for a long time, and 100% utilization means that the resource is a bottleneck. As shown in the figure, in computing-bound workloads, computing resources are bottlenecked, whereas memory resources are under-utilized.

The computing resources of GPGPU can be further classified into a single-precision arithmetic unit, a double-precision arithmetic unit, a control flow unit, a load store unit, and a special function arithmetic unit.

Figure 5 shows the detailed resource utilization of each computing unit for the computing-bound workloads in Figure 4. As shown in the figure, even for the same computing-bound workloads, the specific resource type that causes the bottleneck is different. Of the 7 workloads, Back Propagation, Hotspot, LavaMD, Particle Filter, and Srad exhibit high utilization of over 80% in double-precision arithmetic units, whereas other resources such as single-precision arithmetic units and special function arithmetic units show low utilization. Specifically, in the case of Back Propagation and LavaMD, the utilization of the double-precision arithmetic unit is almost 100%, whereas the utilization of the single-precision arithmetic unit and the special function arithmetic unit is 10% and 0%, respectively. From this analysis, we can summarize that most of computing-bound workloads we consider perform arithmetic operations on 64-bit double-precision data.

On the other hand, Nbody and Conv2D show significantly different resource usage patterns. Specifically, the utilization of the double-precision arithmetic unit is 0%, whereas the utilization of the single-precision arithmetic unit is about 80%. Thus, we can conclude that most computations of Nbody and Conv2D consist of 32-bit single-precision data.

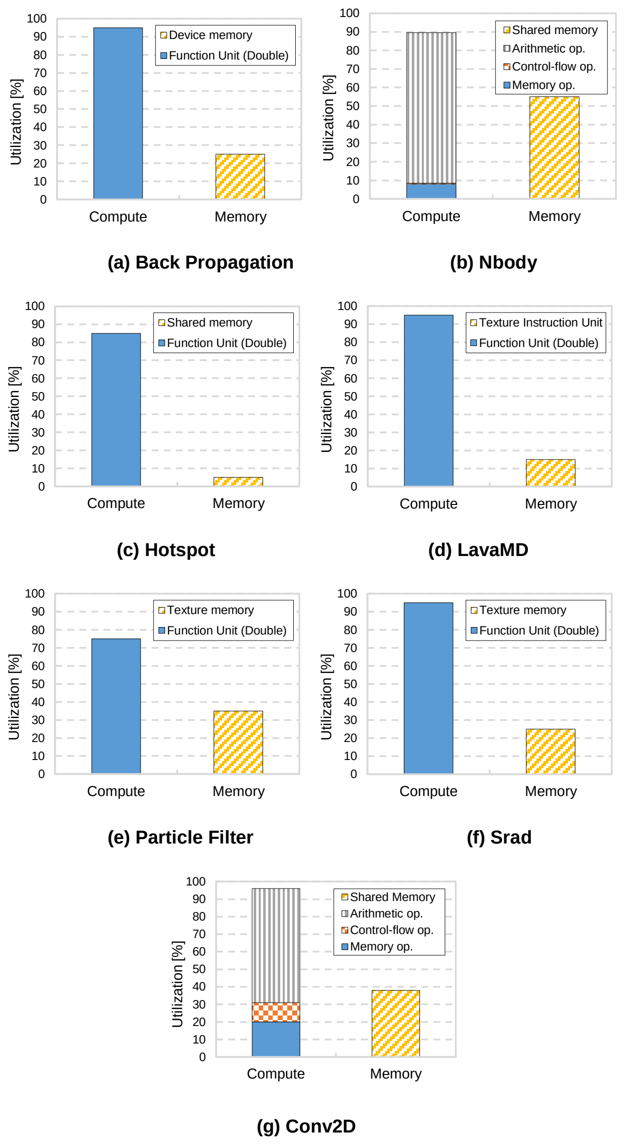

FIGURE 4. **Resource utilization of computing-bound workloads**

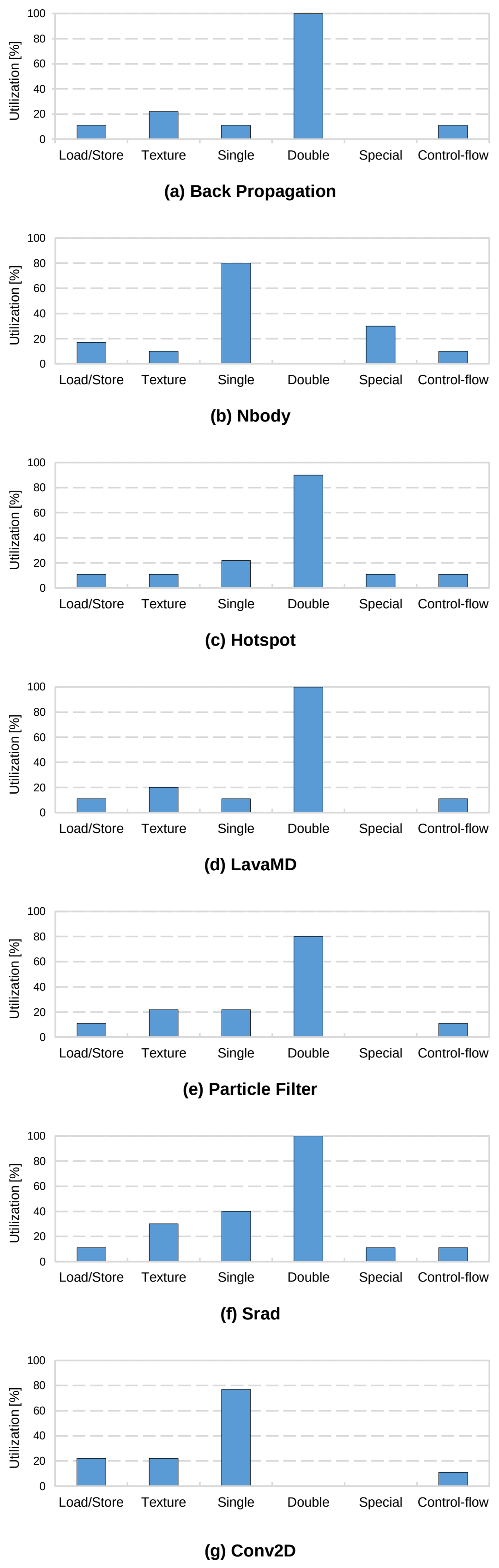

FIGURE 5. **Utilization of each computing resource.**

### *B. MEMORY-BOUND WORKLOADS*

Memory-bound workloads exhibit high memory bandwidth utilization, and thus performance is limited by memory resources within the GPGPU. This occurs when memory resources are not able to provide data at the speed of execution in computing resources. This lowers the utilization of computing resources, thereby limiting overall GPGPU performances.

In this subsection, we show resource utilizations for memory-bound workloads, and identify specific memory resources that cause performance bottlenecks. Specifically, we analyze the utilization of each memory resource in GPGPU, namely Shared Memory, Unified Cache, L2 Cache, Device Memory, and System Memory to identify the performance limiting factors.

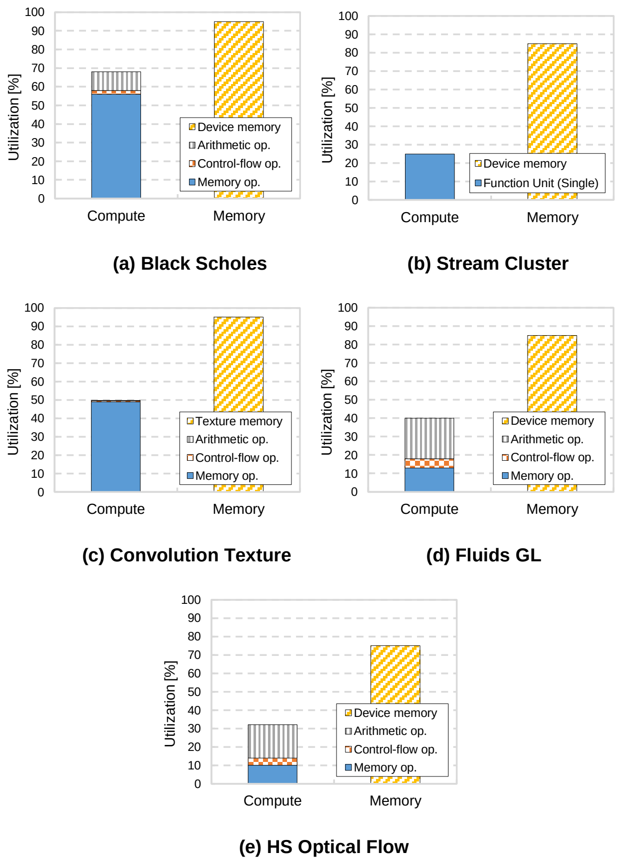

FIGURE 6. **Resource utilization of memory-bound workloads.**

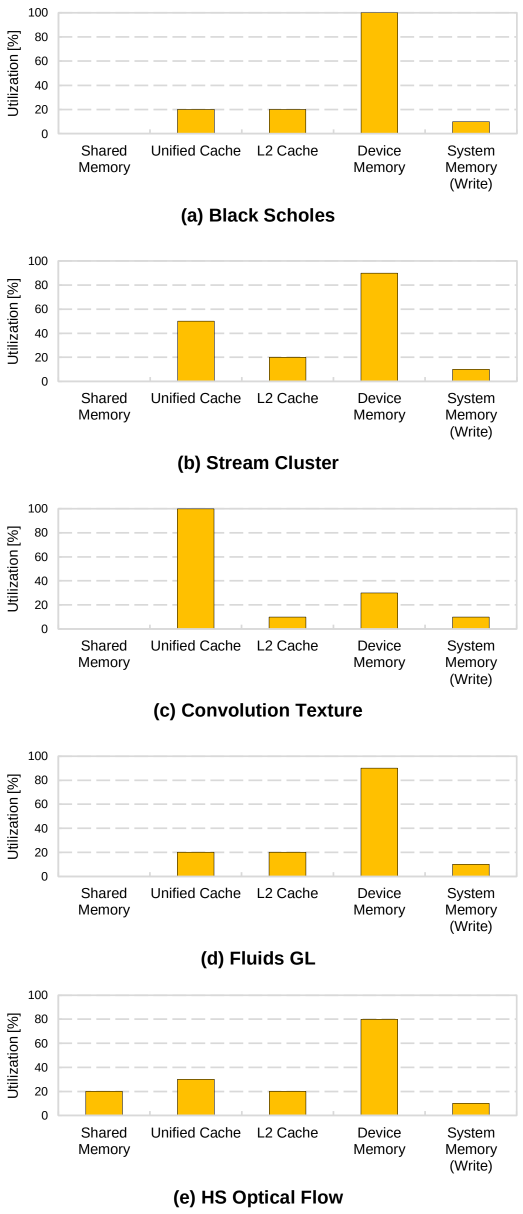

FIGURE 7. **Utilization of each memory resource.**

Figure 6 shows the utilizations of computing resources and memory resources for the memory-bound workloads we investigated. Black Scholes, Stream Cluster, Convolution Texture, Fluids GL, and HS Optical Flow were classified as memory-bound workloads due to their high memory usage. Figure 7 shows the utilizations of each memory resource. As shown in the figure, in all workloads except for Convolution Texture, Device Memory, one of off-chip memories, shows the highest utilization. In the case of Convolution Texture, Unified Cache shows the highest utilization of almost 100%. This is because Convolution Texture performs frequent read accesses to Texture Memory.

Through the analysis in this section, we can summarize that different types of memory resources can be a performance bottleneck even for the same memory-bound workloads.

### *C. DEPENDENCY-LATENCY-BOUND WORKLOADS*

We classify a workload as dependency-latency-bound when both computing and memory resources are under-utilized due to a dependency problem in the execution. This implies that resources are not fully utilized because the dependencies of the workload execution cause a certain stall. Causes of this stall include instruction fetch delay, pipeline busy, synchronization delay, memory dependency, execution dependency, constant miss, texture overhead, and memory throttling.

Figure 8 shows the utilizations of computing and memory resources for workloads classified as dependency-latency-bound. As shown in the figure, neither computing nor memory resources exhibit high utilization in these workloads. Figure 9 analyzes the reasons for stalls in the dependency-latency-bound workloads. The meaning of the stall reasons in the figure can be summarized as follows.

- Instruction fetch delay – The next assembly instruction has not yet been fetched.

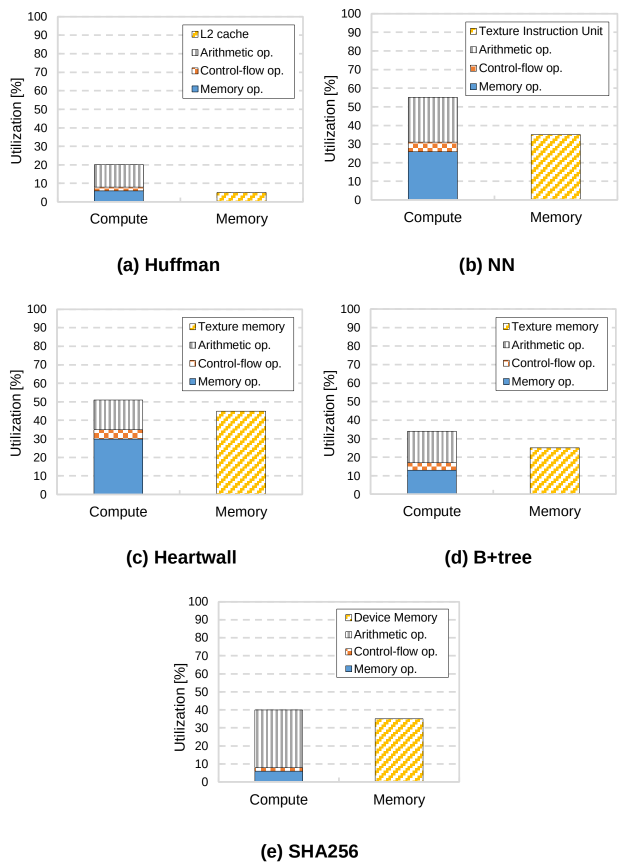

FIGURE 8. **Resource utilization of dependency-latency-bound workloads**

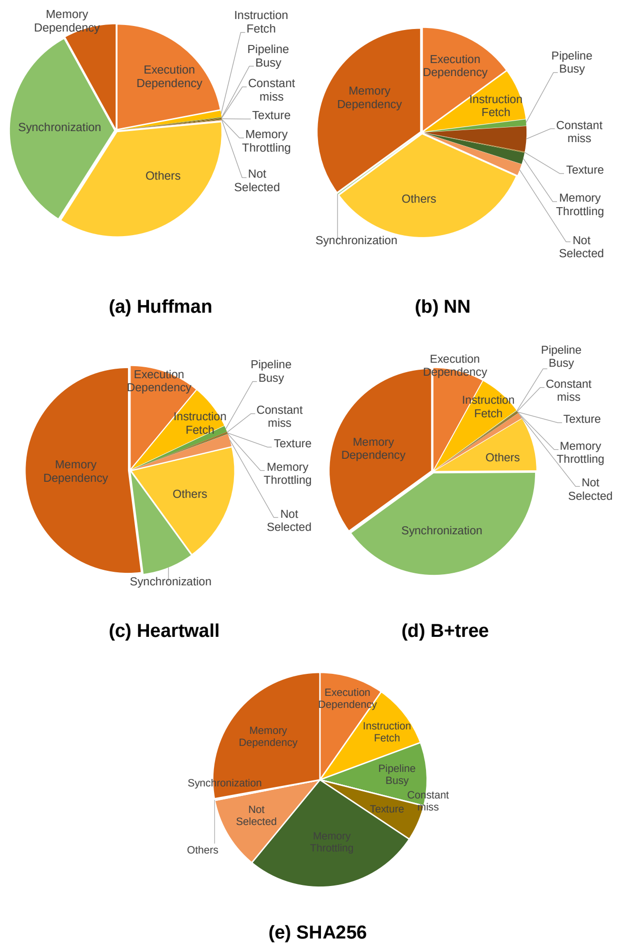

FIGURE 9. **Stall reasons of dependency-latency-bound workloads.**

- Pipeline busy – The computing resource required for the instruction is not yet available.
- Synchronization delay – Execution is blocked at a thread synchronization call.
- Texture overload – The texture subsystem is fully utilized or it has too many requests.
- Memory dependency – A load/store cannot be performed as the target data is not available yet.
- Execution dependency – The input required for the instruction is not yet available.
- Memory throttling – A large number of pending memory operations prevent further forward progress.
- Constant miss – A constant load is blocked due to a miss in the constant cache.
- Not selected – The workload is ready to issue, but another workload has been issued.

As shown in Figure 9, there are different reasons of stalls for each workload. In some cases, such stalls may not be resolved by efficient management of GPGPU resources alone, but improvements can be expected when application developers configure their workloads to improve resource utilizations.

## IV. GPGPU WORKLOAD ALLOCATION BASED ON FINE-GRAINED RESOURCE UTILIZATION

In this section, we present a thread block scheduling policy for GPGPU based on the fine-grained resource utilization analyzed in the previous section. To allocate thread blocks to SMs, GPGPU typically makes use of the Round-Robin scheduling policy that sequentially allocates thread blocks to each SM [22]. Figure 10 shows the basic role of the thread block scheduler, which allocates the next thread block in the queue based on the Round-Robin policy.

As Round-Robin scheduling is simple, easy to implement, and starvation-free, it is efficient to implement in hardware logic [23]. However, Round-Robin scheduling does not consider the resource utilization of workloads. For example, if all the threads allocated are double-precision arithmetic operations, the FP64 units may be overloaded in some SMs,

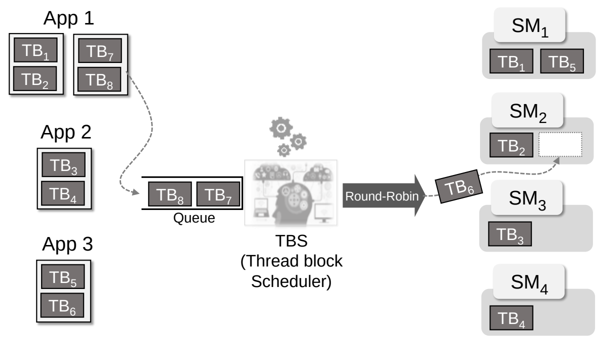

FIGURE 10. **An example of Round-Robin scheduling that allocates Thread Blocks (TB) to SM.**

whereas the same resources may be under-utilized in other SMs, where threads do not have these types of operations.

Similar to Round-Robin, our scheduling policy deletes a pending thread block from the queue, and inserts to an SM, which is selected by the sequential order. Before inserting it, however, our policy checks whether the utilization of a certain resource in the SM exceeds its capacity in case the pending thread block is allocated to it. If so, our policy skips the current SM and moves to the next one. Previous studies also tried to consider resource utilizations by co-locating thread blocks from different workloads within the same SM if their resource usage classification is different [9, 10]. We call this scheme Multi-kernel placement throughout this paper. Although the basic philosophies of Multi-kernel and our policy are similar, our policy considers fine-grained resource utilization whereas Multi-kernel determines whether thread blocks from different applications can be co-located within an SM based on coarse-grained classification (i.e., computing-bound or memory-bound).

From now on, we will explain the exact workings of the proposed policy in comparison with Multi-kernel and Round-Robin with an example. Suppose a GPGPU device that has four SMs and three pending thread blocks in the queue. Tables II and III list the current resource utilizations of SMs and the resource demand of pending thread blocks, respectively. In this simple example, we only consider two computing resources, i.e., single and double precisions, and two memory resources, i.e., device memory and shared memory. The coarse-grained utilization columns in the tables are calculated by the maximum value of the fine-grained utilizations within the same resource classifications.

Now, let us see the scheduling results. In our policy, TB1 is allocated to SM1 as it does not incur the overflow of any fine-grained resource utilizations. Note that the fine-grained utilizations of SM1 are updated from (90, 10, 0, 10) to (100, 100, 0, 20) after TB1 is scheduled. Similarly, TB2 and TB3 are allocated to SM2 and SM3, respectively. In the Multi-

TABLE II
CURRENT RESOURCE UTILIZATIONS OF SMS

| | Fine-grained utilization | | | | Coarse-grained utilization | |
|---|---|---|---|---|---|---|
| | Comp. (Single) | Comp. (Double) | Mem. (Device) | Mem. (Shared) | Comp. | Mem. |
| SM1 | 90 | 10 | 0 | 10 | 90 | 10 |
| SM2 | 50 | 80 | 0 | 10 | 80 | 10 |
| SM3 | 10 | 10 | 0 | 10 | 10 | 10 |
| SM4 | 10 | 80 | 0 | 10 | 80 | 10 |

TABLE III
RESOURCE DEMAND OF PENDING THREAD BLOCKS

| | Fine-grained utilization | | | | Coarse-grained utilization | |
|---|---|---|---|---|---|---|
| | Comp. (Single) | Comp. (Double) | Mem. (Device) | Mem. (Shared) | Comp. | Mem. |
| TB1 | 10 | 90 | 0 | 10 | 90 | 10 |
| TB2 | 10 | 20 | 80 | 10 | 20 | 80 |
| TB3 | 80 | 0 | 10 | 10 | 80 | 10 |

kernel scheme, the scheduling is performed by comparing the coarse-grained utilization of the current SM and that of the pending thread block. Thus, TB1 cannot be allocated to either SM1 or SM2, as it incurs the overflow of computing resources. After proceeding to the next SM, Multi-kernel finally allocates TB1 to SM3. Similarly, TB2 is allocated to SM4. However, TB3 cannot be allocated to any SM even after scanning all SMs from SM1 to SM4 due to the overflow of computing resources. In this case, TB3 should wait in the queue until any SM becomes available. In the Round-Robin scheme, TB1, TB2, and TB3 are allocated to SM1, SM2, and SM3, respectively. Note that Round-Robin allocates thread blocks to SMs one by one without considering resource utilization although it becomes over 100% as long as thread blocks assigned to an SM does not exceed hardware limitations.

As the purpose of our profiling is to extract the characteristics of workloads in advance, we do not perform online profiling at scheduling time. For workloads that have already been executed before, our policy retains previously profiled results and uses them while scheduling the workloads. If there is no profiling history for the workload (i.e. first run), a shadow virtual machine (VM) can perform the profiling separately from the VM that is actually running the workload. Note that a shadow VM is used for the profiling purpose in our cloud GPGPU. Our empirical studies showed that the profiling process requires only 1 to 30 milliseconds for extracting the characteristics of workloads we experimented, which is quite shorter than executing actual workloads in GPGPU. Thus, although our scheduling requires prior knowledge of

TABLE IV
EXPERIMENTAL SCENARIOS IN THE EXPERIMENTS.

| Scenario | VM | Workload | Classification |
|---|---|---|---|
| Scenario 1 | VM–1 | Nbody | Computing-bound |
| | VM–2 | Srad | Computing-bound |
| Scenario 2 | VM–3 | Lava MD | Computing-bound |
| | VM–4 | Nbody | Computing-bound |
| | VM–5 | Back Propagation | Computing-bound |
| Scenario 3 | VM–6 | Black Scholes | Memory-bound |
| | VM–7 | Stream Cluster | Memory-bound |
| | VM–8 | Convolution Texture | Memory-bound |
| Scenario 4 | VM–9 | Nbody | Computing-bound |
| | VM–10 | Hotspot | Computing-bound |
| | VM–11 | LavaMD | Computing-bound |
| | VM–12 | Srad | Computing-bound |
| Scenario 5 | VM–13 | Stream Cluster | Memory-bound |
| | VM–14 | Convolution Texture | Memory-bound |
| | VM–15 | Fluids GL | Memory-bound |
| | VM–16 | HS Optical Flow | Memory-bound |
| Scenario 6 | VM–17 | Particle Filter | Computing-bound |
| | VM–18 | Hotspot | Computing-bound |
| | VM–19 | Convolution Texture | Memory-bound |
| | VM–20 | Black Scholes | Memory-bound |
| Scenario 7 | VM–21 | Conv2D | Computing-bound |
| | VM–22 | SHA256 | Dependency-latency |
| | VM–23 | Back Propagation | Computing-bound |
| | VM–24 | Srad | Computing-bound |
| | VM–25 | Nbody | Computing-bound |

workloads, it is also possible to utilize profiling results after a short training period during the current run.

Separation of profiling and scheduling has the advantage of reducing the run-time complexity of previous studies that relied on online profiling. That is, previous studies mostly arrange and re-arrange workloads to SMs periodically based on performance metrics such as throughput and execution time rather than utilizing the prior knowledge of resource usage patterns. Thus, in a cloud system with many concurrent workloads, the number of possible combinations becomes excessively large. For example, suppose that there are four workloads A, B, C, and D, without the prior knowledge of workload characteristics. Then, all possible combinations, i.e., AB, AC, AD, BC, BD, CD, ABC, ABD, ACD, BCD, and ABCD may be tried to find an efficient co-location of workloads on the same SM. In contrast, as we know the fine-grained resource usage patterns of workloads A, B, C, and D, we can find an efficient schedule without checking various combinations. The only run-time overhead in our policy is to identify the resource utilizations of each SM. Actually, this does not incur significant overhead as the thread block scheduler can keep track of the resource utilizations of each SM by aggregating the resource demand of thread blocks whenever a thread block is newly allocated or completed.

To assess the effectiveness of our thread block scheduling policy, we perform simulation experiments under 7 scenarios consisting of 25 virtual machines. Tables IV and V list the workload situations of each scenario we experimented and the system and resource characteristics of our experiments.

TABLE V
SYSTEM AND RESOURCE CHARACTERISTICS OF OUR EXPERIMENTS.

| Configuration | Value |
|---|---|
| # of host CPU cores | 20 |
| Host memory capacity | 64GB |
| # of GPGPU devices | 2 |
| Emulated GPGPU model | Nvidia Tesla V100 |
| # of SMs | 80 |
| GPGPU memory capacity | 16GB |
| Shared memory per SM | 96KB |
| Register file size per SM | 256KB |

Scenario 1 consists of two virtual machines, VM-1 and VM-2, which execute two computing-bound workloads, Nbody and Srad, respectively. Scenario 2 consists of three virtual machines, VM-3, VM-4, and VM-5, which execute three computing-bound workloads, Lava MD, Nbody, and Back Propagation, respectively. Scenario 3 consists of three virtual machines, VM-6, VM-7, and VM-8, which execute three memory-bound workloads, Black Scholes, Stream Cluster, and Convolution Texture, respectively. Scenario 4 consists of four virtual machines, VM-9, VM-10, VM-11, and VM-12, which perform four computing-bound workloads, Nbody, Hotspot, LavaMD, and Srad, respectively. Scenario 5 consists of four virtual machines, VM-13, VM-14, VM-15, and VM-16, which execute four memory-bound workloads, Stream Cluster, Convolution Texture, Fluids GL, and HS Optical Flow, respectively. Scenario 6 consists of four virtual machines, of which VM-17 and VM-18 execute computing-bound workloads, Particle Filter and Hotspot, respectively, whereas VM-19 and VM-20 execute memory-bound workloads, Convolution Texture and Black Scholes, respectively. Finally, Scenario 7 consists of five virtual machines, of which VM-21, VM-23, VM-24, and VM-25 execute computing-bound workloads, Conv2D, Back Propagation, Srad, and Nbody, respectively, whereas VM-22 executes a dependency-latency-bound workload, SHA256. We assume that each virtual machine executes the given workload repeatedly to see the effect of simultaneous execution of heterogeous workloads in cloud systems.

Under these scenarios, we conduct simulation experiments with the three scheduling policies, Round-Robin (RR), Multi-kernel, and the proposed policy that we call FRU-RR (Fine-grained Resource Utilization aware Round-Robin). The utilization metrics that we use for FRU-RR are the resource utilizations of each computing unit (i.e., single-precision, double-precision, control flow, load-store, and special function) and each memory unit (i.e., shared memory, unified cache, L2 cache, device memory, and system memory). For performance metric, we use the throughput of each virtual machine and the host machine. The throughput of a virtual machine indicates the number of the workload's threads completed per unit time, and we show the throughput of each scheduling policy normalized to that of the Round-Robin. The

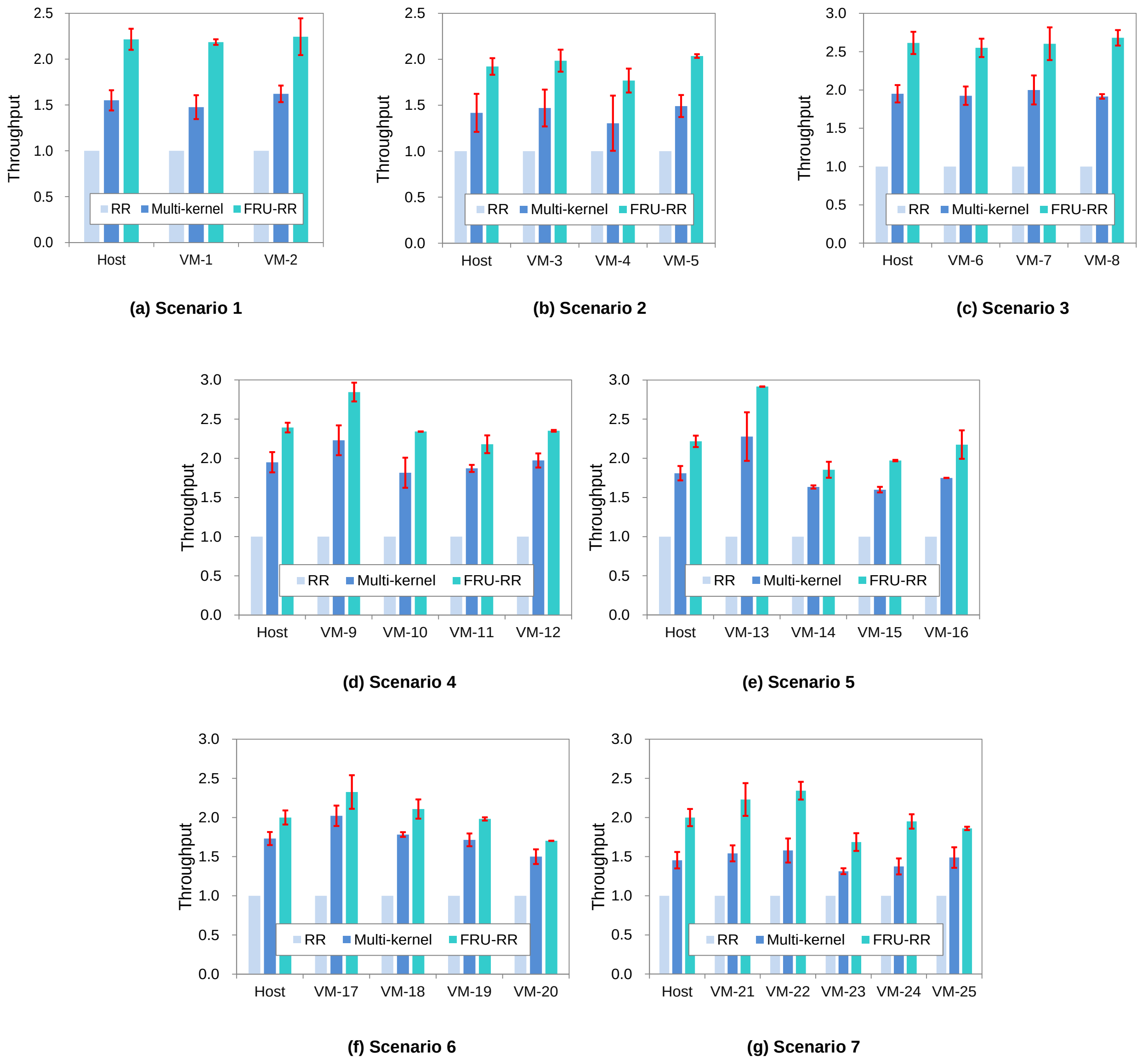

FIGURE 11. **Comparisons of the Round Robin, Multi-kernel, and the proposed policy for various virtual machine scenarios.**

throughput of a host machine indicates the total number of threads completed per time unit including the threads of all virtual machines and the host itself. In case of our policy, the profiling overhead is reflected in the host throughput, implying that the threads executed for profiling are excluded from the host throughput after counting all threads that have completed.

Figure 11 shows the throughput of the proposed policy, FRU-RR, in comparison with Round-Robin and Multi-kernel. We experimented each scenario ten times and plotted the average and their standard deviations. Let us first discuss the performance comparison of our policy with Round-Robin. As shown in the figure, the proposed policy performs significantly better than Round-Robin in all scenarios and workloads. Specifically, the performance improvement is 119.5% on average and up to 191.7%. The main reason of Round-Robin's low performance is that it fails to balance the load between SMs. Although Round-Robin pursues load balancing by distributing thread blocks to each SM one by one, the result did not come out as intended because it does not consider the characteristics of workloads. This implies that Round-Robin, which is currently adopted as the thread block scheduler of most GPGPU devices, can be improved significantly.

Now, let us discuss the performance of our policy in comparison with Multi-kernel. As shown in Figure 11, Multi-kernel also yields better performance than Round-Robin, but the performance improvement of our policy against Multi-kernel is 30.1% on average and up to 42.9%. This is because our policy accurately analyzes the fine-grained bottleneck resource for each application and allocates workloads that do not have the same bottleneck resource on the same SM, whereas Multi-kernel does this based-on the coarse-grained classification. The largest improvement is observed in Scenario 1, where the improvement of our policy is 42.9%. This is because the two workloads in Scenario 1 are all computing-intensive, and thus Multi-kernel cannot co-locate them within the same SM. However, the bottleneck resources for the two virtual machines are different. Specifically, VM-1 uses the FP32 unit but does not use the FP64 unit at all, whereas VM-2 mostly uses the FP64 unit, and thus our policy can place the two workloads together within the same SM without conflicting situations, which is not possible in Multi-kernel.

In Scenarios 2 and 3, the system adopting our policy performs 35.7% and 34.1% better than Multi-kernel, respectively. Similar to Scenario 1, all virtual machines in Scenarios 2 and 3 have homogeneous workloads. That is, workloads in Scenario 2 are all computing-bound, whereas Scenario 3 has all memory-bound workloads. Thus, Multi-kernel cannot co-locate any two workloads within the same SM, whereas our policy can do so in case the resource that causes bottleneck is different. When we compare the results with Scenario 1, the performance gap between Multi-kernel and our policy becomes relatively small in Scenarios 2 and 3. This is due to the characteristics of virtual machines in these scenarios. In particular, Scenario 1 has only two virtual machines, VM-1 and VM-2, which can be placed on the same SM in our policy, but Scenarios 2 and 3 have three virtual machines, two of which have the same bottleneck resources, and thus chances for placing different VMs on the same SM have been reduced.

Now, let us see the results for Scenarios 4 and 5, in which the number of virtual machines has been increased to four. As shown in Figures 11(d) and 11(e), the performance improvement of our policy against Multi-kernel is 22.7% and 22.6%, respectively, for Scenarios 4 and 5. Similar to previous scenarios, these two scenarios consist of homogeneous workloads, and thus Multi-kernel cannot co-locate any two workloads within the same SM. In this case, some resource types in the SM are certain to be under-utilized. For example, if one workload performs single-precision arithmetic while the other performs double-precision arithmetic, allocating a single workload to an SM cannot utilize both types of resources. Unlike Multi-kernel, however, our policy puts these two computing-bound workloads together on the same SM as they do not have conflicting resource usage patterns. Thus, both single- and double-precision arithmetic units can be fully utilized. This will eventually improve the throughput of the workloads as well as the host machine.

Scenario 4 consists of 4 computing-bound workloads, of which VM-9 mainly performs single-precision arithmetic, whereas VM-10, VM-11, and VM-12 perform double-precision arithmetic operations. Thus, our policy can locate VM-9 to any SM to execute single-precision arithmetic without conflicting situations as all the other VMs have different bottleneck resources of double-precision arithmetic. However, VM-10 to VM-12 have the same bottleneck resource, and thus they cannot be co-located within the same SM in our policy. Thus, the performance improvement is not large compared to previous scenarios. Note that similar results can be observed for Scenario 5, where computing-bound workloads are replaced by memory-bound workloads.

Now, let us discuss the performance of our policy in comparison with Multi-kernel in Scenario 6. This scenario consists of four virtual machines, of which VM-17 and VM-18 are computing-bound, whereas VM-19 and VM-20 are memory-bound. Thus, Multi-kernel can co-locate workloads with different classifications on the same SM in this scenario. However, our policy further increases the possibility of co-locating VMs as the two memory-bound workloads, VM-19 and VM-20, have different bottleneck resources. Although the performance gap is not wide, our policy performs better than Multi-kernel by 15.5% on average.

Finally, let us see the results of Scenario 7. This scenario consists of 5 virtual machines, of which VM-21, VM-23, VM-24, and VM-25 perform computing-bound workloads, whereas VM-22 performs a dependency-latency-bound workload. Multi-kernel can co-locate VM-22 with other VMs as they have different workload classifications. However, balancing resource utilization is difficult in Multi-kernel as a majority of workloads have the same classifications of computing-bound. Our policy has further chances to balance resource utilizations since fine-grained resource usage of the computing-bound workloads is different. Specifically, most operations in VM-21 and VM-25 are single-precision arithmetic, whereas those in VM-23 and VM-24 are double-precision arithmetic. Due to this reason, the performance improvement of our policy against Multi-kernel in this scenario is as large as 37.5%.

## V. RELATED WORKS

In the early days of GPGPU multitasking, a single SM could not accommodate multiple workloads, and thus the main issue with multitasking was determining the number of SMs assigned to each workload. In these environments, Kayıran et al. [24] observed that GPGPU performance drops sharply when some types of memory-bound workloads occupy more than a certain number of SMs. They found that such configurations significantly reduce the number of active instructions due to the stalls caused by global memory, which is shared between SMs. Based on this, they proposed DYNCTA that limits the number of SMs allocated to workloads using global memory, not to cause performance degradations. By doing so, computing-bound workloads have relatively high priorities, and this incurs the fairness problem as memory-bound workloads have performance penalties.

TABLE VI
A SUMMARY OF GPGPU MULTITASKING SCHEMES.

| | System or algorithm | Multi-tasking granularity | Profiling metric | Resource Granularity | Profiling method |
|---|---|---|---|---|---|
| Kayıran et al. [24] | DYNCTA | Inter-SM level | Idle cycles, memory stalls | Computing / Memory | Online profiling with workload run |
| Lee et al. [12] | LCS, mCKE | Intra-SM level | # of instructions executed | Computing / Memory | Online profiling with first TB run |
| Xu et al. [25] | Warped-Slicer | Intra-SM level | IPC (Instructions per Cycle) | Computing / Memory | Short online profiling runs |
| Wang et al. [9] | SMK | Intra-SM level | WIPC (Warp Instructions per Cycle) | Computing / Memory | Online profiling with dedicated SMs |
| Park et al. [10] | Maestro | Intra-SM level | Performance counter | Computing / Memory | Dynamic profiling for different partitions with performance counters |
| Allen et al. [5] | Slate | Intra-SM level | FLOPS, memory bandwidth | Computing / Memory | Online profiling at the first run |
| Proposed scheme | FRU-RR | Intra-SM level | Resource utilizations | Fine-grained unit | Separate profiling |

Lee et al. [12] showed that GPGPU performance is gradually improved as the number of thread blocks allocated to an SM increases, but the performance drops sharply when it exceeds a certain resource limit. To address this problem, they determine the maximum number of thread blocks allocated per SM based on the profiling of the first thread block's execution. They call this scheme LCS (Lazy Cooperative-thread-arrays Scheduling). After allocating by LCS, they observed some idle resources in the SM, and they additionally proposed mCKE (mixed Concurrent Kernel Execution) to allocate other workloads with different resource usage characteristics on the same SM.

Studies on GPGPU multitasking increasingly partitions a single SM across multiple workloads. Xu et al. [25] explored various intra-SM slicing strategies, and showed that there is not one intra-SM slicing strategy that derives the best performance for all application pairs. Based on this observation, they proposed Warped-Slicer, a dynamic intra-SM slicing strategy that determines how each workload's performance is varied as more thread blocks are assigned to an SM based on an analytical model. They showed that their analytical model can be applied to each workload by performing short on-line profiling runs.

SMK [9] co-executes workloads with compensating resource usage in the same SM to achieve high utilization and efficiency. Specifically, SMK pairs a memory-intensive workload with a computing-intensive workload in the same SM to greatly improve utilization of both memory and computing resources. SMK also takes into account the fairness among different workloads while dispatching thread blocks.

Park et al. [10] proposed GPGPU Maestro, which runs multiple workloads on the same SM like SMK, but dynamically manages resource partitioning on GPGPU to maximize the system performance. Specifically, they showed that multitasking performance varies heavily because of dynamism within a workload and interference between workloads. To cope with this situation, Maestro monitors the performance counters for different allocations with a subset of SMs, and discovers the best performing resource partition exploiting both spatial multitasking and SMK.

Allen et al. [5] presented Slate, a software-based workload-aware GPGPU multitasking framework. Similar to Maestro, Slate selects concurrent workloads that have complementary resource demands at run-time to minimize interference for individual workloads and improve resource utilization. Unlike Maestro, however, Slate classifies computing and memory resource usage into three stages: low, medium, and high, and categorizes each workload through online or offline profiling. Based on this, Slate determines whether two different workloads can be performed on the same SM according to the resource characteristics of them.

Table VI briefly compares the aforementioned schemes with respect to multi-tasking granularities, profiling metrics, resource granularities, and profiling methods.

## VI. CONCLUSIONS

With the widespread adoption of multitasking GPGPU in cloud systems, maximizing the resource utilization by judicious allocation of heterogeneous workloads in GPGPU has become an important issue. In this article, we investigated the resource utilization characteristics of 17 popular GPGPU workloads, and classified them into computing-bound, memory-bound, and dependency-latency-bound, and then refined the classifications based on the detailed resource that causes the performance bottleneck. As the bottleneck resource may be different even for workloads with the same classifications, we presented a thread block scheduling policy that considers the fine-grained resource utilization for cloud systems. Unlike previous approaches, we separated profiling from scheduling, thereby reducing the run-time complexity of scheduling. By co-locating workloads with complementary resource demands on the same SM, our policy aims to maximize the overall resource utilizations in the GPGPU device. Experimental results with various virtual machine scenarios showed that our thread block scheduling policy improves the GPGPU throughput of cloud systems by 119.5% on average and up to 191.7%.

## REFERENCES

[1] A. Fonseca and B. Cabral, “Prototyping a GPGPU neural network for deep-learning big data analysis,” Big Data Research, vol. 8, pp. 50-56, 2017.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng “Tensorflow: a system for large-scale machine learning,” Proceedings of USENX OSDI Conferences, pp. 265-283, 2016.

[3] D. Shin, K. Cho, and H. Bahn, “File type and access pattern aware buffer cache management for rendering systems,” Electronics, vol. 9, no. 1, article 164, 2020.

[4] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. Keckler, M. Kandemir, and C. Das “Anatomy of GPU memory system for multi-application execution,” Proceedings of the ACM Symposium on Memory Systems, pp 223-234, 2015.

[5] T. Allen, X. Feng, and R. Ge, “Slate: enabling workload-aware efficient multiprocessing for modern GPGPUs,” Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 252-261, 2019.

[6] S. Pai, M.J. Thazhuthaveetil, and R. Govindarajan, “Improving GPGPU concurrency with elastic kernels,” Proceedings of the ACM SIGARCH Computer Architecture News, vol. 41, pp. 407-418, 2013.

[7] H. Bahn and J. Kim, “Separation of virtual machine I/O in cloud systems,” IEEE Access, vol. 8, pp. 223756-223764, 2020.

[8] E. Lee, H. Bahn, M. Jeong, S. Kim, J. Yeon, S. Yoo, S. Noh, and K. Shin, “Reducing journaling harm on virtualized I/O systems,” Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR), pp. 1-6, 2016.

[9] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, “Simultaneous multikernel GPU: multi-tasking throughput processors via fine-grained sharing,” Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA), pp.358-369, 2016.

[10] J. Park, Y. Park, and S. Mahlke, “Dynamic resource management for efficient utilization of multitasking GPUs,” Proceedings of the 21nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 527-540, 2017.

[11] O. Maitre, N. Lachiche, P. Clauss, L. Baumes, A. Corma, and P. Collet, “Efficient parallel implementation of evolutionary algorithms on GPGPU cards,” Proceedings of the European Conference on Parallel Processing, Springer, Berlin, Heidelberg, 2009.

[12] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, “Improving GPGPU resource utilization through alternative thread block scheduling,” Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 260-271, 2014.

[13] M. Huzaifa, J. Alsop, A. Mahmoud, G. Salvador, M. D. Sinclair, and S. V. Adve, “Inter-kernel reuse-aware thread block scheduling,” ACM Transactions on Architecture and Code Optimization, vol. 17, no. 3, article 24, pp. 1-27, 2020.

[14] G. Gilman, S. S. Ogden, T. Guo, and R. J. Walls, “Demystifying the placement policies of the NVIDIA GPU thread block scheduler for concurrent kernels,” ACM SIGMETRICS Performance Evaluation Review, vol. 48, no. 3, pp. 81-88, 2020.

[15] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, “Many-thread aware prefetching mechanisms for GPGPU applications,” Proceedings of the IEEE/ACM International Symposium on Microarchitecture, 2010.

[16] K. Cho and H. Bahn, “Performance analysis of thread block schedulers in GPGPU and its implications,” Applied Sciences, vol. 10, no. 24, article 9121, 2020.

[17] C. Su, P. Chen, C. Lan, L. Huang, and K. Wu, “Overview and comparison of OpenCL and CUDA technology for GPGPU,” IEEE Asia Pacific Conference on Circuits and Systems, 2012.

[18] G. Gasior, “Nvidia’s Volta GPU to feature on-chip DRAM,” Techical Report. Available at https://techreport.com/news/24529/nvidias-volta-gpu-to-feature-on-chip-dram/.

[19] NVIDIA CUDA, CUDA C Programming Guide, 2018. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[20] GPU Computing SDK, http://developer.nvidia.com/gpu-computing-sdk.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” Proc. IEEE Symp. Workload Characterization, pp. 44-54, 2009.

[22] Q. Wang and C. Xiaowen, “GPGPU performance estimation with core and memory frequency scaling,” IEEE Transactions on Parallel and Distributed Systems, vol. 31, pp. 2865-2881, 2020.

[23] J. Nieh, C. Vaill, and H. Zhong, “Virtual-time Round-Robin: an O(1) proportional share scheduler,” Proceedings of the 2001 USENIX AnnualTechnical Conference, 2001.

[24] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, “Neither more nor less: Optimizing thread-level parallelism for GPGPUs,” Proceedings of 22nd IEEE International Conference on Parallel Architectures and Compilation Techniques, pp. 157-166, 2013.

[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,” Proceedings of 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA), pp. 230-242, 2016.

**KYUNGWOON CHO** received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Seoul National University, in 1995, 1997, and 2012, respectively. He is currently a senior researcher at the Embedded Software Research Center, Ewha University, Seoul, Republic of Korea. Before joining Ewha, he was a chief officer in the Clunix R&D Center, Seoul, Republic of Korea. His research interests include multimedia systems, cloud computing, real-time systems, embedded systems, and operating systems.

**HYOKYUNG BAHN** (M’02) received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Seoul National University, in 1997, 1999, and 2002, respectively. He is currently a full professor of computer science and engineering at Ewha University, Seoul, Republic of Korea. His research interests include operating systems, caching algorithms, storage systems, embedded systems, system optimizations, and real-time systems. Prof. Bahn has published more than 100 papers in leading conferences and journals in these fields, including USENIX FAST, IEEE Transactions on Computers, IEEE Transactions on Knowledge and Data Engineering, and ACM Transactions on Storage. He also received the Best Paper Awards at the USENIX Conference on File and Storage Technologies in 2013.