symmetry

MDPI

Article

GPGPU Task Scheduling Technique for Reducing the Performance Deviation of Multiple GPGPU Tasks in RPC-Based GPU Virtualization Environments

Jihun Kang and Heonchang Yu *

Department of Computer Science and Engineering, Korea University, Seoul 02841, Korea; k2j23h@korea.ac.kr
* Correspondence: yuhc@korea.ac.kr

Abstract: In remote procedure call (RPC)-based graphic processing unit (GPU) virtualization environments, GPU tasks requested by multiple-user virtual machines (VMs) are delivered to the VM owning the GPU and are processed in a multi-process form. However, because the thread executing the computing on general GPUs cannot arbitrarily stop the task or trigger context switching, GPU monopoly may be prolonged owing to a long-running general-purpose computing on graphics processing unit (GPGPU) task. Furthermore, when scheduling tasks on the GPU, the time for which each user VM uses the GPU is not considered. Thus, in cloud environments that must provide fair use of computing resources, equal use of GPUs between each user VM cannot be guaranteed. We propose a GPGPU task scheduling scheme based on thread division processing that supports GPU use evenly by multiple VMs that process GPGPU tasks in an RPC-based GPU virtualization environment. Our method divides the threads of the GPGPU task into several groups and controls the execution time of each thread group to prevent a specific GPGPU task from a long time monopolizing the GPU. The efficiency of the proposed technique is verified through an experiment in an environment where multiple VMs simultaneously perform GPGPU tasks.

Keywords: HPC cloud; GPGPU computing; GPU virtualization; GPU sharing

Citation: Kang, J.; Yu, H. GPGPU Task Scheduling Technique for Reducing the Performance Deviation of Multiple GPGPU Tasks in RPC-Based GPU Virtualization Environments. *Symmetry* **2021**, *13*, 508. https://doi.org/10.3390/sym13030508

Academic Editor: José Carlos R. Alcantud

Received: 10 February 2021
Accepted: 18 March 2021
Published: 20 March 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The cloud environment provides multiple virtual machines (VMs) by logically multiplexing the resources of a single server through virtualization technology. The central processing unit (CPU), memory, and storage are computing resources of traditional cloud-based infrastructures. In cloud environments, multiple VMs use the CPU evenly based on CPU usage time, and memory and storage are provided in the form of blocks so that the resources can be used without affecting each other's performance as long as the resources are allowed. Each computing resource is provided fairly in accordance with the user's resource requirements through hardware support, such as VT-x [1], that can efficiently manage resources in a virtualization environment and resource sharing technique, which is the core of virtualization technology, such as CPU usage time-based VM scheduling on Xen [2] and kernel-based virtual machine (KVM) [3].

In the cloud environment, a graphics processing unit (GPU) device is provided to the VM to support high-performance computation along with CPU, memory, and storage. Commercial cloud providers also provide high-performance computing (HPC) cloud services [4–7] that support high-performance computations, and by providing VMs that can access GPUs, VM users can use GPUs to execute high-performance tasks on VMs. Thus, GPU virtualization technology can be used to manage GPU resources while providing GPUs to multiple VMs in a cloud environment and providing HPC services.

General GPUs are not designed for virtualization environments [8]. In the case of cloud-only GPUs [9,10], hardware-side virtualization technology is provided so that they can be shared and used evenly by multiple VMs. However, because general GPUs

are designed for a single user, hardware support technology for resource-sharing is not provided. In the case of cloud-only GPUs, hardware support is only available for specific products. Thus, when using GPUs in a cloud environment, general GPUs cannot be utilized, which poses a problem.

GPU virtualization can improve resource utilization by supporting multiple VMs to share a single GPU and can provide GPUs to multiple VMs running on a physical server. In addition, when multiple VMs share a single GPU, the GPU tasks running on multiple VMs can utilize approximately 100% of the GPU if a task of a size suitable for the available GPU resources is executed, thus minimizing the occurrence of idle resources. However, general-purpose GPU (GPGPU) [11,12] tasks executed on a GPU are scheduled in a non-preemptive manner [13,14]. Therefore, if a thread of a GPGPU task on the GPU runs for a long time, other tasks must wait until the running task is completed. In addition, because GPU tasks are scheduled through a hardware scheduler inside the GPU device, it is impossible to manage the task sequence or GPU usage time from the outside.

In a cloud environment in which multiple VMs share physical resources, the most important aspect when sharing a GPU device with multiple VMs is to prevent performance imbalance by minimizing the performance impact of other VMs when the VM performs tasks. Multiple users of VMs must be supported to use resources for even times without affecting each other under the same conditions to prevent an imbalance between resource use opportunities and performance. However, as described above, GPUs are not designed for resource sharing or cloud environments. In the GPU, when a task starts, it is not possible to arbitrarily adjust the GPU usage time from the outside or perform an appropriate context switching between GPGPU tasks based on the GPU usage time so that the GPU can be used evenly between VMs. The hypervisor cannot adjust the GPU usage time of multiple VMs use the GPU for an equal amount of time, and commercial virtualization platforms treat the GPU as an I/O device; thus, information on GPU usage time is not managed.

In this paper, we propose a GPGPU scheduling technique to solve the limitations due to the characteristics of existing GPU in a virtualization environment. Our work minimizes the GPU monopoly time of long-running GPGPU tasks in remote procedure call (RPC)-based GPU virtualization environments and GPGPU tasks performed by each VM to be processed fairly. Our proposed scheduling method divides the threads of the GPGPU task running in the VM and determines task priorities according to the GPU usage time and it can minimize the GPU monopoly time for a long-running GPGPU task owing to the characteristics of the GPGPU task and provide fairness for GPU usage between VMs.

The main contributions of our work are:

- In this paper, we propose a GPGPU task scheduling technique that supports VMs to use a GPU evenly in an RPC-based GPU virtualization environment where multiple VMs can share a single GPU.
- The approach of this paper alleviates the problem of long-term GPU occupancy of a specific VM because it divides and processes the threads of GPGPU tasks executed in multiple VMs into several thread groups.
- It is possible to minimize the waiting time for other VMs to wait for GPU use by preventing long-term GPU monopoly of VMs that perform long-running GPGPU tasks.
- The approach of this paper does not require the exchange of information about the GPGPU task between each user VM to schedule the GPGPU task and does not require an additional modification of the user program code. Our work ensures transparency on the system side.
- Because the technique proposed in this paper does not modify system components such as operating system (OS), Hypervisor, and GPU driver, it has little dependence on system components and can be applied relatively easily to other virtualization platforms.

This paper describes the techniques proposed in this paper in Section 2 and background techniques for describing the implementation environment. In Section 3, the motivation of this paper is explained along with performance experiments on the FIFO-based

scheduling environment. Section 4 describes the proposed GPGPU task scheduling technique and describes the GPGPU task information management technique for this in detail. Section 5 describes the performance evaluation of the GPGPU task scheduling technique proposed in this paper. It also verifies the effectiveness of the proposed technique. In addition, Section 6 discusses the limitations of the technique proposed in this paper, Section 7 describes related studies, and Section 8 concludes the paper.

2. Background

2.1. GPGPU Programming Model

The GPGPU programming model is a technology for using the GPU as a general-purpose computing device. In the GPGPU programming model, the GPU is used as the auxiliary computing unit of the CPU, and it cannot start program code or perform tasks alone without the CPU. Because the GPU is a simple co-processor, all instructions start from the CPU, and at the request of the CPU. Only the GPU computation part of the program is processed by the GPU. The GPGPU computing environment is a heterogeneous computing environment and uses both CPU and GPU as computing devices. The GPU is the CPU's co-processor and processes the parallel processing part of the program. In a GPGPU programming model, the GPU computation part is generally called a kernel function, and when the kernel function is called inside the program after performing the overall work of the program through the CPU, the GPU performs the kernel function.

The biggest difference between GPGPU tasks and general CPU tasks is in terms of data management of data used for the task. In general, the data to be used by the CPU is managed in the main memory; however, in the case of the GPU, there is a separate GPU memory to load data to be used for GPU computation. However, unlike the main memory, the GPU memory cannot directly input data, and the CPU cannot directly access the GPU memory. Thus, to upload data to be used for GPU computation to GPU memory, data are uploaded to the main memory and then copied to the GPU memory. In addition, to read the result of a GPGPU task, the task result data must be copied from the GPU memory to the main memory to access the task result data. Figure 1 shows the GPGPU processing flow in the GPGPU programming model.

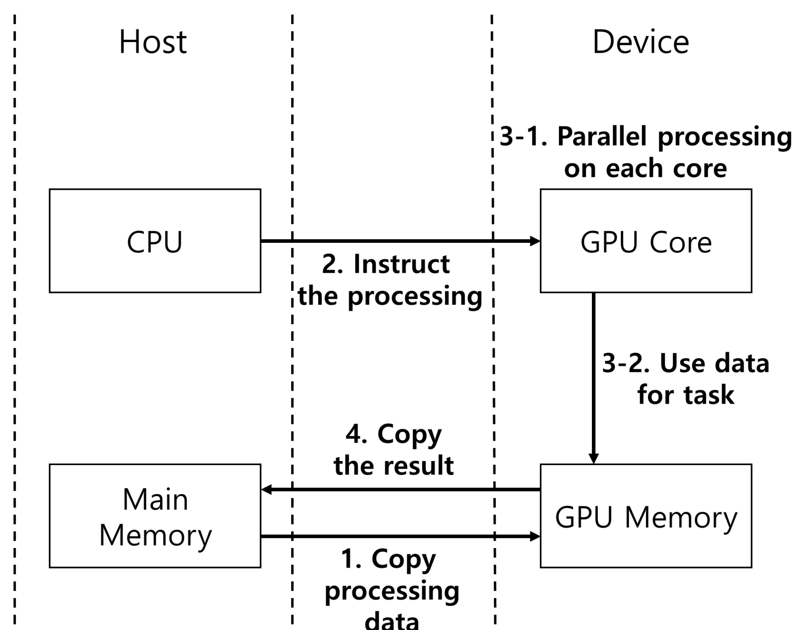


Figure 1. General-purpose computing on graphics processing unit (GPGPU) programming model.

As shown in Figure 1, the GPGPU programming model is classified into host and device areas according to the processing device. In addition, as described above, the data loaded in the GPU memory cannot be directly accessed; thus, data transfer between the

main memory and the GPU memory is indispensable before and after performing the GPU operation. The GPU operation can be performed only when data transfer from the main memory to the GPU memory is completed, and the result of the GPU operation can be checked only when the GPU operation result is copied from the GPU memory to the main memory.

In addition, as described above, the GPU operates without considering the sharing of resources. Kernel functions, that is, GPU operations, are generally processed in a thread group unit [15,16], unlike CPU operations that are generally processed in a time-division method. In the case of CPU, scheduling is performed based on the CPU usage time; thus, multiple processes can evenly share the CPU. In contrast, thousands to tens of thousands of threads performing GPGPU tasks are grouped into a number of groups and processed in a thread group unit. For GPGPU tasks, context switching does not occur in the middle of task execution like in the case of CPU tasks. Except when a high-latency task such as accessing data in GPU memory is executed, threads do not stop in the middle of execution when a task starts executing. This feature causes a performance imbalance problem [17–20] when GPGPU tasks with different execution times are simultaneously hosted. A long-running GPGPU task is more likely to monopolize the GPU for a longer time than a short-running GPGPU task; thus, a long-running GPGPU task causes a delay of other VMs. Conversely, a GPGPU task with a long execution time can hide the delay time to some extent because its execution time is large, even if some waiting time exists because of a GPGPU task with a short execution time. In addition, after the GPGPU task's thread group is executed, the execution order of the thread groups cannot be controlled externally; thus, a limitation exists in ensuring the performance consistency of GPGPU tasks in an environment where GPGPU tasks with different execution times are simultaneously executed.

2.2. GPU Virtualization in Cloud Computing

In a cloud environment, various methods are used to provide a GPU or GPU computation function to a VM. The most basic method of providing a GPU to a VM is a direct pass-through method. Because the direct pass-through method uses hardware support technology, the methods are provided by general commercial virtualization software or open source-based virtualization platforms. Direct pass-through can allocate a GPU to a single VM through the support of hardware technology, and because a VM assigned with a GPU can directly access the GPU, the performance can be similar to that in the native environment. However, in the pass-through method, multiple VMs cannot access the GPU at the same time, and only one VM can access the GPU. In general, because the number of GPUs that can be installed on a single server is limited, the pass-through method has restrictions on the use of GPUs by multiple VMs running on the server.

The main methods can be used to virtualize a GPU and share it with multiple VMs: (1) trap and emulate; (2) full GPU virtualization; (3) API forwarding-based GPU virtualization. The trap and emulate methods are techniques that emulate a GPU device to support performing GPU tasks in an environment without a real GPU. The trap and emulate intercepts when a GPU task is executed in a VM or general computing device and passes it to a fake GPU that emulates how the GPU performs tasks such that even if there is no GPU, a program implemented to run on the GPU can be executed. This method is old technology and performs very poorly because it involves an environment without a GPU.

Full GPU virtualization technology [21–23] multiplexes GPUs by modifying the hypervisor, OS, and GPU device drivers to virtualize GPUs that do not support hardware technology for software-based virtualization. A logically multiplexed GPU can be accessed by multiple VMs, and each VM is allocated resources such as logically divided GPU data channels and GPU memory. Full GPU virtualization technology enables multiple VMs to efficiently share the GPU and supports each VM to use the full GPU functionality. However, as described above, system complexity increases as system components such as hypervisor, OS, and GPU device drivers need to be modified. Therefore, portability is not good when

applied to other virtualization platforms or OSs, and a system for virtualization may need to be newly developed to apply to other virtualization platforms or OSs.

Finally, API forwarding-based GPU virtualization technology, like other GPU virtualization technologies, is used to support GPU operations for virtual machines that are not directly assigned the GPU. Existing API forwarding techniques, such as qCUDA [24], GVirtuS [25], virtio-CL [26] and GViM [27] proposed to support GPGPU tasks for virtual machines use techniques such as modifying GPGPU APIs or re-direction to deliver GPGPU APIs to GPUs and return processing results to virtual machines. In addition to GPGPU APIs, API forwarding-based GPU virtualization technology for virtualizing graphical processing APIs has also been proposed. VMGL [28] is GPU virtualization technologies for graphics processing, support remote rendering to handle openGL-based tasks in virtual machines. In particular, the RPC-based GPU virtualization technology, such as vCUDA [29] and rCUDA [30], used in this paper, one of the API forwarding-based GPU virtualization technology using RPC communications, and the user virtual machine uses a modified GPU API to request GPU operations to the VM or host system that owns GPUs. The user VM has a server–client structure that sends API, API parameters, and data to the GPU owner through internal RPC communication, and the GPU owner processes the requested task and returns the task result to the user VM. When the VM executes the GPU operation API, the modified GPU programming API to transmit the API information and data through internal communication of RPC are executed and delivered to the GPU owner. The GPU owner, who received the GPU programming APIs executed by multiple VMs, executes the GPU task requested by each VM in the form of a multi-process and returns it to each VM. RPC-based GPU virtualization technology does not depend on system components because it does not require any modification of system components such as hypervisor, OS, and GPU device drivers. Because it involves modifying the GPU programming API, it is relatively easy to configure the GPU virtualization environment when commercially available on different virtualization platforms or OSs. In addition, because API information and data are transmitted to the GPU through RPC communication, there is an advantage in that it is possible to easily use the GPU of another server when the server's GPU resources are insufficient. Because the RPC-based GPU virtualization technology delivers API information and data through RPC communication, overhead due to data transmission occurs; However, this overhead can be improved by using a high-performance network interface such as InfiniBand.

2.3. Our RPC-Based GPU Virtualization Environments

In this paper, as described above, to propose and demonstrate the scheduling technique of GPGPU tasks in an RPC-based GPU virtualization environment, a virtualization environment was built using Xen, an open-source-based virtualization platform. In the RPC-based GPU virtualization environment, each user VM is not directly assigned a GPU and uses a modified GPU programming API. When the user VM performs GPU tasks, the modified GPU programming API is executed instead of the native GPU programming API. The modified API delivers API information and data to the GPU owner through an internally defined RPC communication function; it then executes the GPU task and returns the result to the user VM. Figure 2 shows the overall structure of the RPC-based GPU virtualization environment built using Xen [31,32].

As shown in Figure 2, our RPC-based GPU virtualization environment was built using Xen. In Xen, domain 0, a privileged VM, and guest domain, a user VM, exist. Using a separate driver model, domain 0 performs I/O operations of the user VM instead. In the RPC-based GPU virtualization environment used in this paper, a separate user VM assigned a GPU (hereinafter referred to as GPGPUvm) using the direct pass-through technique [33,34] receives the API of the GPU task executed by the user VM and processes it instead. Because domain 0, which is a privileged VM that is not accessible from the user's perspective, exists in Xen, GPUs can be assigned to domain 0; However, because domain 0

performs I/O operations of all user VMs, it is always busy; thus, we use a separate virtual machine for the GPU that the user cannot access.

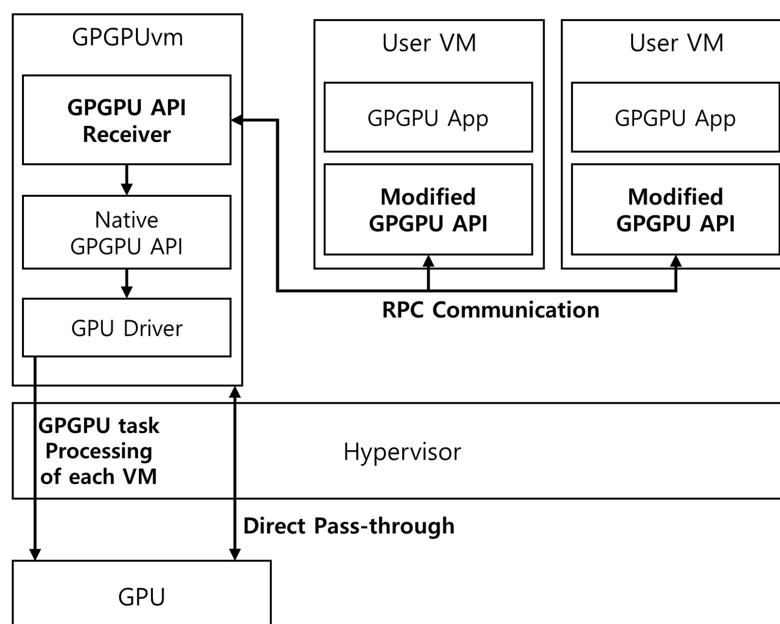


Figure 2. Our remote procedure call (RPC)-based graphic processing unit (GPU) virtualization system.

When each user VM executes the GPGPU tasks, the modified GPU programming API is used, which has a built-in RPC communication function so that API information and data can be transmitted to the VM owning the GPU. In the environment of this paper, VMs execute OpenCL-based GPGPU tasks, which are widely used as GPGPU programming APIs, and the OpenCL API has been modified to support the RPC-based GPU virtualization environment. When each user VM executes the API while executing the GPGPU program, the modified GPGPU API is executed, and the modified GPGPU API delivers API-related information and data to the GPGPUvm. When a VM first requests GPGPU work from GPGPUvm, GPGPUvm creates a thread for each user VM to process the GPGPU work of each user VM and uses the native GPGPU API to process the work on the real GPU.

3. Motivation

In this section, we measure the performance of GPGPU tasks running on multiple VMs in RPC-based GPU virtualization environments and analyze the performance imbalance problem. In the experiment, 6 VMs were used, and the VMs were divided into two groups according to the size of the GPGPU task. Experiments use matrix multiplication implemented with OpenCL. Among the six VMs, three VMs performing 9000×9000 matrix multiplication take a relatively short execution time, while the ones performing $14{,}000 \times 14{,}000$ matrix multiplication required a relatively long execution time. Experiments were performed 10 times in the same environment to check the average performance and performance deviation and analyze the imbalance problem. As described above, in the RPC-based GPU virtualization environment, the data transfer between the GPGPUvm and the user VM occurs. However, in this paper, the scheduling of multiple GPGPU tasks is the main focus; thus, we measured the GPU computation time excluding the data transfer time. Figure 3 shows the experimental results.

As shown in Figure 3, the maximum and minimum values of the execution time of the GPGPU task have very large deviations, and the 9000×9000 matrix multiplication with a relatively short execution time shows a particularly large performance deviation. In Figure 3, VMs 2 and 3 execute a relatively smaller task; however, in some cases, the performance is worse than those of VMs 4, 5, and 6. In addition, the average performance of long-running GPGPU tasks is relatively uniform compared to short-running GPGPU tasks;

however, the average performance of a GPGPU task that is executed for a short time is not constant. In the case of maximum and minimum performance, all VMs have a similar range. However, in the case of a relatively short-running GPGPU task, the performance deviation is larger than that of a relatively long-running GPGPU task, and the performance of each VM is also not constant.

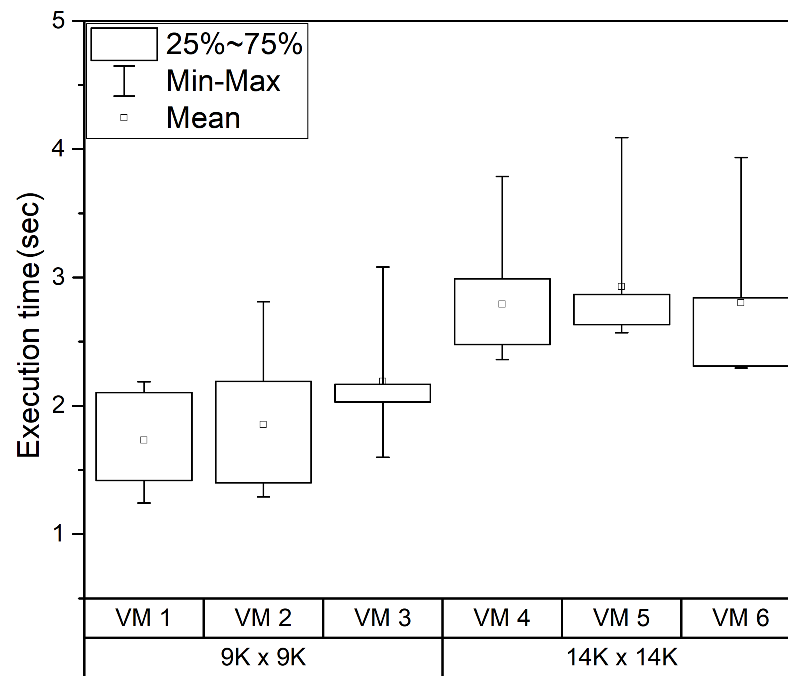
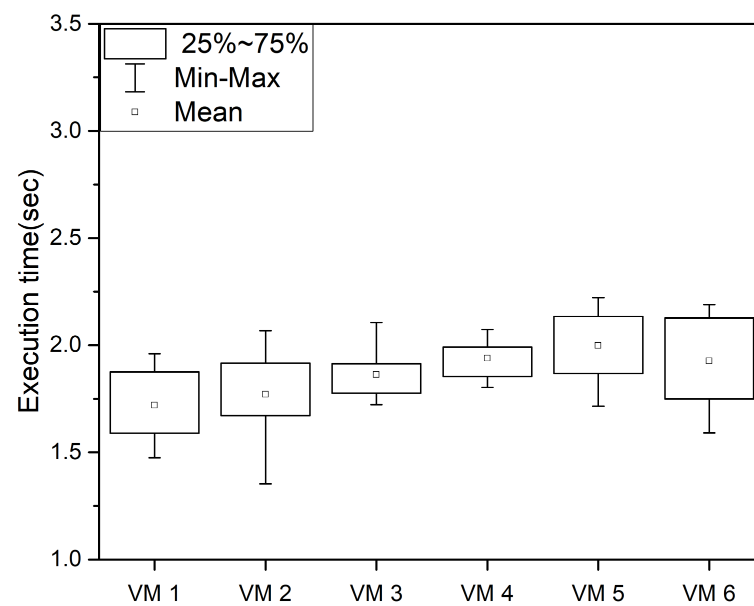
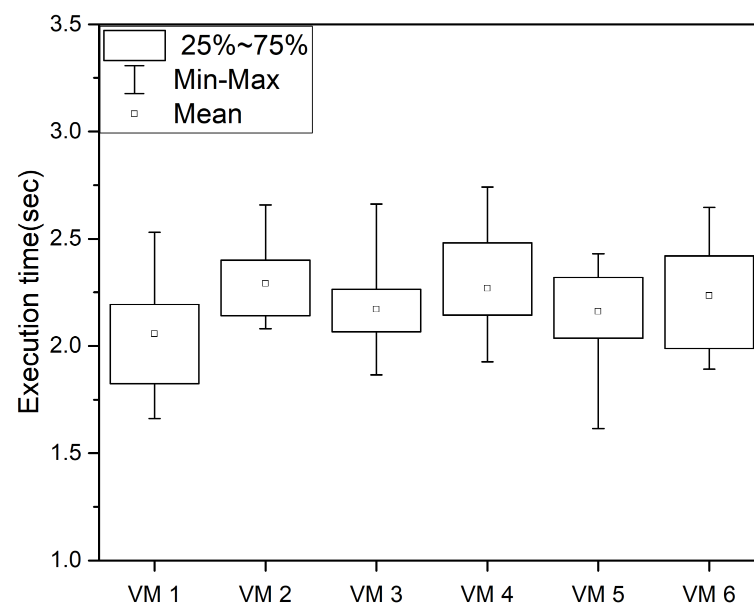


Figure 3. Performance deviation when running long-running GPGPU tasks and short-running GPGPU tasks simultaneously.

The problem of performance variation and performance imbalance of GPGPU tasks is that other thread groups can be executed only after the task of the thread group is completed because of the non-preemptive scheduling of the existing GPU. As described above, the GPU does not provide context switching. When the GPGPU task is executed, the GPGPU tasks of other virtual machines can be executed after the running thread is completed. Thus, the execution time of the GPGPU task, which occupies the GPU, affects the performance of the remaining GPGPU tasks. If short-running GPGPU tasks execute before running a GPGPU task with a long execution time, the short-running GPGPU tasks can quickly complete the task using the GPU earlier. However, if the long-running GPGPU task is executed first, the short-running GPGPU task will have to wait to use GPU until the long-running GPGPU task is complete. The execution order of GPU tasks is determined by the hardware-based scheduler of the GPU. In addition, it is difficult to arbitrarily determine the execution order of GPGPU tasks because the execution time of the task is not known in advance. Thus, VMs that execute GPGPU tasks cannot guarantee consistent performance, and unpredictable performance may occur regardless of the size and scale of the task. Figure 4 shows the performance of VMs running the same GPGPU task. This experiment was performed using six VMs running the same GPGPU task to identify the performance deviation and performance imbalance phenomenon of the GPGPU task and to analyze problems in the FIFO-based scheduling environment. The experiments were performed 10 times.



(**a**) Performance of the 9 k × 9 k matrix multiplication ran on 6 virtual machines.



(**b**) Performance of the 10 k × 10 k matrix multiplication ran on 6 virtual machines.

Figure 4. Performance when the same GPGPU tasks are simultaneously executed in each virtual machine (VM).

Figure 4 shows the performance deviations of running the same GPGPU tasks at the same time. If a GPGPU task with the same run time is executed simultaneously as shown in Figure 4, the performance deviation is smaller than the experimental results shown in Figure 3. The maximum and minimum performance, as well as the 25% to 75% performance deviation. When multiple identical GPGPU tasks run simultaneously, all threads groups run on GPU are performed the same task. The results of Figure 4 show that each GPGPU task achieves a relatively equal performance when all GPGPU tasks executed at the same time are all the same work, rather than when running a somewhat different GPGPU task at the uniformity time.

The focus of this paper is to schedule GPGPU tasks when GPGPU tasks with different execution times are run simultaneously, as with the experimental results of Figure 3. As shown in the experimental results in this section, when different GPGPU tasks are

executed at the same time, GPGPU tasks with shorter run times have been found to have greater performance deviations than GPGPU tasks with relatively long run times. We propose a GPGPU task scheduling technique for an RPC-based GPU virtualized environment based on split processing of threads by referring to the GPGPU working time for each VM to prevent performance deviations and performance imbalances. The technique proposed in this paper aims to prevent the long-term occupancy of the GPU between GPGPU tasks and to minimize latency.

4. Design and Implementation

4.1. System Overview

The proposed GPGPU task scheduling method is based on Xen 4.4.1—an open-source virtualization platform. However, our proposed method is not hypervisor-dependent, and it can be used in various virtualization environments. This section describes three subsystems of the GPGPU task scheduler in sharing GPU to multiple VMs. Figure 5 shows the overall structure of the GPGPU scheduler and the related modules proposed here.

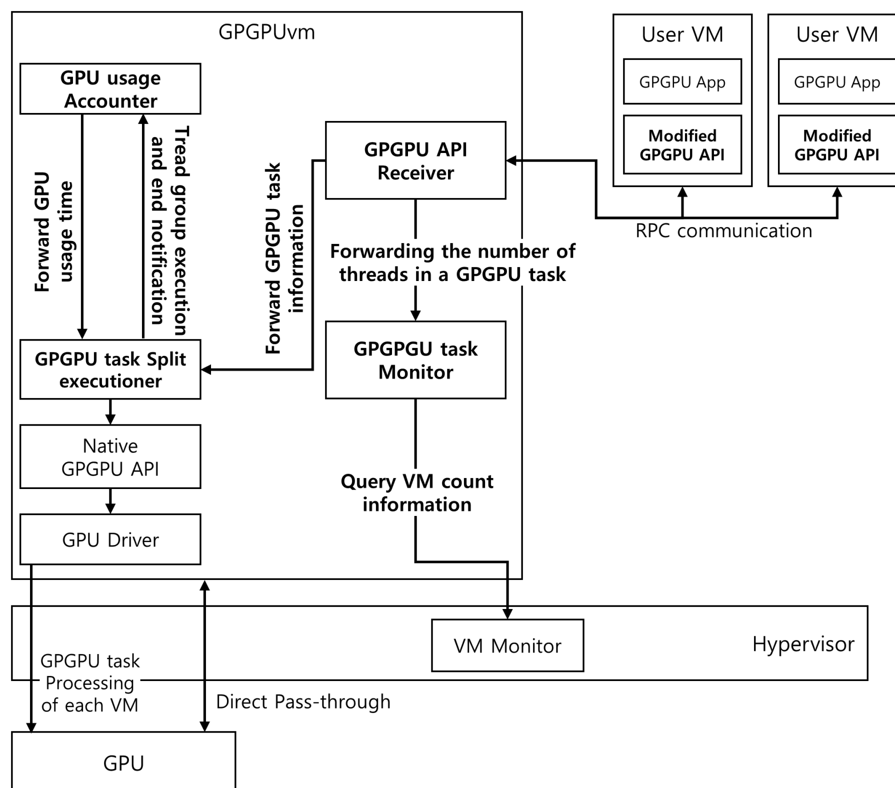


Figure 5. System overall structure.

Our proposed GPGPU scheduling technique was developed for use in an RPC-based GPU virtualization environment that redirects the GPGPU API. However, we believe that the idea of a scheduling method or of the GPGPU task management method of a VM can be applied to other types of GPU virtualization environments. The purposes of the GPGPU scheduling technique proposed in this paper are to prevent GPGPU tasks on certain VMs from monopolizing GPUs and to equalize latency for GPU use among VMs to prevent performance imbalances and performance degradation of certain VMs or monopolization of GPUs.

Our first goal is to equalize the scheduling latency of GPGPU tasks running concurrently. In a cloud environment, multiple VMs share the GPU and perform GPGPU tasks that are identical with or different from each other. However, because of the nature of GPGPU tasks that are not stopped during execution, the long-term GPU occupancy of long-running GPGPU tasks can cause the performance of other VMs to degrade—among

the VMs sharing the GPU. We used GPGPU task execution techniques based on GPU usage time to prevent GPU monopolization of GPGPU tasks with a long run time and to reduce the GPU usage latency for other VMs.

The second goal is to enable GPGPU tasks running on multiple VMs to use the GPU evenly during the scheduling period. The GPGPU task is typically programmed to use all the cores of the GPU and is set up to generate all the threads at once to make the most of the GPU resources. Based on information regarding the GPU usage time of each VM and the number of VMs running the current GPGPU task, the proposed approach adjusts the number of threads that GPGPU tasks can run at a time to prevent GPGPU tasks from monopolizing GPU resources.

The proposed scheme schedules GPGPU tasks performed by multiple VMs through three major subsystems, including RPC-based GPU virtualization technology: (1) GPGPU task monitoring, (2) GPU usage time accounting, and (3) GPGPU task split execution. In the first subsystem, the GPGPU task monitor records and manages information on GPGPU tasks, such as the start, end, number of threads to be processed, and status of thread processing. It also periodically manages information regarding the number of VMs running GPGPU tasks. The information in the GPGPU task monitor is used to schedule GPGPU tasks running on several VMs and to determine the size of the GPGPU task's thread execution. The second subsystem, the GPU usage time counter, records the GPU usage time of VMs by measuring the task execution time when each VM runs the GPGPU task. In the case of GPUs, simple run times should not be considered only because thousands of cores are present. The number of processed threads is recorded together with the GPU usage time to determine the weight for the GPU usage time of each VM. The third subsystem, the GPGPU task split executioner, determines the number of threads to start work among thousands to tens of thousands of threads for GPGPU tasks based on the information from the GPGPU task monitor and the GPU usage time accounter described earlier. The GPGPU task split executioner schedules the GPGPU tasks of each VM to use the GPU equally based on the GPU usage time of each VM. Subsystems for scheduling GPGPU tasks run on GPGPUvm that users cannot access. GPGPU tasks, which run on the user VMs, are executed via a scheduling technique proposed by GPGPUvm, regardless of the predefined workgroup configuration.

The GPGPU task scheduling technique proposed in this paper basically prevents all threads of GPGPU tasks from starting at once when the VM executes the GPGPU task, thus preventing the GPU resource monopoly. Thus, by dividing the threads of the GPGPU tasks into several groups and executing the thread groups according to the GPU usage time, GPGPU tasks on each VM support the use of GPUs as evenly as possible. The details of each subsystem are described in the following subsections.

4.2. GPGPU Task Monitoring

In cloud environments, resource monitoring technologies provide the most important information for managing computing resources and virtual machines. When virtual machine management systems create new virtual machines or allocate resources, they prevent overuse of resources based on information from resource monitors and allocate resources according to the capacity of resources required by each virtual machine. Typically, when resources such as CPU, memory, and storage are allocated to VMs in a cloud environment, they guarantee the resources needed by the user VMs, regardless of their actual resource usage, and manage resources to prevent other virtual machines from invading the resource area. Therefore, for CPU, memory, and storage, the resource allocation information of the VMs can be used to measure the resource usage of the host machine.

For GPUs, resource allocation and monitoring are complex, unlike other computing resources. For typical GPUs, unlike CPUs, GPU cores cannot be partitioned and assigned independently to VMs because they do not consider sharing for multiple users as described in the previous section. This means that unlike computing resources such as CPUs and memory, only a fraction of GPU resources cannot be allocated to a particular VM. In addi-

tion, unlike computing resources that are always used when a virtual machine is running, such as CPU, memory, and storage, GPUs have features that are not used unless the virtual machine performs operations using GPUs. Monitoring methods used to track resource usage on cloud platforms such as Open Stack and Open Nebula use a time-interval method that checks the usage of computing resources at regular intervals. However, due to the nature of GPU resources that are used only when a virtual machine performs GPGPU tasks without being assigned to a specific virtual machine as previously described, time-interval methods that do not recognize GPU usage result in unnecessary monitoring tasks.

Virtualization and cloud platforms such as Xen [31], KVM [35], and OpenStack [36] provide monitoring tools for computing resources such as CPU, main memory, and storage by default; but, no monitoring capability exists for GPU resources. Therefore, cloud providers must use a separate commercial GPU monitoring tool provided by the GPU vendors such as Nvidia management library [37] and GPU performance API [38] to monitor the status of GPU resources. However, existing commercial monitoring modules track the overall state of the GPU device, not the information regarding each GPGPU task. Furthermore, commercial GPU monitoring tools focus on the GPU devices dedicated to the data center, and some GPUs do not provide full functionality and are dependent on GPU vendors. Therefore, in this paper, we propose a GPU monitoring technique that can be handled event-based to recognize the timing of GPU use to prevent unnecessary monitoring tasks and collect individual information about GPGPU tasks of each virtual machine.

In this paper, the proposed GPGPU task scheduling uses an event-based lightweight monitoring method for querying the available GPU resources. The proposed GPGPU task monitoring technique records GPGPU task information for the VM immediately after receiving the task-related information through RPC communication when the GPGPU task of the VM is executed. It also records information on the GPGPU tasks of all VMs currently running on the GPU along with information on other VMs. The proposed GPGPU task monitors four types of information: (1) the number of VMs currently running GPGPU tasks, (2) the total number of threads that should be created in the GPGPU task of each VM, (3) the number of threads that have been processed in the GPGPU task of each VM, and (4) the number of unprocessed threads in the GPGPU task of each VM. The GPGPU task monitor runs in the GPGPUvm, which handles the GPGPU tasks of user VMs and records monitoring information while receiving APIs related to the GPGPU task requested from the user VMs.

The GPGPU task monitoring process proposed in this paper does not use a method that observes resource usage at regular intervals: The proposed method records the GPGPU task information whenever the user VM delivers API information through RPC communication or when the GPGPU task thread is executed. In addition, because the GPGPU task scheduler proposed here determines the thread execution size of the GPGPU task, information regarding its thread execution can be easily recorded whenever the thread is executed and completed.

Commercial GPU monitoring systems [37,38] cannot verify the detailed information of individual GPGPU tasks; they can verify only the overall state of the GPU—the usage of GPU cores and GPU memory. However, the proposed GPGPU task scheduling technique regulates the GPGPU task using details such as the number of threads to be processed. This requires detailed information such as the number of threads generated, the number of threads processed, and the number of threads remaining for each GPGPU task. The detailed information of the GPGPU task requested by each VM extracted by the proposed monitoring method is used in GPGPU task scheduling, described in the next section.

The first goal of the GPU monitor is to find a VM—among VMs running on the host machine—that executes the GPGPU task. The biggest difference between a typical computing environment and an environment where virtual machines sharing GPUs are not fixed is that the number of targets sharing GPUs is not known in advance. In a general computing environment, the number of GPGPU tasks to be executed on the GPU can be known in advance, before the task is executed. However, in a cloud environment

where there are multiple independent users using VMs, we can only check the number of people sharing the GPU after the task using the GPU is executed. In this paper, a two-step verification method was employed to check the number of VMs. In the first step, the GPGPU task monitor checks the number of VMs running on the server, and in the second step, the GPGPU task monitor checks the number of VMs that request GPGPU tasks through RPC communication to check the VMs that actually execute the GPGPU task. The first monitoring step is performed periodically to check the number of VMs running on the server, and the second monitoring step is executed only when the VM delivers the GPGPU API through RPC communication to execute the GPGPU task, and the task is completed.

When the number of virtual machines sharing the GPU is determined through a two-step monitoring method, and the virtual machine executes the GPGPU task, the GPU resource monitor records the GPGPU task execution information of each virtual machine as described above. In addition, the GPU resource usage time of each virtual machine is recorded through the GPU usage accounter described in the next section, and the information collected from the GPU resource monitor is used when scheduling each GPGPU task in the GPGPU task scheduler proposed in this paper. The GPU resource monitoring method proposed in this paper records information only when user VMs deliver APIs through RPC or when the kernel function of the GPGPU task is executed. This prevents unnecessary monitoring tasks, and in the case of GPGPU task execution information, only a few numeric data are recorded, so the impact on the entire system is not significant.

The proposed GPGPU task monitors can extract individual information of GPGPU tasks, such as the number of thread runs and the total number of threads, more detailed information than the commercial monitoring modules described earlier. Furthermore, existing resource monitoring techniques that run in a time-interval manner can make poor resource usage decisions because they cannot recognize variations between monitoring intervals. However, the GPGPU task monitoring technique proposed in this paper works event-based on GPGPU APIs' execution, enabling accurate GPGPU task monitoring. The utilization of monitoring information is described in detail in the next few subsections.

4.3. GPU Usage Time Accounting of User VMs

Xen, the open-source-based virtualization platform, adjusts the CPU usage time of the VM through the credit scheduler, which is based on the CPU usage time of the VM. In the credit scheduler, each virtual machine is assigned a credit value. The credit scheduler works by deducting credit by the amount of time spent on the CPU and is the first to handle the virtual machine's task with the highest credit value; thus, the VMs can use the CPU fairly when performing tasks. The CPU usage time of all VMs is tracked and managed to use the CPU fairly between VMs. Each VM uses CPU cores based on the CPU usage time assigned to it, and the VM scheduler accounts for CPU usage time by measuring the CPU core usage and usage time. For CPUs applied for high-performance servers, the number of cores available is approximately 18, and 36 threads are used, making it relatively easy to track the core usage time of each VM. However, for GPUs, it is difficult to keep track of all GPU cores because thousands of cores are present. In addition, as described earlier, the GPGPU task runs thousands of threads simultaneously; it is meaningless to track the GPU core usage time of VMs for each core. Therefore, the method of measuring resource usage time for VMs used in traditional cloud environments cannot be used for GPUs. Considering the characteristics of the GPGPU task, we propose a GPU usage time measurement method using weight values based on the number of running threads of each VM to measure the GPU usage time for each VM.

When the GPGPU task is executed for VMs, GPGPUvm manages the GPU usage time for each VM for scheduling the GPGPU task. The GPU usage time accounter measures the GPU usage time of each VM when a thread group of each VM is run. In this paper, OpenCL, a GPGPU programming API, was modified to build an RPC-based GPU virtualization

environment. The GPU is the co-processor of the CPU; all operations of the GPGPU task start are handled by the CPU, and only the GPU computation portion is handled by the GPU. Because of these characteristics, when measuring GPU usage time for VMs, accurate GPU usage time has to be measured running time of the kernel function—that is, the running time of thread performs GPU operations—rather than by measuring the time of use from the time the GPGPU task is executed. To measure only the GPU computation time of a VM, OpenCL's clGetEventProfilingInfo() function is used to measure the execution time of each VM's kernel function (i.e., GPU computation time). When only one virtual machine runs the GPGPU task, it does not record GPGPU usage time because there is no resource sharing target, but when two or more virtual machines run GPGPU tasks, GPU usage time is recorded to compare GPU usage time.

GPGPU tasks run thousands to tens of thousands of threads. In environments where only a single GPGPU task is run, simply measuring the running time of the GPGPU task can determine the GPU usage time of the GPGPU task. However, if a large number of independent users, such as the environment in this paper, are running GPGPU tasks simultaneously, simply the execution time information of GPGPU tasks cannot determine GPU usage time. For example, in an environment where two GPGPU tasks are run, if the GPGPU tasks have the same running time of 2 s, but one GPGPU task runs 100 threads and the other runs 200 threads, information about the running time of GPGPU tasks makes it appear that two GPGPU tasks use GPUs for the same amount of time, but the two tasks did not use GPUs the same because the number of threads executed was different. Therefore, we measure GPU usage time for each virtual machine by considering the number of thread runs of each GPGPU task. The VM n's GPU usage time is given as follows:

$$\begin{aligned} &VM\ n's\ GPU\ usage\ time \\ &\qquad = GPGPU\ task\ runtime * \left(1 + \frac{VM\ n's\ number\ of\ running\ thread}{all\ VM's\ number\ of\ running\ thread}\right) \end{aligned} \tag{1}$$

The VM's GPU usage time is managed individually for each VM. In the formula, *GPGPU task runtime* is the execution time of the GPGPU task of the virtual machine. VM n's number of running threads is the number of threads executed by virtual machine n, and all VM's number of running threads is the number of threads of all virtual machines executed in the current scheduling round.

The GPGPU task scheduling technique proposed here regulates aspects such as the number of threads to execute and the timing of execution of the GPGPU task executed by VM according to the GPU usage time of each VM. Therefore, the number of threads running on each VM and the timing of the execution may differ depending on the size of the GPGPU task. In an environment where GPUs are not shared by multiple users, to measure the running time of the GPGPU task, the time the task starts and the time it is finished must be simply measured. However, when several users need to share resources and provide equal resource usage, measurement of resource usage time, including resource usage rate, is required. In this paper, we measure the GPU usage time, weighted by the number of threads running, to measure the GPU usage time by referring to the GPU usage time. The method used to count the GPU usage time proposed in this paper can measure the GPU usage time by considering the GPU resource utilization rate because the measured GPU usage time includes the weight value through the number of running threads, even if the measured GPU computation time of each VM is the same. In addition, a VM's GPU usage time measurement method based on the number of threads executed and the execution time of threads helps measure the GPU usage time efficiently regardless of the running time of the GPGPU task and the number of threads.

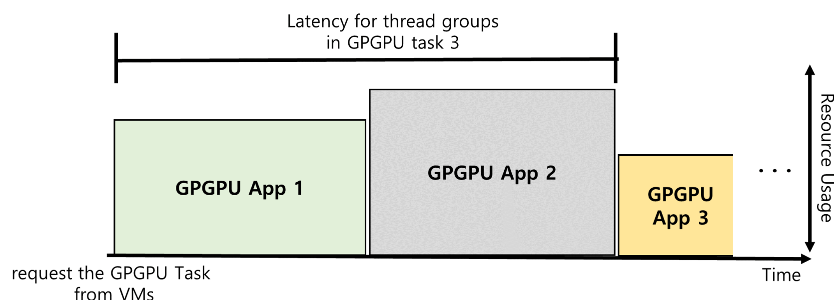
4.4. GPGPU Task Split Execution

The GPGPU task scheduling method in this paper is intended for environments where several VMs run GPGPU tasks simultaneously. Our proposed approach divides

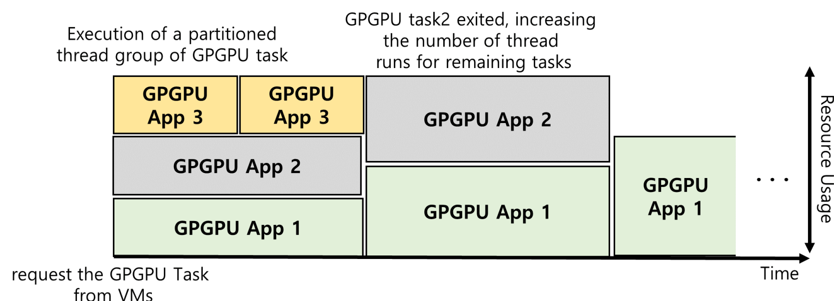
the threads of the GPGPU task and executes only part of all of the threads to avoid the problem of occupying the GPU by running all of the threads of a particular VM at once. The previously described GPGPU task monitor and GPU usage time counter manage information for scheduling GPGPU tasks. Based on information regarding the GPGPU task and the number of VMs, the GPGPU task scheduler decides the GPGPU task's threads into what scale and at what point they are to be executed.

The basic working method of the GPGPU task scheduling technique proposed here is to divide the threads of the GPGPU tasks into several groups and execute the segmented thread group of each VM simultaneously. The GPGPU task for each VM is divided into multiple thread groups, and the GPGPU task on each VM runs only one group of divided threads when using GPU simultaneously, thus minimizing GPU competition and preventing the GPGPU task for a particular VM from using the entire GPU core. In a typical computing environment, the GPGPU application is developed to make the most of the GPU resources to achieve maximum performance. However, in a cloud environment, computing resources are shared across multiple VMs or by multiple users, and cloud administrators must support each VM's equal use of the resources.

Because of the operating characteristics of the GPGPU task, when a thread of the GPGPU task is executed, the context switching or the order of execution of the thread of each GPGPU task cannot be arbitrarily determined from the outside. Thus, in this paper, to prevent GPU resource monopoly of specific VMs, the method of partitioning the threads of GPGPU tasks and executing the partitioned thread group of each VM at the same time is used. In existing GPGPU task execution environments, threads of the GPGPU task on other VMs must be on standby if the threads of the GPGPU task on a particular VM are using the entire GPU core, as shown in Figure 6a. However, the proposed GPGPU task-scheduling techniques, as shown in Figure 6b, limit the amount of GPU resources a single VM can use because each VM executes a group of divided threads. When one virtual machine sharing GPU resources completes the task, it increases other virtual machines' thread group size. This can reduce the latency of GPU usage for certain VMs and mitigate the problem of performance imbalance between VMs.



(**a**) Existing GPGPU task execution environment.



(**b**) The thread division processing method proposed in this paper.

Figure 6. Partitioning GPGPU task in an environment where multiple GPGPU task is running.

As described earlier, it is not possible to generate arbitrary context switching while running a thread when executing a GPGPU task. Thus, the method of stopping a task while running on a VM based on resource usage time, like a VM scheduler, and transferring computing resources to another VM through context switching cannot be used. In this paper, each VM uses a method of partitioning the threads of GPGPU tasks so that it can use GPU resources as simultaneously as possible while eliminating constraints that cannot generate context switching arbitrarily. In this paper, the most critical information in scheduling GPGPU tasks is the number of virtual machines sharing GPUs and each virtual machine's GPU usage time. We divide the thread group of GPGPU tasks based on the number of VMs sharing GPUs. Information on the number of VMs can be obtained on the number of VMs sharing GPUs before the GPGPU task thread, as described in the previous chapter because uploading data to GPU memory is mandatory before the kernel function runs. However, the GPU usage time for each VM is unknown until the kernel is run, so it is difficult to determine the size of the initial thread group to run. Therefore, in the first scheduling round, there is no information such as the execution time of the kernel function executed in each VM, so a predefined fixed amount of threads is executed. Because the running time and number of threads of the GPGPU task that each VM executes may vary, the threads of each GPGPU task are initially grouped into the same number of threads. In the next round, the number of executed threads is determined based on the GPU usage time. The threads that make up the thread group are as follows:

$$Size\ of\ Thread\ group = [\frac{(Number\ of\ GPU\ Cores * \beta)}{Number\ of\ VMs}/64] * 64 \quad (2)$$

In the formula, the *Number of GPU Cores* refers to the number of computational cores of the GPU; β is a variable obtained through experiments that assess the maximum number of run threads required to minimize performance degradation that occurs as a consequence of an increase in running threads. The Number of VMs is the number of virtual machines that perform GPGPU tasks at a given time. In the AMD GPU, we used to implement the RPC-based GPU virtualization environment and the minimum scheduling unit when executing OpenCL tasks. The thread group—frontwave (called "warp" in Nvidia)—contains 64 threads (32 threads in Nvidia). So, in our work, the thread group is executed in multiples of 64. A number of virtual machines execute a fixed number of threads in the first round of scheduling; afterward, the number of threads to be executed in each scheduling round is adjusted according to the GPU usage time.

The GPGPU task scheduling technique in this paper performs scheduling tasks only when VMs run GPGPU tasks. The GPGPU task scheduler waits in its initial state until the VMs perform GPGPU tasks. In the initial state, the process of monitoring information from the VM administrator identifies the number of VMs running on the current server and classifies all VMs as potential GPU users. When the number of VMs that actually use the GPGPU task is determined, the partitioned thread group that runs the GPGPU task on each VM adjoins the thread groups on other VMs. Because the thread of the GPGPU task is divided, the kernel function of the GPGPU task is processed several times. This can adjust the scale of thread execution of the GPGPU tasks and can indirectly limit the use of GPU cores for GPGPU tasks of each VM to prevent long-term occupancy of GPU cores. Unlike CPU virtualization technology, GPU cores cannot be assigned specific computational cores to VMs, as described earlier.

Thus, in this paper, the execution scale of the thread executing the GPGPU task is adjusted to prevent the long-term monopoly of the GPU core of certain VMs. It also measures the GPU usage time based on the GPU usage time accounting information for each VM described earlier to help ensure equal GPU usage time by pausing the execution of thread groups of VMs with higher GPU usage time compared to other VMs and by executing more threads on other VMs. The GPGPU task scheduler proposed here works as in Algorithm 1.

In Algorithm 1, lines 1 to 15 are structures for managing scheduling-related information when scheduling the GPGPU tasks in our work. GPGPU tasks do not work at all times like CPU tasks. The CPU is used not only for the user's application program but also for various tasks for operating the OS, such as background services; the GPU is only used when performing GPGPU tasks in the virtual machine. As shown in line 16, our work prepares a scheduling task for the GPGPU task when the user virtual machine requests a GPU memory allocation task. When the kernel function execution is requested after completing the GPU memory allocation requested by the VM, basic information of the GPGPU task is obtained, as shown in lines 19 to 22, and the preparation step for the scheduling task is performed. As shown in lines 24 to 26, if the GPU usage time of a specific VM is more than twice that of the virtual machine with the least GPU usage time, the GPGPU work with the most GPU usage time is stopped while one scheduling round.

Algorithm 1 GPGPU task processing algorithm

```
1:  Struct Sched_Info{
2:     int Num_VM;                  //The number of virtual machines to be scheduled
3:     int Sched_VM_list[]          //VMs ID to be scheduled
4:     float GPU_UsageTime[]        //all VM's GPU usage time
5:     float GPU_usage_minVM        //GPU usage time of VM with least use GPU
6:  }
7:  Struct Task_Info[]{
8:     float GPU_UsageTime            //VM n's GPU usage time
9:     int Total_Thread_Num           //Number of threads in VM n's kernel functions
10:    int Thread_Per_schedRound      //Number of threads to be executed in each round
11:    int Remain_Thread              //Number of threads not yet processed
12:    int Curr_SchedRound            //Current scheduling round
13:    float Prev_Task_runtime        //Task run time of VM n at previous round
14:    float Prev_GPU_UsageTime       //GPU usage time of VM n at previous round
15: }
16: if VM n requests GPU Memory Allocation then
17:     GPU Memory allocate for VM n;
18:     if VM n request Kernel Function then
19:         Sched_Info->Sched_VM_list[] = VM n's ID;
20:         Sched_Info->Num_VM += 1;
21:         Get Kernel Function parameter;
22:         Task_Info[n]->Remain_Thread = Task_Info[n]->Total_Thread_Num;
23:         while Number of threads to process > 0 do
24:             if  (Sched_Info->GPU_usage_MinVM)*2
25:                                 < Task_Info[n]->Prev_Task_runtime then
26:                 wait_time = Task_Info[n]->Prev_GPGPUTask_runtime;
27:                 wait(VM n, wait_time);
28:             else
29:                 Thread Group size Determination Function;
30:                 VM n's Kernel Function(Task_Info[n]->Thread_Per_schedRound);
31:                 Task_Info[n]->GPU_UsageTime += GPGPU task runtime
32:                                 *(1+(VM n's Number of running thread
33:                                   /all VM's number of running thread));
34:                 Task_Info[n]->Remain_Thread
35:                                 -= Task_Info[n]->Thread_Per_schedRound;
36:                 Task_Info[n]->Curr_SchedRound += 1;
37:             end if
38:         end while
39:     end if
40: end if
```

The kernel functions of the virtual machine are divided into several thread groups and executed sequentially, recording information for controlling the execution of GPGPU tasks, such as the number of executed threads and GPU usage time. The thread group size determination function on line 28 is described in detail in Algorithm 2. Basically, our GPGPU task scheduling method ends when all threads of the kernel function executed by each virtual machine are completed. As in Algorithm 1, the GPGPU task scheduler operates in three stages, depending on the number of VMs using GPUs: (1) when only one VM requests a GPGPU task, (2) when GPGPU tasks are running, and (3) when one or more VMs have completed GPGPU task processing. The GPGPU task scheduling algorithm basically starts scheduling when two or more user VMs run the GPGPU task. The first run thread group of each VM's GPGPU task is split while data are being copied for the GPGPU from the main memory of the user's VM to the GPU memory of GPGPUvm. When the partitioned thread group of the GPGPU tasks is executed and the GPU usage time is measured, Algorithm 2 determines the size of the thread group to be executed in the next scheduling round.

Algorithm 2 Thread group size determination algorithm

```
1:  if Task_Info[n]->Remain_Thread > 0 then
2:      if Task_Info[n]->Remain_Thread > Task_Info[n]->Thread_Per_schedRound then
3:          if Task_Info[n]->Curr_SchedRound == 1 then
4:              Task_Info[n]->Thread_Per_schedRound
5:                  = (int)(((Number of GPU Core * β)/Sched_Info->Num_VM)/64)*64;
6:          else
7:              for Sched_Info->GPU_UsageTime[n]=1,2,...,n do
8:                  All_VMs_GPU_UsageTime += Sched_Info->GPU_UsageTime[n];
9:              end for
10:             GPU_Usage_ratio
11:                     = Task_Info[n]->GPU_UsageTime / All_VMs_GPU_UsageTime;
12:             Task_Info[n]->Thread_Per_schedRound
13:             Reconf_Thread_Num = Task_Info[n]->Thread_Per_schedRound
14:                                 +(((1 / Sched_Info->Num_VM)-GPU_Usage_ratio)
15:                                     *All_VMs_GPU_UsageTime);
16:             Task_Info[n]->Thread_Per_schedRound =
17:                                 Task_Info[n]->Thread_Per_schedRound
18:                                 + ((int)(Reconf_Thread_Num/64)*64)
19:         end if
20:     else
21:         Task_Info[n]->Thread_Per_schedRound = Task_Info[n]->Remain_Thread;
22:     end if
23: else
24:     Sched_Info->Num_VM -= 1;
25:     Remove VM n's ID to Sched_Info->Sched_VM_list[];
26: end if
```

Algorithm 2 runs on a separate thread from Algorithm 1. Algorithm 2 redefines the number of threads to execute the kernel function according to the GPU usage time of each VM while $t+1$ is running after scheduling round t is completed. Algorithm 2 updates scheduling information so that it can be applied to scheduling round $t+2$. If redefined information is used for each scheduling round, more accurate scheduling is possible, but the kernel function has a problem of having to wait until the newly defined thread size is determined for each round. Therefore, in our work, Algorithm 2 is executed in a separate thread so that the scheduler can update the scheduling-related information, and new information can be applied when reading the GPGPU tasks-related information.

When the thread group executed in the first scheduling round is completed, the GPGPU task scheduler knows the runtime when each GPGPU task is executed with the same num-

ber of threads according to the information in the GPU usage time accounter. The size of the redefined thread group is decided from the second scheduling round, and the number of execution threads over the GPU usage time is applied for each VM from the third round because all GPGPU tasks must be paused for deciding and immediately applying the size of the redefined thread group. For example, for VMs with relatively more GPU usage time than for other VMs, at scheduling round t, the VM runs until $t + 1$ and does not run the thread groups that should run at scheduling round $t + 2$, and then, it runs again in scheduling round $t + 3$ after scheduling round $t + 2$ for only other VMs. It also adjusts the size of the should run thread group of GPGPU tasks at scheduling round $t + 1$ for other VMs and executes scheduling round $t + 2$ when the GPGPU task for VMs with high GPU usage is paused in scheduling round $t + 2$.

In our approach, the size of the newly redefined thread group according to the GPU usage time of scheduling round t is decided while scheduling round $t + 1$; thus, there is no need to wait until the new thread size is determined. To prevent temporary suspension of the GPGPU task, scheduling round $t + 1$ is executed in the same manner as scheduling round t, and a new thread group size is applied in scheduling round $t + 2$. Because the running time of the GPGPU task is likely to be different, the time interval of the scheduling round is not the same, and a separate logical scheduling round is used for each GPGPU task. Finally, when the GPGPU task of more than one user's VM is completed, the task is excluded from the scheduling target VM, and the result of the operation is returned; then, the other user's VM executes more threads.

As described earlier, each thread that executes the GPGPU task cannot be mapped directly to the GPU core when executing the GPGPU task; therefore, here, unlike the FIFO-based scheduling environment in which all of the threads are called and executed at once, the thread of the GPGPU task is divided into several groups, and the scale of the execution thread is adjusted depending on the number of VMs performing the GPGPU task to prevent monopolizing the GPU. Unlike VM schedulers with time slices, our method is operated by sequentially executing a group of divided threads of GPGPU tasks and correcting the number of threads to run based on the GPU usage time. In addition, unlike traditional scheduling methods for GPGPU tasks, it is relatively easy to apply the method to other virtualization systems because there is no need to change system components such as GPU drivers, OSs, and hypervisors. The following section verifies the efficiency of the GPGP task scheduling techniques proposed here through experiments and discusses the limitations of the techniques proposed in this paper.

5. Experiment

This section evaluates the performance of the proposed GPGPU task scheduling technique and validates its efficiency through experiments. The proposed technique was developed based on the open-source-based virtualization platform, Xen hypervisor 4.4.1. For performance evaluation, we used the experimental environment for an RPC-based GPU virtualization environment using a Radeon RX 570 GPU with 8 GB of dedicated GPU memory. The GPU was assigned to GPGPUvm using direct pass-through, and other VMs could not access the direct GPU. The guest OS of VMs was Windows 7, as listed in Table 1. The GPGPU task to be executed in the VM was matrix multiplication implemented with OpenCL.

We evaluated the effectiveness of the proposed GPGPU task scheduling method in Section 4 to minimize the performance deviations between VMs that occur when multiple VMs run GPGPU tasks simultaneously. This experiment divides the 6 VMs into two groups and performs matrix multiplication of different sizes, as in Section 4. The purpose of this experiment was to determine how much performance deviation was reduced when each GPGPU task was run several times. Prior to constructing the experimental environment, we confirmed that in the RPC-based GPU virtualization environment used in this paper, the performance of the GPGPU task decreased as the capacity of the GPU memory used by the GPGPU task executed simultaneously increased.

Table 1. Experiment environment.

	Host machine	**GPGPUvm**	**User VM**
CPU	Intel Xeon E3-1231 v3 (4 Cores, 8 Threads)	4 vCPU	2 vCPU
Memory	32 GB	15 GB	2 GB
HDD	1 TB	100 GB	50 GB
GPU	Not used for host OS	Radeon RX 570 (8 GB)	-
OS	Ubuntu 14.04	Windows 7	Windows 7
Hypervisor	Xen 4.4.1	-	-

As shown in Table 2, the dedicated memory of the GPU used in the experiment was 8 GB; and, it caused some acceptable performance degradation until the GPU memory usage was less than 10 GB. As shown in Table 2, a sharp drop in performance occurs when the GPU usage exceeded approximately 10 GB. Therefore, the experimental environment was configured such that the GPGPU tasks consumed less than 10 GB of GPU memory to minimize performance degradation caused by competition in the GPU memory. The GPGPU task of the VM was repeated 10 times to measure the mean performance and performance deviation. In addition, the GPGPU task was scaled so that VMs use less than 10 GB of GPU memory to prevent overuse of GPU memory from impacting performance because the overhead due to GPU memory overuse deviates from the scope of this paper.

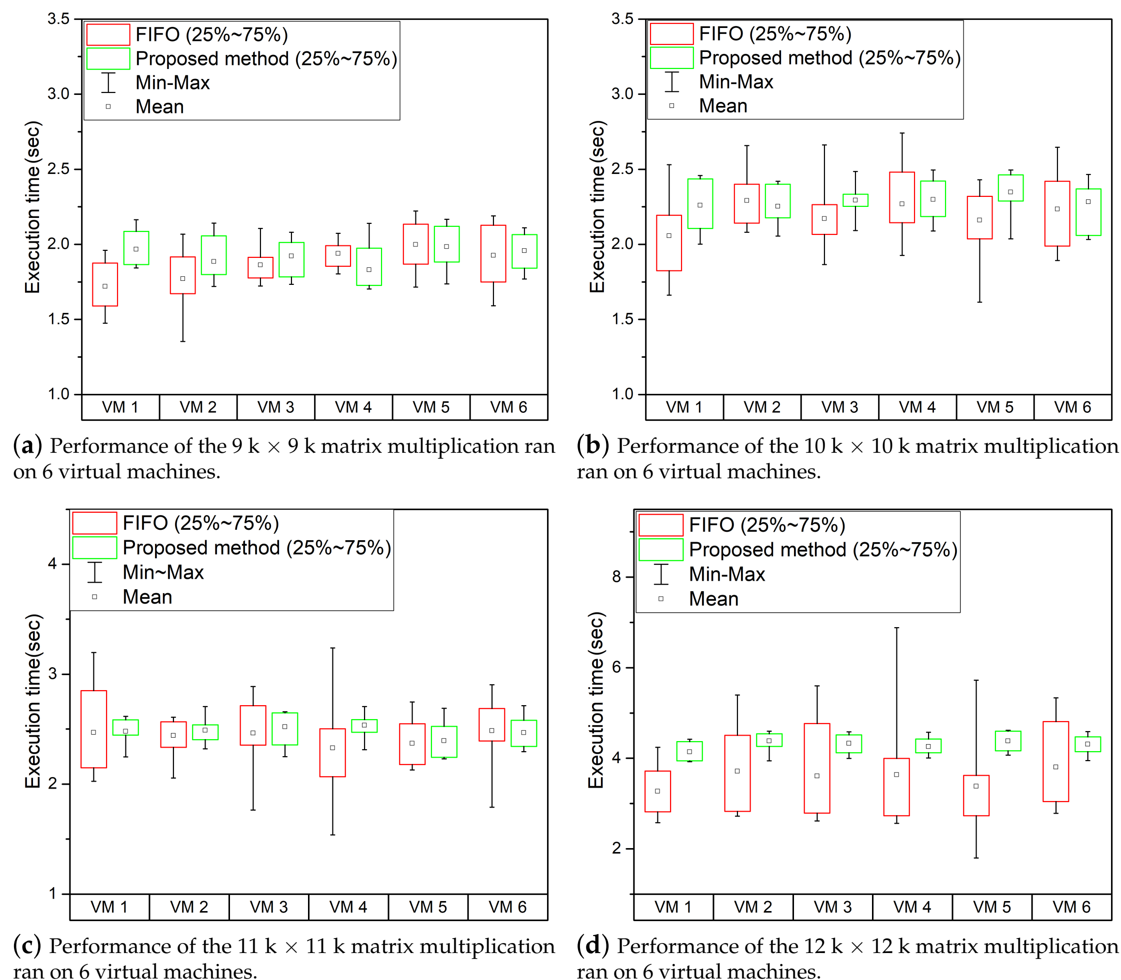
Table 2. GPU memory requirements according to the number of VMs and matrix size.

	12 K × 12 K		**13 K × 13 K**		**14 K × 14 K**	
Number of VM	**Memory Usage**	**Execution Time**	**Memory Usage**	**Execution Time**	**Memory Usage**	**Execution Time**
1	1.6093	0.936	1.8887	1.0450	2.1904	1.155
2	3.2186	1.3965	3.7774	1.5760	4.3809	1.88
3	4.8279	1.872	5.6661	2.1633	6.5714	2.2826
4	6.4373	2.1992	7.5548	2.7297	8.7618	4.1107
5	8.0466	2.6206	9.4436	3.2950	**10.9523**	**11.3974**
6	9.6559	5.213	**11.3323**	**11.9240**	13.1428	19.877
7	**11.2652**	**9.32671**	13.221	22.8784	15.3332	21.09
8	12.8746	21.9885	15.1097	23.7691	17.5237	27.5894

We performed two experiments. First, the same GPGPU tasks were run at the same time, and in the second experiment, the VM was divided into two groups, and different GPGPU tasks were executed (9 K and 14 K, 10 K and 14 K, and 11 K and 13 K). In the combination used in the second experiment, the VMs used 9.3 GB, 9.9 GB, and 9.7 GB, respectively, when running GPGPU tasks. The experimental results are the same as those in Figures 7 and 8.

Figure 7 shows the time required to run all the same GPGPU tasks at the same time. The performance deviation was relatively small compared with the simultaneous execution of GPGPU tasks of different sizes, as shown in Figure 7. If the GPGPU task size is the same, each thread group will have the same run time, and all GPGPU tasks will have the same number of threads. Because all GPGPU tasks are run under the same conditions, relatively less performance deviation occurs than when GPGPU tasks of different run-time are run. However, in the worst case, the performance difference is roughly doubling, resulting in the inconsistent and unpredictable performance of each VM. In contrast, as shown in Figure 8, when using the proposed GPGPU task scheduling technique reduces performance deviation of GPGPU tasks. Our work reduces the performance deviation by determining the size

of the GPGPU task's thread execution based on GPU usage time; but, the performance is slightly degraded by the added work for scheduling. However, performance degradation due to GPGPU task scheduling is minor compared to the performance deviations.



(**a**) Performance of the 9 k × 9 k matrix multiplication ran on 6 virtual machines.

(**b**) Performance of the 10 k × 10 k matrix multiplication ran on 6 virtual machines.

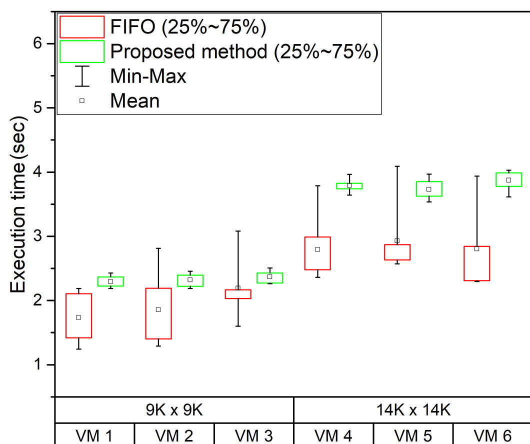
(**c**) Performance of the 11 k × 11 k matrix multiplication ran on 6 virtual machines.

(**d**) Performance of the 12 k × 12 k matrix multiplication ran on 6 virtual machines.

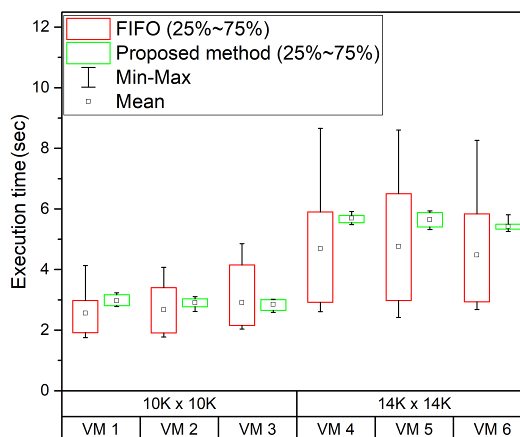
Figure 7. Performance of newly requested GPGPU task in the case of lack of GPU memory.

Figure 8 shows the experiment results in an environment in which different GPGPU tasks are executed simultaneously. In the experiment shown in Figure 8, six VMs were divided into two groups: One group ran relatively short-running GPGPU tasks, and the other group ran relatively long and large-scale GPGPU tasks. As shown in Figure 8, the performance deviation of GPGPU tasks occurs significantly in FIFO-based scheduling environments, as described in the previous section. However, when the GPGPU task scheduling technique proposed in this paper was applied, the performance deviation was considerably reduced compared with that in the FIFO-based scheduling environment. In particular, the GPGPU task, which has a relatively short running time, showed that the performance deviation of all GPGPU tasks was considerably reduced; the performance deviation of 25% to 75% of all GPGPU tasks was reduced overall, and consistent performance was achieved. If using the proposed GPGPU task scheduling method, GPGPU tasks with short run times do not require a long wait time for GPGPU tasks with long run times. GPGPU tasks with long run times will be measured with high GPU usage times, thus increasing the probability that GPGPU tasks with short run times can use GPUs. It also prevents certain VMs from monopolizing GPUs because the size of the GPGPU task that each VM can run is limited. Similar to the environment in which the same GPGPU task was executed earlier, the proposed GPGPU task scheduling method increases the average

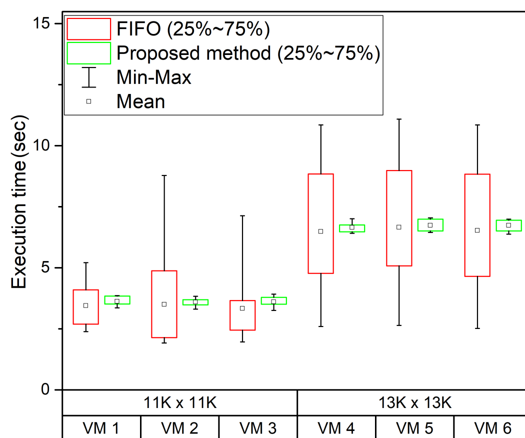
performance of the GPGPU task but has little effect on the overall performance compared to the worst performance due to the performance deviation.



(**a**) Performance of the 9 k × 9 k and 14 k × 14 k matrix multiplication ran on 2 VM groups.



(**b**) Performance of the 10 k × 10 k and 14 k × 14 k matrix multiplication ran on 2 VM groups.

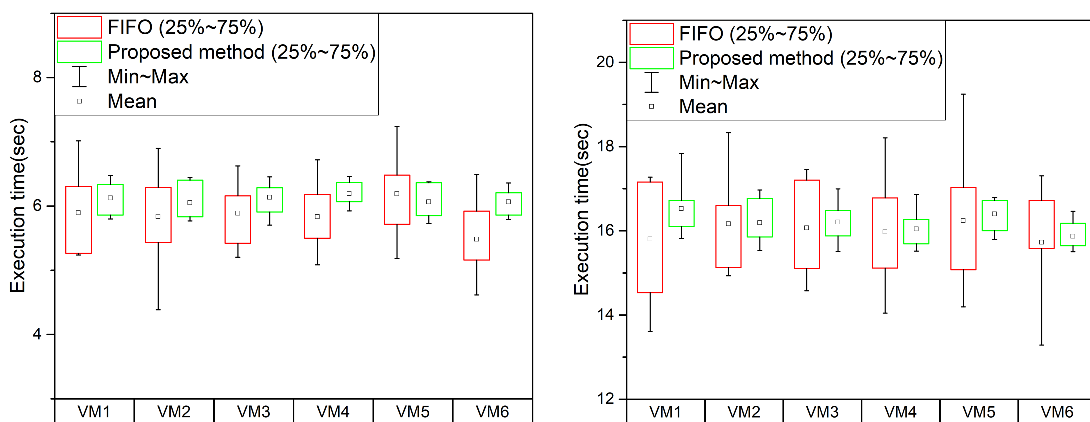


(**c**) Performance of the 11 k × 11 k and 13 k × 13 k matrix multiplication ran on 2 VM groups.

Figure 8. In an environment where different GPGPU tasks are executed in 2 VM groups at the same time, reduced the performance deviation by our proposed method.

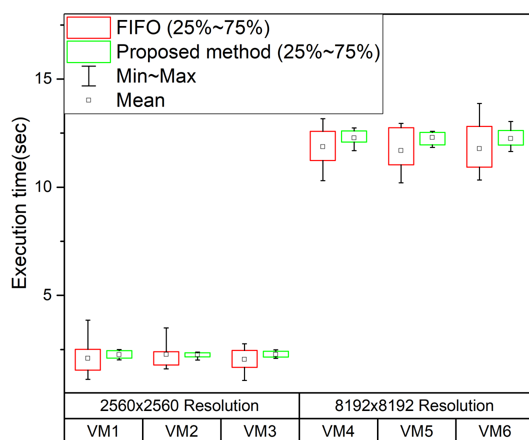
A significant performance deviation in these environments was observed, as was the case in the previous section. Unlike CPU that usage time-based scheduling techniques and context-switching, GPUs that schedule tasks on a thread unit-basis have limitations in using GPUs equally for each GPGPU task. However, the proposed GPGPU task scheduling method shows that it reduces the performance deviation of each VM. Reducing performance deviation means that performance consistency can be achieved when the same task is executed under the same conditions. In situations where the same GPGPU task is executed simultaneously, the proposed GPGPU task scheduling technique adjusts only the degree of execution of the GPGPU task based on the GPU usage time of each VM. When GPGPU tasks of different run-time are run, a GPGPU task can be adjusted based on each VM's GPU usage time to avoid the long-running GPGPU task's GPU occupancy time and to reduce the latency for GPU use in short-running GPGPU tasks. In the FIFO-based scheduling environment, the same GPGPU task under the same conditions results in significant performance deviations each time; however, as shown in the experimental results in Figures 7 and 8, the proposed method can reduce performance deviations. In addition, the experimental results show that efficiency is better in environments with GPGPU tasks of different sizes.

In previous experiments, performance evaluations were performed with matrix multiplication of various sizes. In the following experiments, we conduct experiments using different applications to verify that the efficiency of the GPGPU task scheduling method proposed in this paper. In this experiment, we use the Sobel Image Filter, an image processing application that finds the image's outline. Experiments show the performance of the GPGPU task scheduling technique proposed in this paper when running the Sobel image filter on six virtual machines. The experiments use six virtual machines and the results as shown in Figure 9.



(**a**) Performance of the Sobel image filter using 5120 × 5120 resolution image on 6 VM.

(**b**) Performance of the Sobel image filter using 8192 × 8192 resolution image on 6 VM.



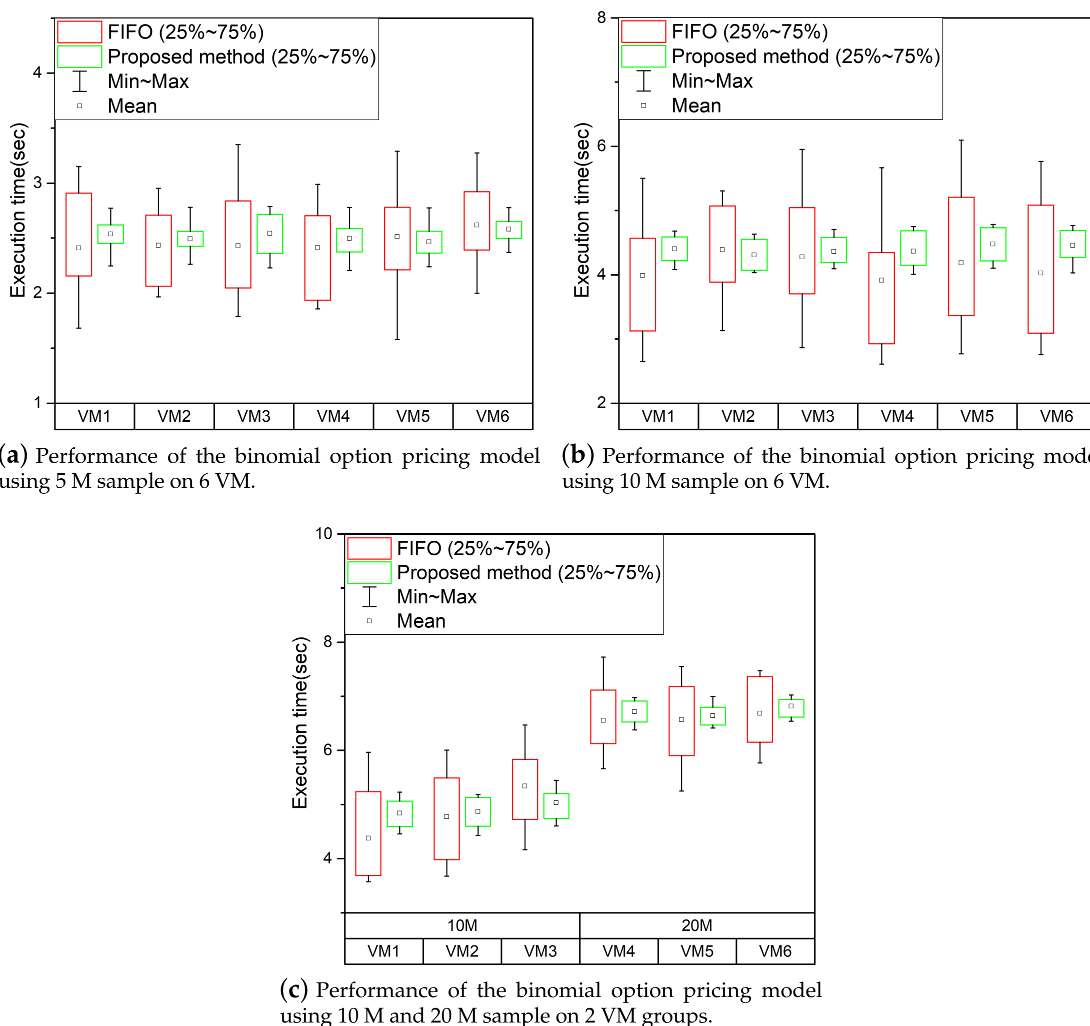
(**c**) Performance of the Sobel image filter using 5120 × 5120 and 8192 × 8192 resolution image on 2 VM groups.

Figure 9. Performance of the Sobel image filter on 6 VMs.

Figure 9a,b show the results of running the Sobel image filter using images with a resolution of 2560 × 2560, 8192 × 8192, respectively. Figure 9c performs the Sobel Image Filter task using images with 2560 × 2560 resolution on three virtual machines and 8192 × 8192 resolution on the other three virtual machines. While large performance deviations occurred in FIFO-based scheduling environments, as shown in Figure 9, using the proposed GPGPU task scheduler can reduce performance deviations. In particular, in the worst case, the execution time increases by about twice. It also shows similar results to the previously performed matrix multiplication experiments.

The application to be used in the following experiments is the binomial option pricing model, which determines the finance option price. This experiment also uses six virtual machines to perform the binomial option pruning model, as previously performed. The experimental results are shown in Figure 10.

The experimental results are shown in Figure 10a,b show the performance when running the binomial option pricing model, respectively, when the number of samples is 5 M and 10 M. As shown in the experiments, performance consistency is not guaranteed in FIFO-based scheduling environments because it does not consider GPU usage time or even thread execution of virtual machines. Figure 10c shows the performance of running GPGPU tasks simultaneously using 10 M samples in three of the six virtual machines and 20 M samples in the other three. As shown in the experimental results, there has been a significant performance deviation between GPGPU tasks in existing FIFO-based scheduling environments. However, the technique proposed in this paper has reduced the performance deviation between GPGPU tasks of VMs.



(**a**) Performance of the binomial option pricing model using 5 M sample on 6 VM.

(**b**) Performance of the binomial option pricing model using 10 M sample on 6 VM.

(**c**) Performance of the binomial option pricing model using 10 M and 20 M sample on 2 VM groups.

Figure 10. Performance of the binomial option pricing model on 6 VMs.

When using the GPGPU task scheduling technique proposed in this paper, thread segmentation results in minor performance degradation, but significantly reduces performance deviations. The threads running GPGPU tasks are divided into several groups and are processed several times. In the proposed GPGPU task scheduling method, an overhead occurs because each VM performs some additional work such as thread segmentation, GPU usage time measurement, and thread generation to use the GPU equally. However, performance degradation due to additional overhead does not have much impact than FIFO-based scheduling environments.

6. Discussion and Limitation of Proposed GPGPU Task Scheduler

In this paper, our proposed GPGPU task scheduling technique prevents the GPU monopolization of long-running GPGPU tasks in RPC-based GPU virtualization environments and minimizes performance deviation. The proposed techniques reduced the GPGPU task performance deviation of the VM, as shown in the experiment in the previous section. Furthermore, the GPGPU task performance degradation problems due to GPU occupancy of the long-running GPGPU task and addressed the problems of the performance deviation of the FIFO-based scheduling environment. As described earlier, we considered the existing environment in which the threads of the GPGPU task cannot be mapped directly to the GPU core, indirectly preventing certain VM threads from monopolizing GPU cores by dividing them into several thread groups instead of calling and executing them at once. Besides, when a virtual machine runs GPGPU tasks, each time it runs a GPGPU task, the GPGPU task is scheduled based on GPU usage time, so it can be handled well even if the virtual machine operates several types of GPGPU tasks.

The GPGPU task scheduling technique proposed in this paper uses partitioning of GPGPU tasks into multiple thread groups to prevent GPU long-term occupancy of specific virtual machines. GPGPU task segmentation methods can affect the overall performance of GPGPU tasks due to the segmentation of threads in the case of GPGPU tasks with short execution times. GPGPU tasks, which do not significantly impact other GPGPU tasks due to short execution times, can be considered for handling the threads at once without splitting them. However, it is hard to pre-find GPGPU tasks with a short execution time because the GPGPU task of the user VM does not know the running time until it runs. Furthermore, arbitrarily handling GPGPU tasks of a particular virtual machine before those of other virtual machines is challenging because it must consider fairness between virtual machines.

Furthermore, as previously described, non-context switching causes GPU long-term occupancy problems when handling GPGPU tasks with very long running times. The GPU long-term occupancy problem intensifies with unknown constraints on GPGPU tasks. However, in this paper, it is possible to avoid long-term resource occupancy regardless of the thread's running time because multiple virtual machines segment GPGPU tasks when executing GPGPU tasks. If a new virtual machine runs a GPGPU task while a single GPGPU task with a long-running time of the thread is running, a long wait time for the newly executed GPGPU task occurs. In this paper, processing methods for newly executed GPGPU tasks are not supported because tasks are scheduled based on GPGPU task segmentation. If a GPU is already occupied and runs a new GPGPU task, we can consider using the existing proposed GPGPU task's stop and restart techniques such as Crane [39] to pause the task that is running and adjust it for other virtual machines to use the GPU.

The experiments in the previous section have also shown that the performance deviation of the GPGPU task running at the same time is reduced; however, the average performance is slightly degraded. This is caused by the added step of dividing the threads of the GPGPU task in the GPGPU task scheduling proposed here. When a GPGPU task is executed in a VM, all of the same work is treated as independent work, and the task of determining the size of the thread division is performed whenever new work is requested. These parts can be identified by historical execution records or analysis of kernel code to identify the same tasks that have been performed in the past and help start the tasks directly from historical information without new segmentation of threads. This will be done in future studies.

7. Related Work

In the cloud environment, GPUs are shared across multiple VMs for HPC of VMs, and studies on the scheduling of GPUs have been conducted continuously. GPUvm, mediated pass-through, and I/O para-virtualization are GPU full virtualization solutions. These solutions modify the hypervisor and GPU driver to multiplex the GPU. This allows

GPUs to be shared among multiple VMs running simultaneously on the host machine and scheduling GPU operations to enable the VM to access the GPU. The solutions for GPU full virtualization provide the full functionality of the GPU to the VMs, enable GPU sharing among multiple VMs, and schedule GPU operations. These GPU full virtualization technologies generally use FIFO scheduling because they process GPU operations requested in the VM in units of instructions. However, the scheduling of the FIFO scheme may cause a situation in which a VM requesting a new task is not scheduled in a timely manner because a GPU operation is waiting in the scheduling queue. Moreover, in a cloud environment, VMs must use computing resources equally, considering resource usage time. Therefore, the method of simply processing work requests in order is limited in supporting the even use of VMs resources.

vCUDA [29], rCUDA [30] and virtio-CL [26] are RPC-based GPU virtualization solutions in a cloud environment. RPC-based GPU virtualization technology has a server–client structure composed of GPGPU task requesters and responders and redirects GPGPU commands to actual GPUs through the modified GPGPU API to deliver the commands. When the process is completed, the responsor returns the result of the task to the corresponding VM. Because the GPU API must be modified, it is limited to providing the full functionality of the GPU to the VM. However, it is easy to apply it to various virtualization platforms and operating systems because there is no need to modify the OS or hypervisor.

RPC-based GPU virtualization technology handles GPGPU API that simultaneously makes requests from multiple VMs. Therefore, the proposed technique uses a dedicated scheduling scheme for GPGPU tasks to avoid conflicts and process them in order. Existing RPC-based GPU virtualization technologies use FIFO-based scheduling techniques, such as the above-described GPU full virtualization technologies, and do not perform scheduling considering the GPU usage time of the VM. The FIFO-based GPU scheduler is a simple, reliable, and widely employed scheduling scheme that schedules tasks based on the order of requests for tasks from multiple VMs and returns the results according to the order of task requests. However, in a cloud environment, as many users share computing resources they pay for, it is important to use limited resources uniformly. Therefore, a scheduling technique that considers the GPU usage time is necessary.

Various studies have been conducted to address the shortcomings of existing FIFO-based GPU scheduling. GPES [13] supports preoccupancy execution which is not supported by GPU, and supports real-time GPGPU tasks through the slicing of the general GPGPU task. GPES focuses on minimizing the time when high-priority GPU tasks are blocked by low-priority GPU tasks when priorities exist between GPU tasks. This approach is not suitable for environments where multiple users share GPUs equally. However, the techniques proposed in this paper prevent certain virtual machines from monopolizing GPUs for a long time by dividing the GPGPU task into multiple thread groups and executing the group of divided threads based on each user's GPU usage time. Our work supports each virtual machine to use GPUs for an even time because it executes a group of divided threads based on GPU usage time. In addition, the GPU usage time-based scheduling method proposed in this paper can be extended to a scheduling method with GPU usage priorities for each virtual machine by recording less GPU usage time for high-priority virtual machines than their actual GPU usage time. Fine-grained scheduling [40] minimizes the idle time of GPU tasks through event-based scheduling. FairGV [8] adjusts each VM to use the GPU equally based on the degree of GPU access for each VM. The study focuses on minimizing the idle time for GPU devices. In addition, mixed workload situations where virtual machines perform different-scale tasks and GPU usage times for each virtual machine are not considered. However, in our work, we prevent GPU long-term occupancy due to GPGPU tasks with long running times and reduce the latency of GPGPU tasks.

User-mode CPU–GPU scheduling [41] optimizes CPU utilization while maintaining the QoS of GPU tasks based on QoS feedback from each VM. This work monitors the QoS of the workload and uses the slip function to limit the performance of the workload with more CPU or GPU access than other workloads. This work allocates as little resources as

possible, focusing on reducing the performance deviation of all workloads and maintaining a certain level of QoS. This allows minimal resource usage, but it places constraints on completing tasks faster because more resources are unavailable to running workloads. However, our work allows GPGPU task completion quickly while maintaining consistency in performance, as it executes as many threads of GPGPU tasks as possible as GPU resources allow. Flep [42] proposed a preemptive scheduling technique that can solve the priority reversal problem in multi-tasking environments caused by the lack of preemption support on existing GPUs and efficiently schedule them according to the priorities of GPU tasks. This work predicts the running time of tasks with the same priority through separate offline-based performance metric measurements, preventing short-running GPGPU tasks from waiting for a long time by long-running GPGPU tasks. Because this method uses performance modeling to predict priority and task execution time, the GPU resource usage of each task is not considered. However, our work schedules tasks based on GPU usage time to weigh the thread execution scale of each virtual machine, which can reduce unnecessary latency of GPGPU tasks with short execution time while equally scaling GPU resource usage of each virtual machine. The fair resource sharing technique [43] proposed a Just-In-Time compiler capable of resource sharing control and scheduling to improve resource fairness between users in an environment where multiple users share GPUs. This work is to configure equally the scale kernel execution when multiple workloads share GPUs, and it does not take into account GPU usage time for each workload. However, our work supports further execution of GPGPU task threads on virtual machines with less GPU usage time by scaling the execution of threads and stopping the thread execution of long-running GPGPU tasks.

Virtual multi-channel GPU pair scheduling [18] proposes a virtual multi-channel GPU virtualization technology that fairly allocates GPU resources to each VM, enabling each VM to use the GPU equally. The fine-grained sharing on GPUs technique [44] proposed QoS management techniques to improve support for service quality in environments where multiple GPUs operate. The two GPU task scheduling approaches described earlier require a modification of system components, such as hypervisors or GPU drivers. This makes portability very low and difficult to apply to other virtualization systems. However, the techniques proposed in this paper are easy to apply to other virtualization systems because they do not require modification of system components.

In this paper, we solve the problem of VMs not using the GPU fairly because they do not consider the GPU usage time in the existing GPGPU task-scheduling techniques in cloud environments. In addition, the purpose of the proposed scheduling scheme is to minimize the waiting time of other VMs caused by the long-running GPGPU task. Accordingly, the scheduling latency of the GPGPU tasks is minimized by determining the work order based on the GPU usage time and by preventing long GPU occupation owing to the long-running GPGPU task by dividing the GPGPU task. Furthermore, our approach does not require modifications to system components such as hypervisors, GPU drivers, and OSs, so it can be extended for application to various platforms.

8. Conclusions

This paper proposed the GPGPU task scheduling techniques in RPC-based GPU virtualization environments where multiple VMs share a single GPU. In FIFO-based scheduling environments, all of the threads for processing the GPGPU task are usually called and executed at once. The GPGPU task cannot arbitrarily generate context-switching from the outside; thus, long-running GPGPU tasks long time monopolize GPU, increasing the wait time of other VM's GPGPU tasks. In a typical computing environment, a single user monopolizes GPUs, but in a cloud environment where several VMs or users share computing resources, the cloud providers must manage to ensure that multiple VMs use computing resources evenly. However, as the experiments in the previous section showed, when multiple GPGPU tasks were run simultaneously in the FIFO-based scheduling, each

VM was unable to use GPUs evenly, resulting in no performance consistency for each GPGPU task and significant performance deviations in repeated experiments.

The GPGPU task scheduling scheme proposed in this paper executes the group of divided threads of the GPGPU task based on the virtual machine's GPU usage time. Each VM executed only one group of divided threads at a time, indirectly limiting GPU usage. Our approach prevents GPU monopoly by a particular VM, increases the opportunity for each VM to use the GPU, and enables each VM to use the GPU simultaneously by running one group of divided threads of each VM at the GPU. To adjust the thread group's execution time based on the GPU usage time, we measured the GPU usage time considering the number of threads executed through the GPU usage time accounter and scheduled GPGPU tasks using the GPU usage time. Our work achieved performance consistency of the GPGPU tasks by solves the problem of the long-time GPU monopoly of long-running GPGPU tasks and the performance deviation between multiple VMs.

In this paper, experiments confirmed that each GPGPU task's performance deviation could be effectively reduced in an environment when long-running GPGPU tasks and short-running GPGPU tasks are executed at the same time. However, in an environment where GPGPU tasks of the same size are executed, as describe the previous section, the proposed GPGPU task scheduling technique's effectiveness was relatively low. In short-running GPGPU tasks, the overhead caused by thread division processing was increased. As the experiment shows, some overhead is caused by the process of partitioning threads in the GPGPU task scheduling phase. This overhead does not significantly affect performance compared to the performance deviations of the FIFO-based scheduling environment but slightly increases the average run time of GPGPU tasks. As described earlier, no information (such as past records) is used to divide the threads of the GPGPU task. Therefore, in future research, we aim to model the performance table to execute a group of threads of a predefined size when the same task is executed again. Therefore, we aim to apply techniques to future research that will model performance tables so that they can utilize predefined information without having to record and analyze GPGPU task execution information for VMs and determine the size of the GPGPU task that they run on each VM. Our experiment also describes how GPU memory over-usage is affecting the performance of GPU tasks. Our current work focuses on the kernel function of the GPGPU task. However, by analyzing GPU memory competition and GPGPU task performance in the future, we will develop GPU memory sharing techniques and integrate them with GPGPU task scheduling technology.

Author Contributions: Conceptualization, J.K. and H.Y.; software, J.K.; validation, J.K. and H.Y.; formal analysis, J.K.; investigation, J.K. and H.Y.; resources, J.K.; data curation, J.K.; writing—original draft preparation, J.K. and H.Y.; writing—review and editing, J.K. and H.Y.; visualization, J.K.; supervision, H.Y.; project administration, H.Y.; funding acquisition, H.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2018-0-01405) supervised by the IITP(Institute for Information and Communications Technology Planning and Evaluation).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Enabling Intel Virtualization Technology Features and Benefits. Available online: https://images.nvidia.com/content/pdf/grid/whitepaper/NVIDIA-GRID-WHITEPAPER-vGPU-Delivering-Scalable-Graphics-Rich-Virtual-Desktops.pdf (accessed on 29 January 2021).
2. Credit Scheduler. Available online: https://wiki.xen.org/wiki/CreditScheduler (accessed on 29 January 2021).

3. Completely Fair Scheduler. Available online: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html (accessed on 25 February 2021).
4. Amazon EC2 P4d Instances. Available online: https://aws.amazon.com/ec2/instance-types/p4/?nc1=h_ls (accessed on 29 January 2021).
5. Microsoft Azure GPU Optimized Virtual Machine Sizes. Available online: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu (accessed on 29 January 2021).
6. Google Cloud GPUs. Available online: https://cloud.google.com/gpu (accessed on 29 January 2021).
7. Tencent GPU Server. Available online: https://cloud.tencent.com/product/gpu (accessed on 29 January 2021).
8. Hong, C.H.; Spence, I.; Nikolopoulos, D.S. FairGV: Fair and fast GPU virtualization. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 3472–3485. [CrossRef]
9. Nvidia Grid vGPU: Delivering Scalable Graphics-Rich Virtual Desktops. Available online: https://images.nvidia.com/content/pdf/grid/whitepaper/NVIDIA-GRID-WHITEPAPER-vGPU-Delivering-Scalable-Graphics-Rich-Virtual-Desktops.pdf (accessed on 29 January 2021).
10. AMD Radeon Pro. Available online: https://www.amd.com/en/products/server-accelerators/amd-radeon-pro-v520 (accessed on 29 January 2021).
11. OpenCL: Open Computing Language. Available online: https://www.khronos.org/opencl/ (accessed on 29 January 2021).
12. CUDA: Compute Unified Device Architecture. Available online: https://developer.nvidia.com/cuda-zone (accessed on 29 January 2021).
13. Zhou, H.; Tong, G.; Liu, C. GPES: A preemptive execution system for GPGPU computing. In Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA, 13–16 April 2015.
14. Long, X.; Gong, X.; Liu, Y.; Que, X.; Wang, W. Toward OS-Level and Device-Level Cooperative Scheduling for Multitasking GPUs. *IEEE Access* **2017**, *8*, 65711–65725. [CrossRef]
15. AMD Accelerated Parallel Processing OpenCL Programming Guide. Available online: https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (accessed on 29 January 2021).
16. NVIDIA CUDA C Programming Guid. Available online: http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf (accessed on 29 January 2021).
17. Goswami, A.; Young, J.; Schwan, K.; Farooqui, N.; Gavrilovska, A.; Wolf, M.; Eisenhauer, G. GPUShare: Fair-sharing middleware for GPU clouds. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, USA, 23–37 May 2016; pp. 1769–1776.
18. Tan, H.; Tan, Y.; He, X.; Li, K.; Li, K. A virtual multi-channel GPU fair scheduling method for virtual machines. *IEEE Trans. Parallel Distrib. Syst.* **2018**, *30*, 257–270. [CrossRef]
19. Sorensen, T.; Evrard, H.; Donaldson, A.F. Cooperative kernels: GPU multitasking for blocking algorithms. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017.
20. Tanasic, I.; Gelado, I.; Cabezas, J.; Ramírez, A.; Navarro, N.; Valero, M. Enabling preemptive multiprogramming on GPUs. *ACM Sigarch Comput. Archit. News* **2014**, *42*, 193–204. [CrossRef]
21. Suzuki, Y.; Kato, S.; Yamada, H.; Kono, K. GPUvm: Why not virtualizing GPUs at thehypervisor? In Proceedings of the USENIX Annual Technical Conference, Philadelphia, PA, USA, 19–20 June 2014; pp. 109–120.
22. Tian, K.; Dong, Y.; Cowperthwaite, D. A Full GPU Virtualization Solutionwith Mediated Pass-Through. In Proceedings of the USENIX Annual Technical Conference, Philadelphia, PA, USA, 19–20 June 2014; pp. 121–132.
23. Amiri Sani, A.; Boos, K.; Qin, S.; Zhong, L. I/O paravirtualization at the device file boundary. *ACM Sigarch Comput. Archit. News* **2014**, *42*, 319–332. [CrossRef]
24. Lin, Y.; Lin, C.; Lee, C.; Chung, Y. qCUDA: GPGPU Virtualization for High Bandwidth Efficiency. In Proceedings of the 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Sydney, Australia, 11–13 December 2019.
25. Giunta, G.; Montella, R.; Agrillo, G.; Coviello, G. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In Proceedings of the Euro-Par 2010-Parallel Processing, Ischia, Italy, 31 August–3 September 2010.
26. Tien, T.; You, Y. Enabling OpenCL support for GPGPU in Kernel-based Virtual Machine. *Softw. Pract. Exp.* **2014**, *44*, 483–510. [CrossRef]
27. Gupta, V.; Gavrilovska, A.; Schwan, K.; Kharche, H.; Tolia, N.; Talwar, V.; Ranganathan, P. GViM: GPU-accelerated virtual machines. In Proceedings of the 3rd ACM Workshop on System-Level Virtualization for High Performance Computing (HPCVirt '09), Nuremburg, Germany, 31 March 2009.
28. Lagar-Cavilla, H.A.; Tolia, N.; Satyanarayanan, M.; de Lara, E. VMM-independent graphics acceleration. In Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07), San Diego, CA, USA, 13–15 June 2007.
29. Shi, L.; Chen, H.; Sun, J.; Li, K. vCUDA: GPU-accelerated highperformance computingin virtual machines. *IEEE Trans. Comput.* **2014**, *61*, 804–816. [CrossRef]
30. Duato, J.; Pena, A.J.; Silla, F.; Mayo, R. Quintana-Ort ES. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In Proceedings of the High Performance Computing and Simulation (HPCS), Caen, France, 28 June–2 July 2010; pp. 224–231.
31. Barham, P.; Dragovic, B.; Fraser, K. Xen and the art of virtualization. *ACM Sigops Oper. Syst. Rev.* **2003**, *37*, 164–177. [CrossRef]
32. Xen. Available online: https://xenproject.org/ (accessed on 29 January 2021).

33. VGA Passthrough. Available online: https://wiki.xen.org/wiki/Xen_VGA_Passthrough (accessed on 29 January 2021).
34. Intel Virtualization Technology for Directed I/O Architecture Specification. Available online: https://software.intel.com/content/www/us/en/develop/download/intel-virtualization-technology-for-directed-io-architecture-specification.html (accessed on 29 January 2021).
35. KVM. Available online: https://www.linux-kvm.org/ (accessed on 25 February 2021).
36. OpenStack. Available online: https://www.openstack.org/ (accessed on 25 February 2021).
37. NVIDIA Management Library. Available online: https://docs.nvidia.com/deploy/nvml-api/index.html (accessed on 25 February 2021).
38. AMD GPU Performance API. Available online: http://developer.amd.com/wordpress/media/2013/12/GPUPerfAPI-UserGuide-2-15.pdf (accessed on 25 February 2021).
39. Gleeson, J.; Kats, D.; Mei, C.; de Lara, E. Crane: Fast and migratable GPU passthrough for OpenCL applications. In Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR '17), Haifa, Israel, 22–24 May 2017.
40. Zhao, X.; Yao, J.; Gao, P.; Guan, H. Efficient sharing and fine-grained scheduling of virtualized GPU resources. In Proceedings of the 2018 IEEE 38th International Conference on Distributed Computing Systems(ICDCS), Vienna, Austria, 2–6 July 2018.
41. Wang, B.; Ma, R.; Qi, Z.; Yao, J.; Guan, H. A user mode CPU–GPU scheduling framework for hybrid workloads. *Future Gener. Comput. Syst.* **2016**, *63*, 25–36. [CrossRef]
42. Wu, B.; Liu, X.; Zhou, X.; Jiang, C. Flep: Enabling flexible and efficient preemption on GPUs. *ACM Sigplan Not.* **2017**, *52*, 483–496. [CrossRef]
43. Margiolas, C.; O'Boyle, M.F.P. Portable and transparent software managed scheduling on accelerators for fair resource sharing. In Proceedings of the 2016 International Symposium on Code Generation and Optimization, Barcelona, Spain, 12–18 March 2016.
44. Wang, Z.; Yang, J.; Melhem, R.G.; Childers, B.R.; Zhang, Y.; Guo, M. Quality of service support for fine-grained sharing on GPUs. In Proceedings of the 44th Annual International Symposium on Computer Architecture, Toronto, ON, Canada, 24–28 June 2017.