Received: 29 May 2019 | Revised: 13 November 2019 | Accepted: 9 January 2020

DOI: 10.1002/spe.2801

SPECIAL ISSUE PAPER

Partial migration technique for GPGPU tasks to Prevent GPU Memory Starvation in RPC-based GPU Virtualization

JiHun Kang[1] | **JongBeom Lim**[2] | **HeonChang Yu**[1]

[1]Department of Computer Science and Engineering, Korea University, Seoul, Republic of Korea

[2]Department of Game and Multimedia Engineering, Korea Polytechnic University, Gyeonggi-do, Siheung-si, Republic of Korea

Correspondence
HeonChang Yu, Department of Computer Science and Engineering, Korea University, 145 Anam-ro, Seongbuk-gu, Seoul, Republic of Korea.
Email: yuhc@korea.ac.kr

Funding information
Institute for Information & communications Technology Promotion, 2018-0-00480; MSIT, IITP-2018-0-01405

Summary
Graphics processing unit (GPU) virtualization technology enables a single GPU to be shared among multiple virtual machines (VMs), thereby allowing multiple VMs to perform GPU operations simultaneously with a single GPU. Because GPUs exhibit lower resource scalability than central processing units (CPUs), memory, and storage, many VMs encounter resource shortages while running GPU operations concurrently, implying that the VM performing the GPU operation must wait to use the GPU. In this paper, we propose a partial migration technique for general-purpose graphics processing unit (GPGPU) tasks to prevent the GPU resource shortage in a remote procedure call-based GPU virtualization environment. The proposed method allows a GPGPU task to be migrated to another physical server's GPU based on the available resources of the target's GPU device, thereby reducing the wait time of the VM to use the GPU. With this approach, we prevent resource shortages and minimize performance degradation for GPGPU operations running on multiple VMs. Our proposed method can prevent GPU memory shortage, improve GPGPU task performance by up to 14%, and improve GPU computational performance by up to 82%. In addition, experiments show that the migration of GPGPU tasks minimizes the impact on other VMs.

KEYWORDS
cloud computing, GPU Virtualization, resource management, task migration

1 | INTRODUCTION

With graphics processing unit (GPU) virtualization technology,[1,2] a single GPU can be shared by multiple virtual machines (VMs). Processing GPU operations running on multiple VMs with a single GPU can improve the utilization of GPU resources, and cloud providers can benefit from low hardware costs since they do not have to provide individual physical GPUs for every single VM.

Although GPU sharing through GPU virtualization can provide many benefits in terms of resource utilization and cost, GPUs exhibit resource shortages more frequently than other computing resources (eg, central processing units [CPUs]) because they have limited resources. In a GPU-only server that accommodates many GPUs, many VMs can share multiple GPUs. However, since a typical general-purpose server has a limited number of GPUs installed, finding an available GPU is difficult, especially when multiple VMs are performing GPU operations simultaneously. The GPU may not always be utilized and is used only when performing GPU operations. Since the data used for GPU operations are stored in

GPU-dedicated memory and reside there until the memory is released according to the operation sequence of the GPU program, GPU cores can only be provided to other VM when the cores are idle. However, because a cloud administrator cannot release a user's GPU memory at his own discretion, the GPU cannot be provided to the required VM at a proper time because of the unavailability of GPU memory procurement technology (eg, data dumping or virtual GPU memory). In such a cloud computing environment, a resource management technique is needed to free up the available GPU resources and prevent the GPU resources shortage at VMs.

In a cloud environment, multiple users share physical server resources, and the cloud provider cannot arbitrarily reallocate resources once they have already been allocated to a VM. It is impossible for a cloud administrator to arbitrarily reallocate resources since the resources must be guaranteed regardless of the actual usage of the resources allocated to the VM; namely, because the VM user pays for the cost of the allocated resources even if the resource utilization is low, the cloud provider still should not adjust the resources allocation for the VM. Because the resources of a single physical server are limited, server resource shortage is inevitable in a cloud computing environment without a dedicated resource management technique. As the usage of physical resources can be changed frequently in a cloud environment, VM migration and distributed resource concentration techniques are used to replace VM's physical servers depending on the policy.[3]

Migration is used for various purposes such as resource starvation prevention, fault tolerance, and load balancing because it can actively cope with the resource fluctuations of cloud servers. In a cloud environment, migration typically targets VMs. Previous researches have resolved resource starvation prevention, fault tolerance, and load balancing by migrating VMs to other physical servers while minimizing the downtime in the VMs. However, because VM migration requires transferring the image of the VM to another physical node, it uses significant network bandwidth and momentarily stops working during the migration. The network usage due to VM migration affects user's experiences of VMs because the VM user can use the limited network resources shared by multiple VMs. In addition, CPUs are shared by VMs on a time-sharing basis, thereby enabling context switching at any time, while the GPUs must be stopped working for migration. However, in a typical cloud environment, there is no technology for GPUs to resume general-purpose graphics processing unit (GPGPU)[4,5] tasks after the VM migration.

The GPU task migration technique that is used to reduce a VM's migration overhead migrates the entire GPU task to another node if the available GPU resources cannot accommodate the GPU task. However, the full migration method moves the task to another node when it cannot accommodate the entire GPU task although desired GPU resources are available. This causes the available GPU resources to be idle, thereby reducing the GPU resource utilization. This problem worsens with the increase in the size of newly hosted GPU workloads. For example, if the available GPU memory is 100 MB, GPU tasks with less than 100 MB GPU memory usage can be hosted without any problems. However, GPU tasks with GPU memory usage greater than 100 MB must be migrated to other nodes. If a GPU task requests 110 MB GPU memory, a resource shortage occurs. In an environment where GPU tasks run concurrently, the GPU memory capacity should be large enough to accommodate the amount of data requested by a GPU task. In such environment, GPU task migration is likely to occur due to insufficient GPU memory. Additionally, the remaining resources of the GPU remain idle until a workload of appropriate size is requested.

The CPU and memory can be shared by many VMs and can logically divide computing resources (eg, cores and memory regions). Thereby isolating the VM resources, so it is possible to easily identify the resource area that should not be accessed when a new VM is placed or migrated there. In addition, the resource usage of the physical server can be checked by comparing the total and allocated resource amounts. However, the GPU cannot specify the core to be used by the VM or isolate it to a specific area, and neither the GPU usage time nor the capacity of the GPU resources to be used can be defined without specifying the VM user. Therefore, it is very difficult to know when to use the GPU and how many times to use it. This problem makes it difficult to select the target physical node because the node cannot tell the lack of GPU resources.

In this paper, we propose a method of partially migrating GPU tasks to prevent GPU resource shortage in a physical server. Our approach is much more efficient than migrating the entire VM because it only migrates some of the GPGPU's work to avoid downtime and subsequent restart overhead due to VM migration. Our approach involves migrating the partial GPGPU work to a destination server whose resource is enough while maintaining the VM intact on the source server when the GPU resources are low. Our method uses the server-client architecture of the remote procedure call (RPC)-based GPU virtualization technique. When the GPGPU task is completely migrated to the destination server, the operation result is returned to the corresponding VM, and the result data processed by the destination server are combined for the VM (cf, Section 4 for details).

We consider the size of the data to be migrated and the number of VMs currently sharing the GPU to avoid GPU memory shortages. We extend our previous research, RPC-based GPU virtualization technology, on the virtualization platform using Xen[6] to implement the GPU task migration technique.

The main contributions of this paper are as follows:

- We propose a method of partially migrating GPU tasks, which can prevent the degradation of the GPU work performance of a VM (owing to a lack of GPU resources) while maintaining a constant VM performance.
- Our approach is to migrate only part of the GPU work while maintaining the GPU work of the VM hosting the GPU work; thus, less network overhead is introduced than a method to migrate the whole VM and there is no work stoppage for the migration.
- In our partial migration technique for GPGPU tasks, if the remaining GPU memory cannot accommodate all the data for a GPGPU task, a portion of the task for an acceptable size runs on the source node, and the remaining tasks are migrated to the destination node. This approach results in higher utilization of the GPU memory compared to a full migration of the GPGPU operations.
- Most of the existing tasks and VM migration methods for resolving resource shortages on the physical server migrate the entire task or VM upon the detection of resource shortage. On the other hand, our approach avoids running out of GPU resources by retrieving the GPU memory requirements of the currently running GPGPU tasks before new GPGPU tasks are scheduled.
- Our GPU Task migration method migrates a part without stopping the GPU task, and divides it into source and destination nodes according to the available GPU memory capacity to minimize the latency of the entire task.
- Our approach provides transparency when migrating GPU tasks so that VM users do not have to manage the size of the GPU tasks to be migrated during the programming phase.

Section 2 describes the underlying technologies for our GPU task migration technique, including our previous work, and Section 3 details the performance degradation analysis due to the resource shortage and the research motivation of the proposed technique in the RPC-based GPU virtualization environment. Section 4 describes in detail the techniques we propose. Section 5 explains the experiments we performed, and Section 6 explains related works. Finally, Section 7 summarizes the main findings of the work.

2 | BACKGROUND

2.1 | GPGPU programming model

The GPGPU programming model[7,8] is a technology that uses the GPU as a general-purpose computing device. In the GPGPU programming model, the GPU performs massively parallel operations with thousands of compute cores, achieving a higher level of performance than a CPU. The major differences between the GPGPU programming model and the general CPU-based one are twofold: (a) the data input and output method to be used for the operation and (b) the manner of performing the operation. In a typical CPU-based task, the data to be used for the task are usually stored in the main memory. For GPU-based calculations, the data are stored in the GPU memory, a dedicated memory inside the GPU. However, in the GPGPU programming model, a programmer cannot directly write data to the GPU memory when entering data into the GPU memory. Rather, the programmer manages the data used in the operation by inputting the data into the main memory and copying them to the GPU memory. Because GPU memory cannot be directly accessed when reading the GPU operation result data, the result data in the GPU memory must be copied to the main memory before the data can be read. Furthermore, although most of the GPGPU programming model handles most of the work on the GPU, the GPU structurally acts as a coprocessor for the CPU. Because the programmer cannot execute commands or input data directly into GPU devices, a GPU is not a computing-type device that performs tasks independently but a heterogeneous one that performs tasks together with a CPU. This, the GPGPU programming model is structurally separated into a host area for performing operations using the CPU and main memory, and a device area for performing operations using the GPU core and GPU memory. Figure 1 depicts a flow chart of the GPU operations in the GPGPU programming model.

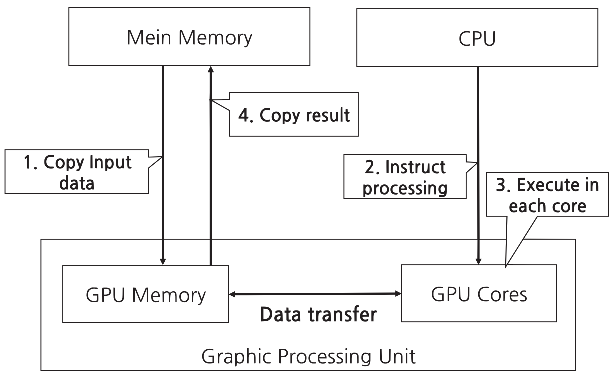


FIGURE 1 General-purpose graphics processing unit programming model

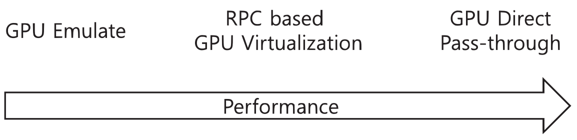


FIGURE 2 Graphics processing unit virtualization spectrum

2.2 | RPC-based GPU virtualization

In a cloud environment, a variety of methods are used to support high-performance computing for VMs. GPU virtualization technology is one that allows VMs to access GPU devices to perform high-performance GPU operations. In the cloud environment, there are various methods of supporting GPUs in VMs: (a) GPU emulation, (b) full GPU virtualization, and (c) RPC-based GPU virtualization.

The GPU emulation method emulates the tasks the GPU must perform on the CPU. This approach does not provide a GPU to a VM, but it mainly supports the GPU operations without a GPU, so no high performance is expected. Full GPU virtualization provides full GPU functionality to a VM, thereby enabling the VM to perform a variety of GPU tasks including 3D image rendering and GPGPU operations. Although full GPU virtualization[9-11] provides the full functionality of the GPU to the VM, the host and guest operating system (OS), hypervisor, and GPU drivers must be modified because full GPU virtualization is system-dependent and there are a limited number of virtualization platforms and OSs that can be applied. Finally, RPC-based GPU virtualization[12-14] does not directly virtualize a GPU; instead it virtualizes the GPU programming application programming interface (API). RPC-based GPU virtualization uses a server-client architecture wherein one VM requests the GPU operations, and other processes the GPU work for the other VMs and return the result. The VM that performs the GPU operations does not own the GPU and uses the modified GPU programming API to redirect the GPU to the assigned VM rather than delivering the work and data to the GPU driver. Therefore, the VM that owns the actual GPU performs the GPU operations and returns the results. To enable the RPC-based virtualization environment, a cloud provider must modify the GPU programming API to send information such as the command number, execution task function, function parameters, and the data to be executed in the VM that performs the GPU work through RPC communication. Figure 2 shows the GPU virtualization spectrum and Figure 3 shows the basic structure of the RPC-based GPU virtualization environment.

RPC-based GPU virtualization has the disadvantage of modifying the GPU programming API, so it must modify all the necessary GPU programming APIs. However, because it is not dependent on OSs or hypervisors, it is easily applied to a container or other hypervisors; RPC-based GPU virtualization is technologically less complex than full GPU virtualization since the limited numbers of APIs are required for the modification. Moreover, the machine acting as the RPC server has direct access to the GPU, thereby resulting in better performance than emulation.

2.3 | Our GPU virtualization

Our proposed GPU task migration technique works based on our previous research on the RPC-based GPU virtualization environment. Our GPU virtualization environment is similar to the environment used for RPC-based GPU virtualization. Each user VM (RPC client) executes the tasks through the modified GPGPU API, and the modified GPGPU API sends

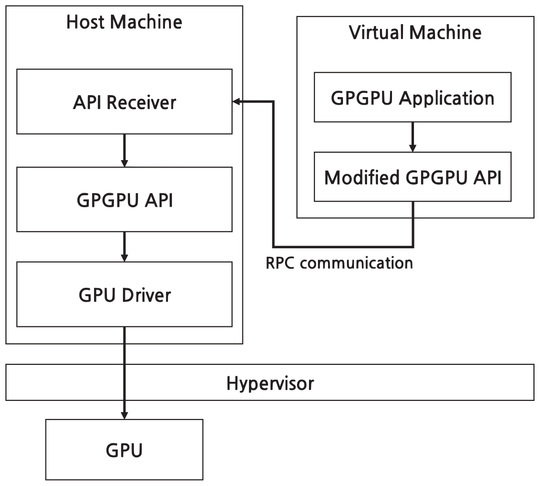


FIGURE 3 Basic structure of remote procedure call-based virtualization environment

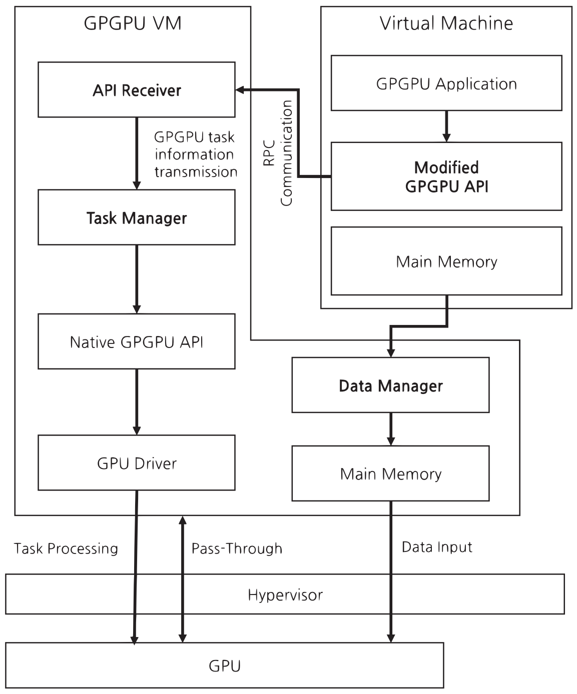


FIGURE 4 Our remote procedure call-based graphics processing unit virtualization

the VM identifier (ID), instruction number, parameters, and input data to the RPC server that performs the GPU operations on behalf of the user VM. The VM that performs the GPU operations has an RPC-based GPU virtualization system structure that processes the GPGPU API and returns the results to the corresponding VM. Our previous work divided the GPGPU work of the VMs to prevent the GPU from monopolizing the long-running GPU work. Because GPGPU operations are not context switchable during execution, the long-running GPGPU tasks occupy the GPU, and the other VMs must wait until the work of the VM occupying the GPU has finished. By dividing the GPGPU work of VMs, it is possible to reduce the amount of time that the VM occupies the GPU and give more chances for other VMs to occupy the GPU, thereby preventing the long-term GPU occupation, which was not guaranteed in the existing RPC-based GPU virtualization environment. Figure 4 shows the structure of the RPC-based GPU virtualization environment used for the proposed GPU task migration.

In the RPC-based GPU virtualization environment, an RPC server is required to handle the GPU tasks of the user VMs. In our RPC-based GPU virtualization environment, we use a separate VM to handle the GPU tasks of the user VMs to configure the GPU virtualization. We term the VM that handles the GPU tasks of users VMs as GPGPUvm. In the RPC-based GPU virtualization environment, no GPU is allocated to the user VMs. The GPUs in the physical server are exclusive to GPGPUvm. Moreover, GPGPUvm is also a VM; therefore, it accesses the GPU in a pass-through[15] manner.

Additionally, when a GPGPU task is requested from a user VM, GPGPUvm receives information related to each VM's GPU tasks, and then executes a process mapped to each VM to manage the information and data of the task until the task is completed. Furthermore, it maintains the IP addresses of VMs and identifies a VM and sends the corresponding result to it.

Our method of configuring an RPC server as a VM for processing the GPU tasks of a user VM can adjust the allocation of the CPU and the main memory of the GPGPUvm according to the number of VMs sharing the GPU and the size of the task. Moreover, VMs can share GPUs in a single server without requiring a separate GPU server to handle the GPU tasks for user VMs. Our method exchanges the data and API information about the GPGPU operations through internal RPC communication between VMs in a single server, which has the advantage of reduced network overhead than using an independent server externally.

3 | MOTIVATION

This section analyzes the performance of GPU operations based on the number of VMs running GPU workloads to determine the impact of a lack of GPU resources on the performance of the GPU operations running on each VM. The GPU does not support resource isolation and fairness resource sharing for multiple VMs. Cloud-exclusive GPUs that support virtualization also supports performance and resource isolation so that all the VMs can share a certain amount of GPU resources, but they do not flexibly cope with the VM's resource requirements because they allocate the GPU resources statically.

A typical GPU does not have features such as garbage collection that automatically releases unused GPU memory space. Therefore, when multiple GPU tasks are running simultaneously, the subsequent GPU operations must wait until there is GPU memory available for normal execution when the available GPU memory is low. This is causing an unfairness problem and performance degradation and limits the number of GPU operations that can be hosted simultaneously on the physical server. In this experiment, we analyze the performance of GPGPU tasks executed in VMs when the resource requirement of the running GPU tasks exceeds the available resources of the GPU. The GPGPU tasks used in the experiment are listed in Table 1.

In this experiment, we used one GPGPUvm and eight user VMs to perform GPU operations. On receiving the user VM's requests for GPGPU tasks, the GPGPUvm processes the APIs received through RPC communication in an ordered manner. VMs that perform GPU operations process matrix multiplication of various sizes. We measure the performance of the GPGPU operations in cases of sufficient and insufficient GPU memory, and determine the impact of GPU memory shortages on the performance of GPGPU operations performed by the VMs. Each VM performs 9000×9000, $10\,000 \times 10\,000$, $11\,000 \times 11\,000$, $12\,000 \times 12\,000$, $13\,000 \times 13\,000$ and $14\,000 \times 14\,000$ matrix multiplications, in the increment of the number of VMs from one to eight. Through this experiment, we can figure out the performance of the VMs in the cases of sufficient and insufficient GPU memory. The experimental results are depicted in Figure 5.

As shown by the experimental results, the more the GPU runs out of memory, the worse the performance of the GPGPU operations. Specifically, the execution time of the kernel function that performs the actual GPU operation does not degrade the performance suddenly, even if the number of VMs and the size of the matrix increase. However, as the number of concurrently executed VMs increases, the data input/output time increases significantly. As the matrix size and the number of VMs of the matrix multiplication performed in the VM increase, both the data input/output time and

TABLE 1 General-purpose graphics processing unit tasks and individual execution times used for performance evaluation

Matrix size	Data transfer	Data input	Kernel processing	Result output	Result return	Total
$9,000 \times 9,000$	4.1200	0.4370	0.0310	0.1560	2.0582	6.8022
$10\,000 \times 10\,000$	5.2284	0.4840	0.0310	0.3430	2.6142	8.7006
$11\,000 \times 11\,000$	6.3265	0.5770	0.0310	0.2660	3.1632	10.3637
$12\,000 \times 12\,000$	7.5290	0.5920	0.0320	0.3120	3.7645	12.2295
$13\,000 \times 13\,000$	8.8361	0.6390	0.0320	0.3740	4.4181	14.2992
$14\,000 \times 14\,000$	10.2478	0.7180	0.0310	0.4060	5.1239	16.5267

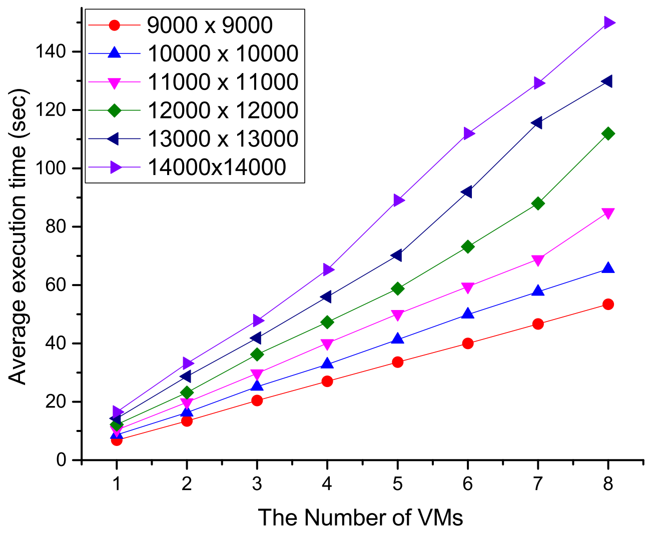


FIGURE 5 The average performance of general-purpose graphics processing unit tasks according to the number of virtual machines performing matrix multiplication [Colour figure can be viewed at wileyonlinelibrary.com]

GPU processing time increase. In particular, in the case of data input/output, the performance of processing decreases in the order of tens to hundreds of times.

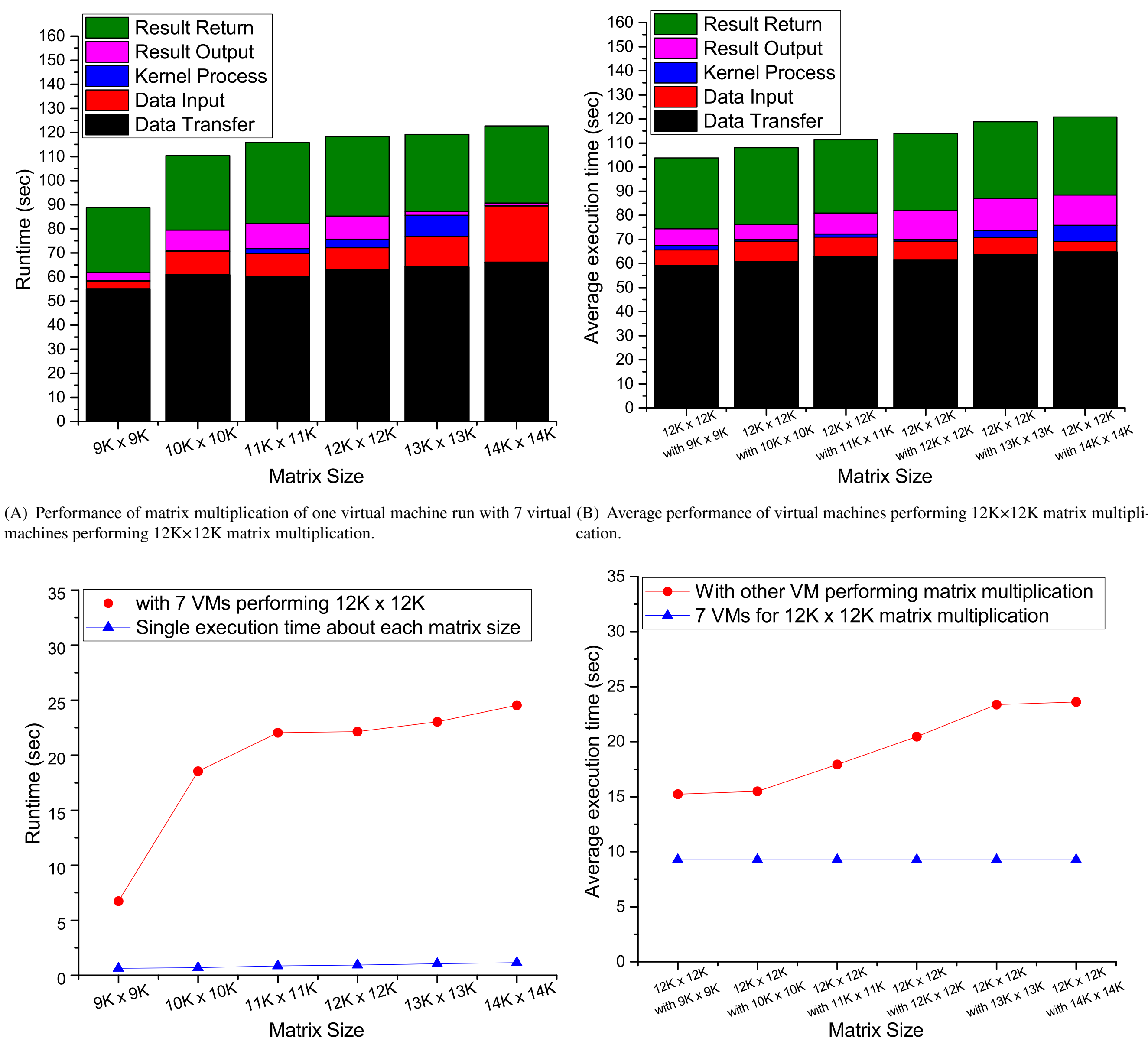
The second experiment was conducted in the same environment as the previous one. In this experiment, the VMs are divided into two groups. The first group comprises seven VMs that preoccupy the GPU memory. The second group comprises one VM that measures the performance of newly requested GPGPU tasks using various sizes of matrices in the case of lack of GPU memory. The first group performs 12 000 × 12 000 matrix multiplication simultaneously, whereas the second group performs matrix multiplication after 1 second. This experiment shows how the lack of GPU memory affects the performance of newly executing GPGPU tasks.

In Figure 6, data transfer represents the time to transfer data for a GPGPU task from a user VM to GPGPUvm. The data input, kernel process, and result output show the time to input data from main memory to GPU memory, process the kernel function, and copy the result to main memory in a typical GPGPU task processing flow, receptively. Finally, the result return is the time to return the GPGPU task result from GPGPUvm to the user VM.

As shown in Figure 6, the performance of the newly requested GPGPU task is not very good when the GPU memory is low. Figure 6A,B shows the performance when processing a GPGPU task through RPC communication. Figure 6C,D shows the performance of the GPGPU task processing part only except for data transfer. The experimental results in Figure 6 show that the performance of GPGPU operations is degraded by up to several orders of magnitude when compared to the individual execution times listed in Table 1. In Figure 6, the performance of newly requested GPGPU operations is worse than when multiple VMs are running the GPGPU tasks simultaneously as in Figure 5. The performance of the seven VMs that comprise the first group of VMs was evaluated during the GPGPU operation; the performance decreases compared to the results of the experiment using the seven VMs in Figure 5. In addition, the VMs that comprise the second group started working with insufficient GPU memory; therefore, the input/output time of the data to the GPU memory is increased. Consequently, the overall performance of the GPGPU work was reduced.

We additionally perform network performance measurements between the VM and the GPGPUvm to see how the data transfer time affects the performance of the GPGPU task when the user VM sends data to the GPGPUvm for use by the GPGPU task. The higher the network performance deviation between VMs, the higher the difference in the GPU operation start time for each VM when GPGPUvm processes the GPGPU task. If the network performance is irregular between VMs, even when the same task is executed at the same time, the time to complete the network transfer affects the GPU usage time of the GPGPU task, resulting in unpredictable performance imbalance problems. We used Iperf[16] to measure network bandwidth usage between VMs and additionally measured the data transfer performance of each VM's GPGPU task using the data used for the actual GPGPU task. The experimental results are shown in Figure 7.

As as shown in Figure 7, when the input data for each matrix operation is sent to GPGPUvm, the data transfer time of each VM shows similar performance and the deviation occurs within 4%. This allows GPGPU operations requested by multiple VMs to perform GPU memory input operations and GPU operations at the same time in



(A) Performance of matrix multiplication of one virtual machine run with 7 virtual machines performing 12K×12K matrix multiplication.

(B) Average performance of virtual machines performing 12K×12K matrix multiplication.

(C) GPU computational performance of seven virtual machines performing 12K×12K matrix multiplication.

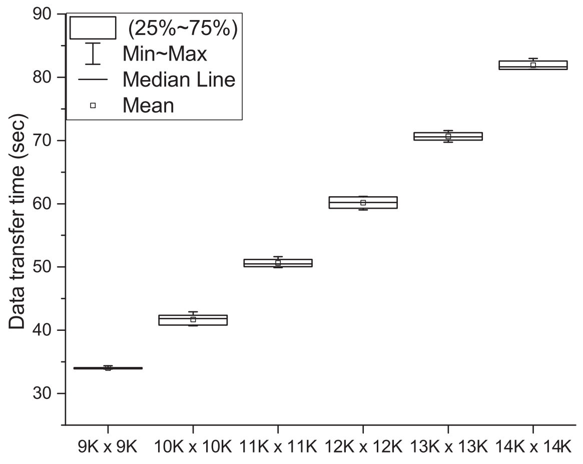
(D) The average performance of GPU operations on seven virtual machines performing 12Kx12K matrix multiplication.

FIGURE 6 Performance of newly requested general-purpose graphics processing unit task in the case of lack of graphics processing unit memory [Colour figure can be viewed at wileyonlinelibrary.com]

GPGPUvm. Since each VM uses network bandwidth evenly, the data transfer time increases uniformly as the size of data to be used for GPGPU tasks increases. Since the data transfer time is uniform in the execution time of each VM's GPGPU task, data transfer does not cause a problem of delaying the execution time of a specific VM's GPGPU task.

In the previous experiments, we confirmed that the performance of GPGPU tasks is degraded owing to insufficient GPU memory in the environment where multiple VMs are running GPGPU tasks simultaneously. In particular, in the case of the newly requested GPGPU task when the GPU memory is insufficient, the performance is considerably low owing to GPU memory contention. Furthermore, experimental results in Figure 6C indicates that the sequential GPGPU tasks processing exhibits a shorter execution time than the concurrent execution. Since it is almost impossible to improve the data transfer speed between the VM and GPGPUvm without replacing the network device, we improve the performance of GPGPU task processing by improving the performance of the GPU operation part. Also, because performance

FIGURE 7 Data transfer performance between user virtual machine and GPGPUvm



of communication between VMs, GPGPUvm and other nodes can be resolved through high-performance network Ethernet devices, our paper focuses on improving the performance of GPU operations. The GPU used in this experiment has 8 GB dedicated memory, and it can access approximately 3.7 GB shared system memory. Consequently, up to 11.7 GB data can be used for the GPGPU operations. However, the experimental results show that performance degradation was severe when the GPU memory requirements for the GPGPU tasks exceeded 10 GB. On considering this, we set the total GPU memory to 10 GB in our implementation.

In this paper, we propose a partial migration scheme of GPGPU tasks to increase the utilization of GPU memory and minimize the GPU memory contention. In our GPGPU task partial migration method, acceptable tasks are executed on the source node, and the remaining tasks are migrated to another node. Our approach differs from previous studies in that we do not migrate the entire GPGPU tasks or the VM, that is, we leave some of the allowable GPGPU tasks that can be run on the source node to minimize the unused GPU memory. In our approach, GPGPU tasks that cannot be accommodated in the source node are scheduled for migration; therefore, the burden on the destination node is reduced. Furthermore, the partial migration of the GPGPU tasks reduces the amount of the data to be transferred via the network. The proposed partial migration scheme for GPGPU tasks is described in the next section in detail.

4 | DESIGN AND IMPLEMENTATION

The proposed GPGPU task migration method works through three subsystems: (a) GPU resource monitoring, (b) GPGPU task migrating, and (c) data managing. GPU resource monitoring monitors the state of the GPU resources of the physical node and supports the selection of a destination node that has enough GPU resources to execute the GPGPU task without stopping the monitoring process. The GPGPU task migrator selects a task and inputs the data to migrate a GPGPU task for which data could not be input owing to insufficient GPU memory, and then, migrates the task to a destination node. Finally, the data manager selects the input data to be migrated together with the GPGPU task when the task is migrated. When the task has been completely migrated to the destination node, the data manager returns the operation result to the source node and integrates the result data therein.

4.1 | GPU resource monitoring and physical node selection

The most important step in task migration to prevent resource shortages is to find a destination node to which the task can be migrated. To do this, the tasks must be monitored to identify the correct amount of GPU resources available. To prevent performance degradation due to the lack of GPU resources, the GPGPUvm should find another physical node that can process the GPU tasks before they are migrated. In other words, the GPU task to be migrated must find a physical server that has enough resources to process the migration without any delay.

When GPU tasks are migrated, network overhead is inevitably generated because the data to be used for the GPU operations and the information about the functions to be executed are transmitted simultaneously through the network. Monitoring the network to find a destination physical server that has enough GPU resources to migrate the GPU work without any delay also generates network overhead. To minimize any unnecessary migration, an appropriate physical node must be selected to prevent chain task migration, such as the problem of remigrating the GPU task of a VM already on a destination node because the destination node exhibited a resource shortage owing to task migration from the source node.

The allocation of resources such as CPU, memory, and storage to the VM must be equal to the resources required by the user VM regardless of the actual resource usage of the VM. However, in the case of GPU resources, the GPU core cannot be divided and allocated to VMs independently, and general commercial GPU devices do not provide transparent GPU memory isolation. Moreover, resource monitoring is more complicated because the VMs that share the GPU change over time, depending on whether the GPU task is executed or not. Consequently, while monitoring GPU resources, the method of monitoring the amount of resources allocated to a running VM cannot be used.

Virtualization and cloud platforms such as Xen, KVM, and OpenStack provide monitoring tools for computing resources such as CPU, main memory, and storage by default; however, no monitoring capability exists for GPU resources. Therefore, cloud providers must use a separate GPU monitoring tool provided by GPU vendors to monitor the status of GPU resources. However, existing commercial monitoring modules track the overall state of the GPU device, not the information about 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. In the absence of available GPU memory, it is necessary to know the GPU memory requirements for each of the GPGPU tasks to perform their partial migration based on the available GPU memory capacity before hosting them.

In computing resource monitoring, the simplest way to measure the accurate GPU usage in real time is to set a short monitoring cycle. However, this method incurs network overhead owing to the frequent transmission of monitoring information. Furthermore, the frequent running monitoring task consumes computing resources. On the contrary, if the monitoring interval is long, the monitoring overhead can be minimized; however, the accuracy of the monitoring information is sacrificed, and the monitoring information can become meaningless.

In this study, we use an event-based lightweight monitoring technique for querying the available GPU memory. Our monitoring method records the GPU memory usage each time GPU memory allocation or deallocation occur. When a new GPGPU task is requested, the proposed method analyzes the GPU memory requirements of the VM through the parameters of the API related to the GPU memory allocation received from the user VM, then, compares it with the total capacity of the GPU memory to check whether there is a resource shortage. Our GPU monitor retrieves three pieces of information: (a) total GPU memory capacity, (b) individual GPU memory usage of GPGPU tasks, and (c) total number of executing GPGPU tasks. The GPU resource monitor runs in the GPGPUvm, which handles the GPGPU tasks of user VMs and records the monitoring information while receiving GPGPU APIs related to memory allocation requested from the user VMs. We use OpenCL for RPC-based GPU virtualization to query the total GPU memory capacity of the GPU device. The query for the total GPU memory capacity is executed once when the system is running, after which the GPU does not perform any monitoring work.

Information about the GPU memory capacity in use and total GPU memory capacity is used to determine if there is enough GPU memory available when a new GPGPU task is requested. If the amount of available GPU memory is less than the GPU memory requirement of the GPGPU task, it regards that GPU memory is insufficient. This procedure is done before executing the newly requested GPGPU task and the partial migration of the GPGPU task begins. Information about the total number of threads in the running GPGPU tasks is used to determine the destination node to which the partial GPGPU tasks will be migrated. We use the total thread count information of the GPGPU tasks running in the GPGPUvm to estimate the load level of the GPU core. In our previous experiments, we found that when the number of threads executing the GPGPU task is above a certain level, the core usage becomes 100% during the GPU computation. If we perform 9000×9000 matrix multiplication (the smallest matrix size for the experiment), the usage of the GPU core reaches 100% during the GPU processing in the GPGPU task. Consequently, we focus on the number of threads that perform GPGPU operations on the user VM currently running on the GPGPUvm rather than the actual usage of the GPU core. Information about the total number of threads of the running GPGPU tasks is collected through the kernel function parameter, and the thread count information is updated as the kernel function executes and completes.

In our GPGPU task partial migration technique, if the source node runs out of GPU memory, an appropriate destination node is selected and the GPGPU task is scheduled for migration. From the information of the available GPU memory capacity and the GPU memory allocation for newly requested GPGPU tasks, we can preview GPU memory shortages before new GPGPU tasks run on the GPU. If the source node cannot accommodate all the requested GPGPU tasks, a thread group of appropriate size for the available GPU memory runs on the source node, and the remaining threads are migrated to the destination node. To this end, we perform a two-step information exchange to select a destination node. In the first step, the source node notifies the destination nodes of the GPU memory shortage by sending the GPU memory capacity information required by the GPGPU task. In the second step, the destination nodes that received the GPU memory requirements inform the source node that they can migrate by sending the total number of threads of the running GPGPU tasks to the source node if their GPU memory is sufficient. In this stage, candidate destination nodes that do not have sufficient GPU memory do not respond. The source node that receives information on the number of threads selects the destination node with the least total number of threads among the currently running GPGPU tasks. The proposed GPU resource monitoring and migration destination node selection method are presented in Algorithm 1.

Algorithm 1.

```
 1: ToTGPUmem = Get total GPU memory capacity
 2: usedGPUmem = 0
 3: while When GPGPU task is running do
 4:     if GPGPU task requested from VM n then
 5:         if GPU memory allocation requested from VM then
 6:             if available GPU memory > Allocation Requested Capacity then
 7:                 usedGPUmem =+ Allocation Requested Capacity
 8:                 Data upload to GPU memory
 9:             else                                                  ▷ Node Selection
10:                 Send GPU memory requirements to all nearby nodes
11:                 if The number of responded nodes == 1 then
12:                     Start GPGPU task partial migration
13:                 else if The number of responded nodes > 1 then
14:                     Sort the node IDs in ascending order by number of threads on nodes
15:                     Migrate GPGPU tasks to the node with the least number of threads
16:                 else if The number of responded nodes == 0 then
17:                     Wait for available GPU memory
18:                 end if
19:             end if
20:         end if
21:         if GPU memory free requested from VM then
22:             usedGPUmem =- The capacity of the requested object
23:         end if
24:         if kernel function requested from VM then
25:             threadNum[n] = global work size of VM n
26:         end if
27:     end if
28: end while
```

Our GPU resource monitor (lines 1-21) continuously runs until the GPGPUvm releases all the allocated GPU memory area by the user VM's request. In other words, there is a data object sent by the user VM to the GPGPU task left, the GPU resource monitoring task is executed. As shown in lines 9, 15, and 17, whenever a user VM requests GPU memory allocation, freeing, and kernel function execution, it records the GPU memory usage and the number of threads executing the kernel function. The node selection algorithm (lines 22-30) executes when the GPU memory capacity required by the newly requested GPGPU task exceeds the available GPU memory capacity as shown in line 14. If it is determined that the GPU memory capacity is insufficient, a migration is requested from a neighboring node as shown in line 23, and the node with the least number of threads executing the kernel function is selected as the destination node as shown in lines 27 to 28.

As shown in the algorithm, our GPU monitoring method uses the GPU API query only once to determine the total GPU memory usage. Then, it analyzes the APIs related to GPU memory allocation and deallocation that are requested by the user VM and records the available GPU memory capacity. In the case of GPU core usage, the load of the GPU core is determined based on the total number of threads of the GPGPU tasks running by the GPGPUvm. In our proposed GPU resource monitoring, we do not perform periodic monitoring to track GPU memory information. In addition, when multiple GPU tasks are executed, the GPU core usage quickly reaches 100%; therefore, the information about the GPU core usage is not very helpful while selecting nodes with the least GPU core to migrate the GPGPU operations. Consequently, measuring the total number of threads of the GPGPU tasks requested by user VMs is more efficient than checking the amount of the GPU core usage.

The proposed GPU resource monitoring and target node selection method for migration require only tens of kilobytes of memory to manage the GPU resource monitoring information, and it performs minimal information exchange for the task migration. Our monitoring method collects GPU resource information based on events such as GPU memory allocation, freeing, and kernel execution of the user's VM, which requires considerably less overhead than the conventional method of collecting periodic monitoring information. Moreover, the monitoring task has no effect on the GPU because no instructions are performed by the GPU to extract the GPU resource information.

4.2 | Input data management

To migrate GPU tasks, the input data used for the GPU operations also must be transmitted from the main memory to the GPU memory because the data cannot be directly written to the GPU memory. Although a few commercial technologies are designed for transferring data directly between GPUs at different nodes, many open source-based GPU drivers do not support data transfer between GPUs. Therefore, we transfer the data from the main memory of the source node to the GPU memory of the target node, as shown in Figure 8.

In the RPC based GPU virtualization environment, the data to be used for GPU operations is sent from the user VM to GPGPUvm. The GPGPUvm uploads the input data of the GPGPU task requested by the user's VM to the GPU memory and performs the work according to the GPGPU programming flow. Uploading GPGPU input data to GPU memory when the memory is running low leads to GPU memory contention among the VMs, thereby causing performance degradation. To minimize GPU memory contention, our partial migration technique splits the data to be uploaded to the GPU memory; it uploads only that data that is acceptable at the source node, and migrates the rest to the destination node.

The data partitioning during the migration to the destination node is complex owing to some characteristics. Specifically, the following characteristics of the GPGPU programming model must be considered while partitioning the data for its upload to the GPU memory. First, GPGPU tasks cannot stop and restart in the middle of thread execution. These tasks do not have context switching capabilities, unlike what CPUs have. Second, multiple threads share a data object and access the elements of the data object in a defined pattern. As aforementioned, the data to be used for GPU

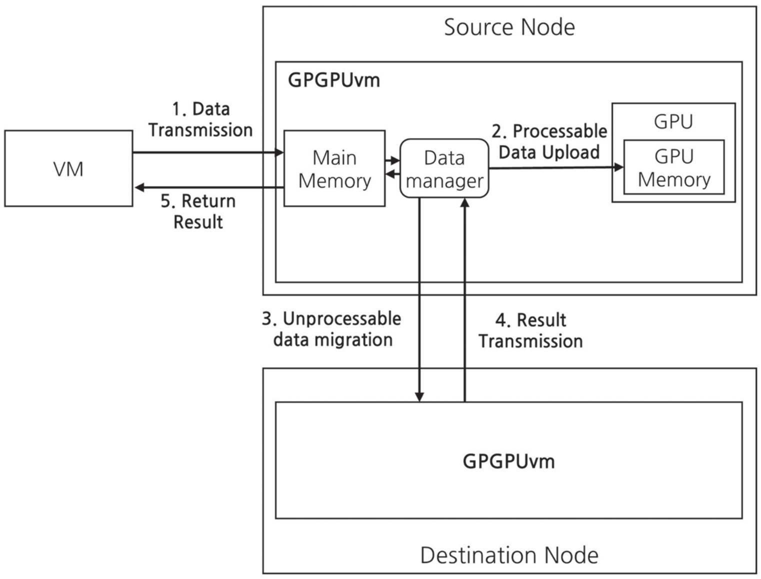


FIGURE 8 Data migration for graphics processing unit task

operations must reside in the GPU memory. If the data required by each thread does not exist in the GPU memory, the task will fail. These two characteristics cause GPU tasks to fail if the data required by each thread is not present in GPU memory. Moreover, context switching is not possible during thread execution; therefore, it is not possible to suspend a thread and restart it after uploading the data. For successful completion of GPGPU tasks, the data required by each thread must exist in the GPU memory before the thread starts. Data objects that are uploaded to the GPU memory for GPU work are a one-dimensional collection. To manage data objects in thread units, the index information of data objects is required according to each thread ID. We use the source code information to identify the index of the data object by thread ID. OpenCL, which was used for the RPC-based GPU virtualization environment, supports an online compilation method that compiles kernel codes before the GPU program and the kernel function are executed. GPGPUvm can be used to collect the index information of data objects before executing the kernel code sent by the user VM.

In other words, this method analyzes the kernel source code, extracts the index of the data object by each thread ID, and manages the data for each thread. Data divided in the thread units are classified into two groups. The first is the source group that uses the source node's available GPU memory, and the second is the destination group running on the destination node after migration. The source group runs on the available GPU memory of the source node, and the destination group is migrated to the destination node to avoid GPU memory shortage. In other words, among the total threads performing the GPGPU task, the remaining threads and data in the source node are migrated to the destination node once the threads that use the available GPU memory of the source node are determined. Our partial migration technique for GPU tasks can make the best use of the GPU memory of the source node and minimize the GPU memory footprint through partial migration of GPGPU tasks. More importantly, our method does not fully migrate GPGPU tasks; therefore, the data to be migrated is minimized, thereby reducing the burden on the destination node to process the migrated tasks. How to determine the partition size of a data object is described in the next section in detail.

4.3 | Task migration and processing

Lack of GPU resources can be explained as either a lack of a GPU core or a lack of GPU memory. The former means that there are many VMs using the GPU, such that multiple GPU tasks are scheduled to use the GPU, which increases the wait time of the task. The latter means that although the GPU data must be ready for the GPU operation in the GPU memory, the GPU operation cannot be executed since the input data cannot be copied to the GPU memory. GPU memory shortages delay the GPU memory usage of GPGPU tasks. As a result, kernel function cannot be executed and it is not included in the scheduling target. Note that the lack of a GPU core is a less serious problem than the lack of GPU memory because the GPGPU task is completed relatively quickly. Performance degradation due to the lack of a GPU core is likely to start quickly once the GPU memory stores the data for the task. Therefore, we focus on the problem of GPU memory shortage.

GPGPU task migration in an RPC-based GPU virtualization environment is straightforward. Our system shows information such as the parameters of the GPGPU function and efficiently divides the thread of the GPGPU operation. In the foregoing monitoring system, when a destination node to which a task is to be transmitted is found, a part of the GPGPU function is transmitted through RPC communication, which is possible because the GPGPU task identifies each thread as an index owing to the nature of GPGPU parallel programming, wherein all the threads bound to a function are different, and they all handle the same task. In our method, the input data for the GPGPU operation and the index value of the thread are sent to the GPGPU VM at the destination node to process the task and it returns the task result. As our task migration method also migrates GPU tasks to other nodes and transfers the data to be used for the operations, the GPU memory and main memory areas to be used by the migrated task must be allocated to the GPU memory of the destination node. For dealing with this, there is a dedicated GPU data management system for managing the data of migrated tasks. Figure 9 shows the overall flow of our proposed partial migration scheme for GPGPU tasks.

The GPGPUvm checks the GPU memory capacity required by the GPGPU task through the parameter information of the GPU memory allocation API when a new GPU task of the user VM is requested. Next, the GPGPUvm compares the available GPU memory capacity with the GPU memory capacity required by the GPGPU task to check whether the newly requested GPGPU task is eligible for migration. If there is sufficient available GPU memory, the GPGPUvm hosts the entire GPGPU task at the source node without performing the data partitioning for migration.

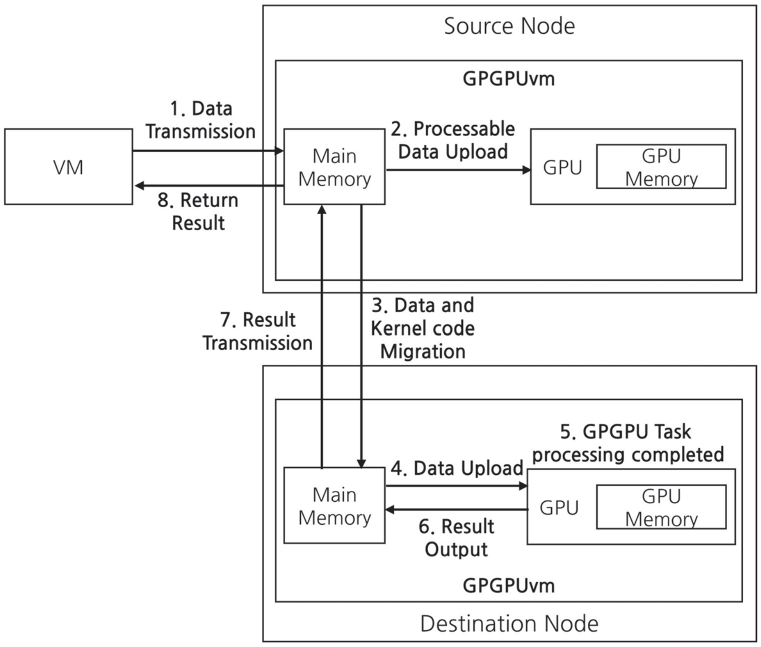


FIGURE 9 Task partial migration for general-purpose graphics processing unit task

To divide the GPGPU task between the source and the destination nodes, the threads created by the GPGPU task are divided based on their thread IDs. All threads created in a single GPGPU task perform the same GPU operations. When GPGPU work is executed, each thread is executed in the form of a two-dimensional work group, and it is managed with two thread IDs for the x and y axes. Before executing the kernel function, the total number of threads and the size of the workgroup to execute the kernel function can be known in advance through the workgroup and kernel function parameters defined by the GPGPU API. Information about the total number of threads and the workgroup is used to determine how many threads can be distributed to the source and destination nodes. The GPU data partitioning divides data in thread units. As aforementioned, we manage the index of the data required by each thread from thread ID 0 to n for the partial migration of tasks. GPGPUvm divides the threads into those to be processed on the source node and those to be migrated to the destination node based on the amount of data required for each thread. Thread partitioning determines whether to process a thread on the source node or to migrate it to the destination node in order from ID (0, 0). The working of our partial migration technique for GPGPU tasks is outlined in Algorithm 2.

Algorithm 2.

```
1: TotalThreadnum = global work size of VM n;
2: ThreadToSource = Amount of Available GPU Memory / Sizeof(DataPerThread)
3: if ThreadToSource < Total Number of Threads * ($n$/100) then
4:     GPGPU Task full migration
5: else
6:     ThreadToMigrate = (Total Number of Threads - ThreadToSource)
7:     SourceThreadID[2] = { 0, ThreadToSource }
8:     MigratedThreadID[2] = { ThreadToSource + 1, Total Number of Threads -1 }
9:     Send Kernel code file to Destination Node
10:    RunKernel( SourceThreadID );
11:    Number of threads to migrate = Total Number of Threads - ThreadToSource
12:    TaskMigration(MigratedThreadID[], Data Object, Kernel code file);
13: end if
```

As aforementioned, while performing a GPGPU task migration, the entire GPGPU task is migrated when the newly requested GPGPU task has less than n% of the GPU memory available, as shown in lines 3-4 (n is set to 30 in our experiment). If extremely less GPU memory is available, then data partitioning to migrate GPGPU tasks to the destination node is an unnecessary overhead because most of the GPGPU tasks must be migrated to the remote node. During the partial migration of GPGPU tasks, the tasks are migrated based on the thread IDs as shown in lines 6-10, and the number of threads processed in the source and destination nodes is recorded and managed. The thread ID that we manage for the GPGPU task migration is not the ID of the actual thread; it is a virtual ID for the convenience of management that manages the number of threads processed in the source and the destination nodes by mapping OpenCL's two-dimensional thread ID as one-dimensional ID.

In our partial migration scheme, GPGPU tasks that are split between the source and the destination nodes are essentially a single task. Splitting a GPGPU task means that one GPGPU task is split into two processes: one for the source node and the other for the destination node. If the GPGPU task requested by the user VM is passed to the GPGPUvm, and the GPGPU task needs to be migrated, then the GPGPUvm processes the threads that it can accommodate. The remaining threads that cannot be processed owing to insufficient GPU memory request the GPGPUvm of the destination node to process part of the GPU work. The GPGPUvm of the source node requests the GPGPU task to the destination node's GPGPUvm in the same manner that the user VM requests the GPGPU task. The GPGPUvm on the source node sends the data and the kernel code for the GPGPU task to the GPGPUvm on the destination node, which handles the migrated GPGPU task in the same manner that it handles the GPGPU task of the user's VM. The GPGPUvm of the source node records the thread ID corresponding to the GPGPU task that migrated to the destination node while requesting a part of the GPGPU tasks to the destination node's GPGPUvm.

When the GPGPUvm of a destination node processes a migrated GPGPU task, it recognizes the task as an independent GPGPU task; therefore, the ID of the thread handling the GPGPU task will start at (0, 0). The task result of the migrated GPGPU task must be integrated with the result of the GPGPU task processed by the source node and returned to the user VM that requested the GPGPU task. Hence, to manage the result of a task as if it were processed by one GPGPU task, the GPGPUvm must manage the thread ID system as if it were one, so that, the correct index value corresponding to the thread ID can be determined. When the source node returns the result of the migrated GPGPU task to the destination node, it records and manages the thread ID corresponding to the migrated GPGPU task to integrate the result of the GPGPU task processed with the returned result. As aforementioned, we split the GPGPU tasks in the order of thread IDs; therefore, only the start and the last thread IDs of the migrated GPGPU task need to be managed.

When we perform partial migration of GPGPU tasks in our proposed GPGPU task partial migration scheme, the tasks and the data are transferred in two stages: (a) from the user VM to the GPGPUvm of the source node and (b) from the source node to the GPGPUvm of the destination node. However, if there is extremely little GPU memory available on the source node, most of the GPGPU tasks may be sent to the destination node's GPGPUvm. If the source node cannot handle the appropriate part of the GPGPU task, then sending the GPGPU task in two stages to the GPGPUvm of the destination node from the user VM introduces unnecessary overhead. To avoid this unnecessary network overhead and shorten the migration phase, we also support full task migration, wherein the entire GPGPU task of the VM can be sent directly to the GPGPUvm of the destination node if the source node's available GPU memory is below a required size. The full migration of GPGPU tasks is as follows. If GPU memory allocation is requested from the user VM when there is almost no available GPU memory, the IP address of the RPC server registered in the user VM is changed to the GPGPUvm of the destination node so that the GPGPU task can be delivered directly to the destination node. In the partial migration scheme of GPGPU tasks, we set the condition for the full migration of the GPGPU task to be that the available GPU memory capacity is lower than 10% of the GPU memory requirement. The threshold of 10% is based on the result of measuring the performance of the GPGPU task used in our experiment.

4.4 | Return of GPGPU task results

Our method of migrating tasks passes input data and function parameters to the GPGPU VM at the destination node. Furthermore, when the GPGPU task has been completely migrated, the task result must be returned to the user's VM. Because the GPGPU function cannot allocate GPU memory during the execution, the GPU memory area required for storing the result of the GPGPU operation must be declared before the GPGPU function is executed. Since our method uses a modified GPGPU API in an RPC-based GPU virtualization environment, the GPGPU VM receives information about the function related to the GPU memory allocation during the GPGPU task execution, so it knows the types and sizes of the input and result data. Multiple threads share a single data object in the case of the result data of the GPGPU task, which is similar to the input data. Threads generally perform only data reads on input data objects; therefore, multiple threads can access the duplicate indexes of data objects. However, in the case of the result data object, each thread performs data creation; therefore, the thread ID and the index of the result data object are mapped in a one-to-one correspondence to prevent data creation conflict. We use the characteristic that the thread ID and the index of the data object mapped as a one-to-one fashion while integrating the result of the migrated GPGPU task with the GPGPU task processed by the source node and returning it to the user VM. As aforementioned, the resulting data object must be allocated before the kernel function is executed. Similar to the manner in which we split the input data object, we analyze the kernel code and manage the mapping information between each thread ID and the index of the result data object.

When the destination node returns the result data, the GPGPUvm of the source node stores the data in main memory. After the result of the GPGPU task processed at the source node is output to the main memory, it must be integrated and returned to the user VM. Immediately after the completion of the GPU operation, the resulting data can be found in the GPU memory. APIs such as clEnqueueReadBuffer() that copies the result data from the GPU memory to main memory must be called before the result data is copied to main memory. However, our migration method does not upload the result data returned from the destination node to the GPU memory to minimize GPU memory consumption. The results returned from the destination node are stored in main memory because they are sent over the network. We do not upload the result data object to the GPU memory until the result data object is called again for reuse to avoid unnecessary GPU memory allocation. When the GPGPU tasks complete, the result data is stored in main memory because it is more likely to be returned to the user VM. The resulting data object is managed by GPGPUvm until the clEnqueueReadBuffer() API is requested to output the data object in GPU memory to main memory from the VM.

While performing the partial migration of GPGPU tasks, three-step data copy operations are performed: (a) the input data is output from the source node's GPU memory to the main memory, (b) transferred to the destination node's main memory via RPC, and then (c) copied to the GPU memory. In the general GPGPU programming flow, when the kernel function finishes executing, the result data of the kernel function is in the GPU memory. Therefore, to return the result data to the source node, three-step data copy operations must be performed in the reverse direction. However, we keep the result data in the source node's main memory while returning the result data to the source node to limit the use of the GPU memory and for the speedy return of the result to the user VM. If the GPGPU task calls the result data object to reuse the result data, no additional overhead incurs because only the last step omitted from the three-step data copy operations is performed.

5 | PERFORMANCE EVALUATION

In this section, we evaluate the performance of our partial migration technique for GPGPU tasks and validate its efficiency through experiments. Our proposed migration technique was developed based on the open-source-based virtualization platform, Xen hypervisor 4.4.1. We setup 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 is assigned to GPGPUvm using direct pass-through and other VMs cannot access the GPU. The guest OSes of VMs are Windows 7 as listed in Table 2. The GPGPU task to be executed in the VM is the matrix multiplication.

TABLE 2 Experiment environment

	Source and & destination node	Source node's GPGPUvm & destination node's GPGPUvm	User virtual machine
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) & Radeon R9 380(2 GB)	—
OS	Ubuntu 14.04	Windows 7	Windows 7
Hypervisor	Xen 4.4.1	—	—
Network	1Gb/s Ethernet interface		

Abbreviations: CPU, central processing unit; GPGPUvm, VM that handles the GPU tasks of users VMs.

TABLE 3 Experiment scenario

	Group 1 (7 VMs monopolizing GPU memory)	**Group 2 (1 VM for migration)**
Experiment 1 (GPGPU task partial migration)	11K × 11K	9K × 9K
	11K × 11K	10K × 10K
	11K × 11K	11K × 11K
	11K × 11K	12K × 12K
	11K × 11K	13K × 13K
Experiment 2 (GPGPU task full migration)	12K × 12K	9K × 9K
	12K × 12K	10K × 10K
	12K × 12K	11K × 11K
	12K × 12K	12K × 12K
	12K × 12K	13K × 13K
	12K × 12K	14K × 14K

Abbreviations: GPU, Graphics processing unit; GPGPU, general-purpose graphics processing unit; VM, virtual machine.

We use two physical nodes for the experiment as shown in Table 1. A source node is a node wherein several VMs perform the GPGPU tasks, and a destination node is a node for processing only the migrated GPGPU tasks when the GPU memory of the source node is insufficient. The destination node used in the experiment also runs GPGPUvm, which a VM for processing migrated GPGPU tasks.

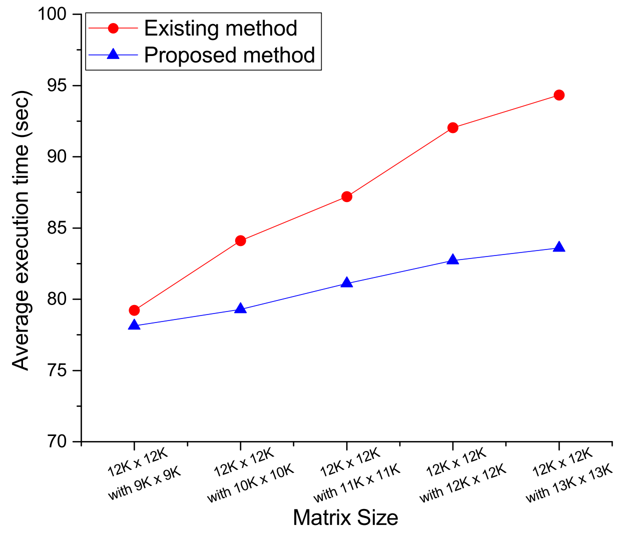
5.1 | Performance evaluation of VMs

Two experiments are configured for the evaluation as shown in Table 3. The first experiment is to show the performance of the VMs when the partial migration method of GPGPU tasks is used. The second experiment is to show the performance of the VMs when the full migration of GPGPU tasks is performed. VMs are classified into two groups. Group 1 consists of seven VMs for monopolizing GPU memory, and group 2 has one VM to be migrated. The sizes of matrix multiplications performed by each group are shown in Table 3.

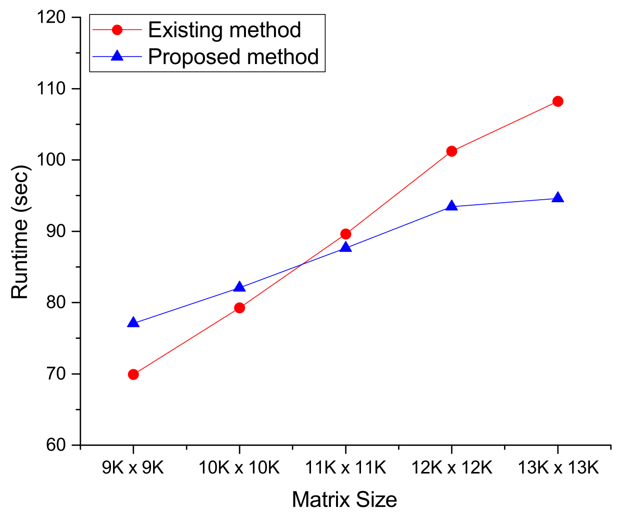
The first experiment measures the performance of VMs when using the proposed partial migration technique of GPGPU tasks in the absence of GPU memory. Here, we use seven VMs that monopolize the GPU memory until the GPU memory runs out owing to the GPU memory shortage caused by an eighth VM (group 2). The seven VMs run the same GPGPU task simultaneously, and the eighth VM is scheduled to run the GPGPU task, one second later than the seven VMs. When only seven VMs run, the usage of GPU memory is within the allowed range; however, when the eighth VM is running, it gradually consumes the remaining GPU memory and eventually causes the out of GPU memory situation. The scale of the migrated GPGPU task can be diverse since the available GPU memory changes according to the size of the GPGPU tasks performed by the VMs. This experiment measures the performance of a VM when the partial migration of GPGPU tasks is used when the GPU memory required by the GPGPU tasks exceeds the available GPU memory. Figure 10 shows the experimental results for the experiment.

As shown in Figure 10, in the existing environment, when the GPU memory is insufficient, the performance of the GPGPU tasks running in the VM reduces significantly. As shown by the results presented in Section 3, when the GPU memory was sufficient, there was performance degradation when we increase the number of concurrent VMs, but it did not significant affect the execution. However, when the GPU memory was insufficient, the performance degradation of the VMs is substantial. The seven VMs running the GPGPU task experience performance degradation if a new VM is added to run the GPGPU task. In particular, the GPGPU task of the VM that has executed one previously leads to an increased execution time to obtain the result data; the main reason of the performance degradation is due to the newly added VM when the GPU memory is insufficient.

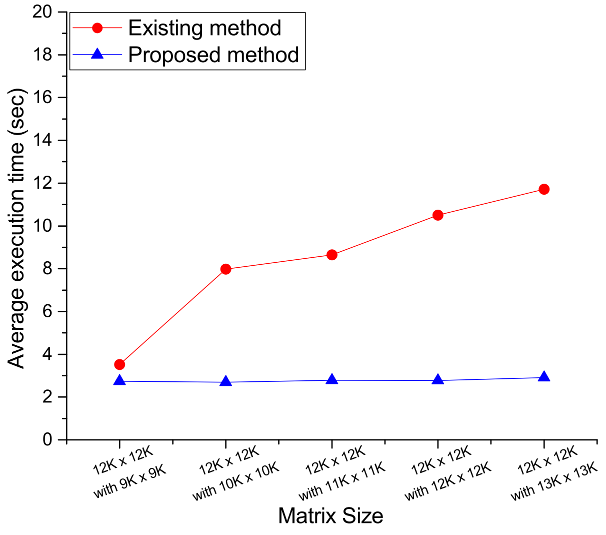
As explained in Section 3, we revealed that the performance of the GPU used in our experiments degraded the performance significantly when the required GPU memory for the GPGPU tasks exceeds 10 GB during execution. In the



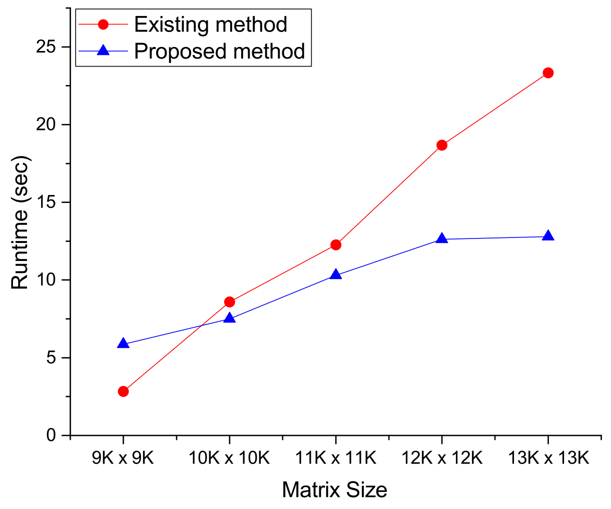
(A) The average performance of seven VMs performing 11K×11K matrix multipli-cation.



(B) Matrix Multiplication Performance for Virtual Machines Running with Seven VMs Performing 11K×11K matrix multiplication.



(C) The average execution time of the GPU operation portion of VMs performing 11K×11K matrix multiplication.

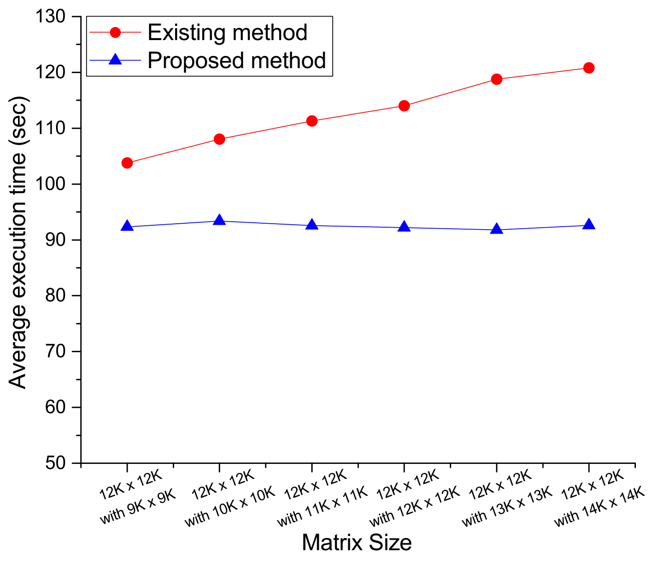


(D) Execution time of the GPU operation portion of the VM running with the VMs performing 11K×11K matrix multiplication.

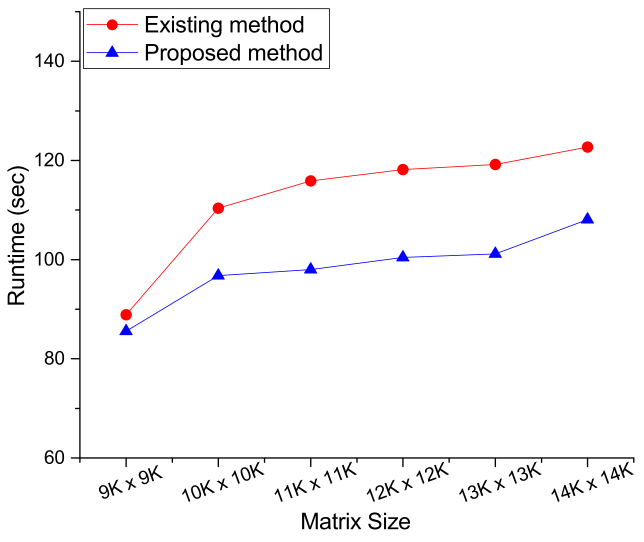
FIGURE 10 Performance improvements owing to graphics processing unit task partial migration [Colour figure can be viewed at wileyonlinelibrary.com]

experiment performing 11 000 × 11 000 matrix multiplications, each VM requires approximately 1.3 GB GPU memory for input and output data, and the seven VMs require approximately 9.4 GB GPU memory during concurrent executions. In consequence, all the eight VMs performing the matrix multiplication require approximately 11 GB GPU memory. For this reason, the seven VMs running GPGPU tasks can be executed tolerating some degree of performance degradation. However, if one additional VM is added, the total amount of the GPU memory in use will exceed 10 GB. As shown in Figure 7, when GPU memory requirements exceed 10 GB in a traditional environment, there is a significant performance penalty. However, with our partial migration scheme of GPGPU tasks, the VMs that are executed first experience considerably less overhead, and it can afford the GPGPU task of the newly added VM. Since some of the migrated GPGPU tasks are performed on other nodes in the cluster, they can be the tasks without the delay caused owing to the wait for the GPU resource.

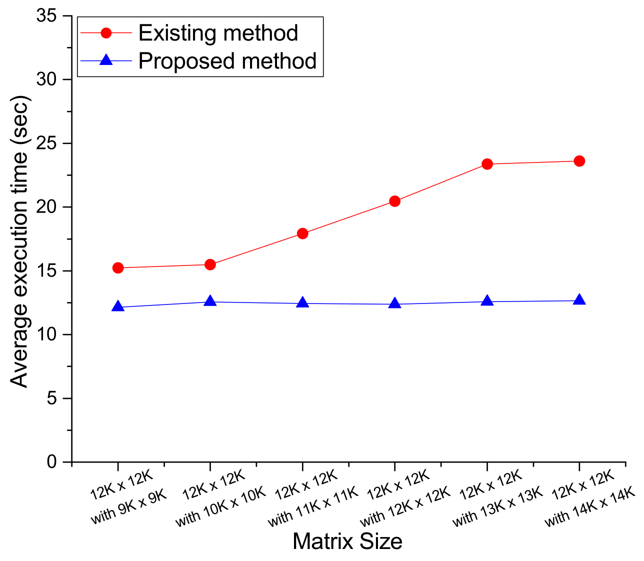
The second experiment measures the performance of a newly added VM in a situation where there is insufficient GPU memory. In other words, we measure the overall migration performance of the GPGPU task. Similar to the previous experiment, this experiment uses eight VMs; seven VMs for occupying the GPU memory, and the eighth VM for



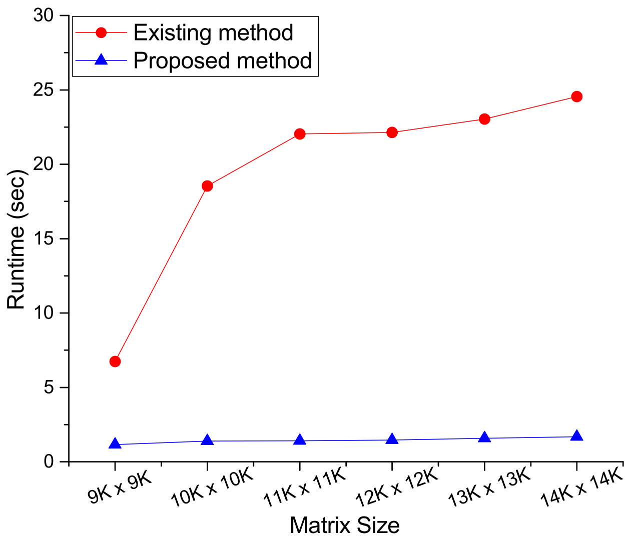
(A) Average performance of seven VMs performing 12K×12K matrix multiplication



(B) Matrix Multiplication Performance for VMs Running with Seven VMs Performing 12K×12K matrix multiplication



(C) Average execution time of the GPU operation portion of VMs performing 12K×12K matrix multiplication



(D) Execution time of the GPU operation portion of the virtual machine running with the VM performing 12K×12K matrix multiplication

FIGURE 11 Reduced resource competition for virtual machines due to full migration of general-purpose graphics processing unit task [Colour figure can be viewed at wileyonlinelibrary.com]

measuring the performance of GPGPU task migration. The seven VMs perform 12 000 × 12 000 matrix multiplications, each requiring approximately 1.6 GB GPU memory. They require 11 GB GPU memory in total, thereby over-using the GPU memory. The eighth VM performs various sizes of matrix multiplications to measure the performance. As described in the previous Section 4, the user VM is connected to the GPGPUvm on another node if no available GPU memory exists to avoid GPU memory contention. Furthermore, when the GPU memory usage is high, the high utilization rate of the GPU core is likely to persist for a long time. However, our proposed technique minimizes the impact on existing GPGPU tasks by recognizing GPU memory shortages and migrating the GPGPU tasks before uploading the data to GPU memory.

As shown by the experimental results in Figure 11, the full migration of GPGPU tasks does not cause performance degradation of the currently running GPGPU tasks. However, such a migration requires a full migration of the input data used for the GPGPU task, which results in increased data transfer times because more data must be migrated than that for the proposed partial migration technique.

Using our proposed partial migration technique of GPGPU tasks, it checks for insufficient GPU memory for a new GPGPU task before uploading data to the GPU memory, processes only those threads that are available in the available GPU memory, and migrates the remaining GPGPU tasks to other nodes. Hence, the impact on the existing GPGPU tasks is considerably small and the performance of the newly executed GPGPU task are improved in comparison to the existing environment, wherein eight VMs are running simultaneously. Our approach increases the GPU memory utilization of the source node and reduces the size of the GPGPU tasks for migration to the destination node, thereby reducing the overhead of network data transfer. As shown in Figures 10 and 11, our partial migration technique of GPGPU tasks has reduced the data transfer overhead than that of the full migration.

Although the proposed partial migration scheme of GPGPU tasks solves the performance degradation problem of VMs owing to the lack of GPU memory, additional overhead is introduced by the partial migration of GPGPU tasks. The overhead incurred by GPGPUvm owing to the task migration can be divided into three cases: (a) when partitioning the GPGPU task data for migration, (b) when sending the data of the requested GPGPU task to the destination node, and (c) when returning the result of the migrated GPGPU task to the user VM. This overhead incurred by the partial migration of GPGPU tasks is discussed in the next subsections in detail.

5.2 | Additional overhead in migration

In the previous experiments, we found that the partial migration of GPGPU tasks solves the performance problem due to insufficient GPU memory. In this experiment, we evaluate the additional overhead due to the partial migration of GPGPU tasks. The additional overhead incurred in our proposed scheme is related to the GPGPU API information, data transmission, monitoring to locate the destination node while migrating the GPGPU task, and returning the result of the operation to the user VMs. We measure the turnaround time (TAT) of the migrated GPGPU tasks to analyze the overhead of the partial migration.

In general, the larger the GPGPU task being migrated, larger the number of threads that execute the data and kernel functions used for the GPGPU task. After migrating to the destination node, the kernel functions are executed in the form of a file; therefore, the number of threads in the migrated GPGPU tasks does not significantly affect the migration performance. Moreover, the transfers a kernel code file of several tens of KB, despite the execution of any number of threads; therefore, the number of the migrated threads does not contribute to an increase in the overhead.

The most significant part that impacts on the TAT of the GPGPU task is owing to the sizes of the data to be used by the GPGPU task and the result data to be returned to the user VM. The data transfer task consumes more than 95% of the GPGPU task's migration time. To figure out the overhead, we measure the TAT (from the partial migration of GPGPU tasks starts to it returns the results to the user VM) and the percentage of each overhead subset over the total TAT. Note that we assume a single VM runs GPGPU tasks of varying sizes and 50% of the GPGPU tasks are migrated to the destination node. We run all the additional work for the partial migration of GPGPU tasks to measure the overhead incurred by the partial migration.

The TAT values with varying sizes of the migrated GPGPU tasks are shown in Figure 12. The migration of GPGPU tasks incurs additional overhead in comparison to just running a GPGPU task on the source node since the data and information related to the GPGPU task must be sent to the destination node, and the results of the task must be transferred over the network. However, comparing the GPGPU task migration overhead with the performance degradation caused by insufficient GPU memory, the migration overhead is relatively low, and the partial migration effectively solves the poor performance problem when executing all GPGPU tasks simultaneously in case of insufficient GPU memory. Most of the migration overhead can be attributed to the data transfer operations. As shown in the experimental results, monitoring and data partitioning account for less than 5% of the overall overhead. However, data transmission accounts for the largest part (95%) of the overall overhead because data must be transferred from a source node to a destination node. When the user VM and the GPGPUvm are on the same node, they use the internal network; therefore, there is less overhead of the data transmission. However, when migrating GPGPU tasks to another physical node, data must be sent to the destination node over the network, thereby incurring significantly high network overhead.

The migration overhead can be measured as follows. In the case of migration overhead, the overhead increases with the size of the data for the GPGPU task. In the case of resource monitoring overhead, the overhead will increase as the number of physical nodes increases. To reduce the resource monitoring overhead, we modified the GPGPU API to keep track of GPU memory usage about allocation and deallocation commands for the GPU memory with the modified GPGPU API. When a GPU memory allocation command is requested, it is immediately possible to check the physical node for the

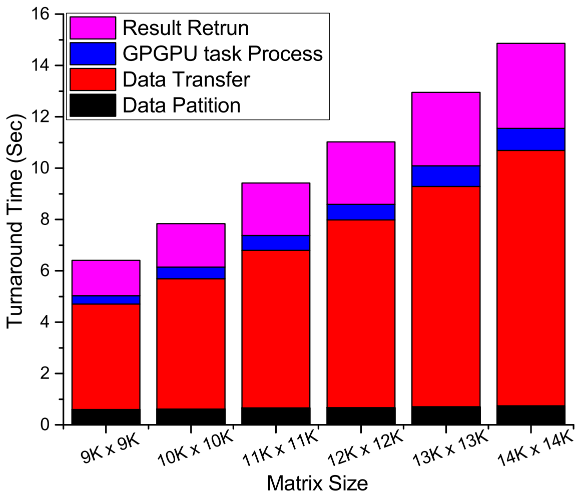


FIGURE 12 Turnaround time of general-purpose graphics processing unit task when migrating 50% of each matrix multiplication [Colour figure can be viewed at wileyonlinelibrary.com]

lack of GPU memory. If so, the physical nodes do not need to look up the GPU memory usage status whenever a request for the GPU resource information occurs on another node. In addition, only physical nodes that are satisfied with GPU memory requirements are configured to return monitoring information to minimize unnecessary overhead. In our experimental setup, the source and the destination nodes are connected by a 1 Gb/s Ethernet. However, using high-performance network interfaces such as InfiniBand and high-performance Ethernet devices is capable of transferring data with the magnitude of several tens of Gb/s, thereby reducing the overhead of data transmission.

As shown in the experimental results, we can confirm that the proposed method efficiently avoids the GPU memory shortage phenomenon. In addition, even when the GPU memory is insufficient, the VMs keep running since it migrates only a part of the GPGPU task, such that the VMs do not stop and the data transmitted through the network is relatively smaller than the migration of the entire GPGPU task. For the GPU resource monitoring, the modified GPGPU API is used to keep track the information about GPU memory allocation and deallocation, thereby reducing the monitoring overhead because no unnecessary monitoring is performed. This monitoring information takes up only a few kilobytes of memory and, thus, introduces little network overheads for the monitoring information exchange and migration scheduling.

6 | RELATED WORK

Previous work has proposed various methods of migrating VMs to prevent resource shortages. In a cloud environment, “migration" usually refers to the migration of the entire VM. Most of the existing researches on VM migration focus on minimizing the VM's downtime due to migration and on enabling VMs to get up and running again as quickly as possible after they have been migrated. However, the full migration of a VM causes network overheads and, therefore, the impact on the whole system cannot be ignored since users cannot access the VM in the meantime of the migration. In addition, the existing VM migration technologies cannot be used in GPU virtualization environments because such technologies do not support the stopping and restarting of GPGPU operations. Previous work for migrating a VM performing a GPU task has suggested the possibility of a migratable GPGPU API mechanism that can save the state of a task and restart the task when necessary for the safe stopping and restarting of GPU operations.[17] The GPU operation of the VM is paused when the GPU workload is being migrated, and the state of the work is saved, which enables the GPU work to be restarted when the VM has been completely migrated to another physical node. In this VM migration technique, the GPU operations are suspended for the migration downtime, and the overhead is inevitably generated during saving task states and restarting the GPU task. We summarize related work into the three categories: (a) GPU memory race condition avoidance, (b), GPU task migration, and (c) VM migration for GPU resources.

6.1 | GPU memory race condition avoidance

Running multiple GPU tasks concurrently on a single GPU can not only enhance the utilization of GPU resources but also improve the system throughput. However, environments with multi-GPU tasks invariably run into resource contention. When GPGPU operations simultaneously access the GPU memory for writing or outputting data, a race condition arises. Furthermore, if the GPU memory capacity required by multiple GPGPU operations exceeds the available GPU memory capacity, then overhead is incurred owing to GPU memory contention, as shown in the experiments performed in our study; this results in overall performance degradation of the GPU operations. In the previous researches, various methods were introduced to minimize the contention of GPU memory.

GPUswap[18] is a technique to expand the GPU memory area to the main memory area when the GPU memory is over-reserved, and automatically swap unused data to the main memory to secure available GPU memory space. This technique copies a certain amount of data from a GPU application that uses the most GPU memory into main memory to prevent GPU memory running out when multiple GPU tasks share the GPU, allowing other GPU operations to use the GPU memory. In throughput-oriented GPU memory allocation,[19] the authors proposed a dynamic GPU memory allocation technique to improve concurrency in an environment where multiple GPU operations are executed simultaneously to prevent bottlenecks in GPU memory allocation. It uses bulk semaphores for dynamic GPU memory allocation, allowing simultaneous allocation of resources. LD[20] detects GPU race conditions and enables deterministic GPU execution. The LD uses compiler support and run-time technology to detect GPU resource contention and to support noncompetitive executions of GPU resources.

The aforementioned studies require several components changes of the system such as device drivers and OS kernels. This approach increases the complexity of the system and that of managing the cloud provider's systems due to the changes of a number of components. It also has a high dependency on the system, making it less adaptable to new devices and systems. On the other hand, our partial GPGPU task migration technique is relatively simple to apply to new GPU devices because it is not dependent on the system components such as OSs, hypervisors, and compilers of the GPGPU programming languages.

6.2 | GPU task migration

The migration of GPU tasks has traditionally been used in distributed processing systems to improve the performance by distributing a single GPU task to multiple physical nodes. Even in RPC-based GPU virtualization environments, the GPU task is migrated by changing the physical location of the destination server to process the migrated GPU task when GPU resources become scarce. This technique is similar to the basic mechanism of the traditional GPU task migration method to prevent performance degradation. Moreover, in RPC-based virtualization environments that share GPUs, GPU resource shortages create resource contention between VMs that share GPUs, reducing the performance of all GPU operations running on physical servers.

The general GPU task migration method includes the entire GPGPU task. VOCL[21] solves the local limitations of GPU devices by using modified GPGPU APIs to migrate GPGPU tasks to remote nodes so that nodes without GPUs can handle GPGPU tasks. Based on VOLC, the method in accelerator migration,[22] supports the load balancing of GPU resources across GPU clusters by migrating GPGPU tasks in the GPU cluster environment. Floating devices[23] is one of the GPU migration techniques based on rCUDA,[13] an RPC-based GPU sharing technology. The proposed technology supports the transparent provision of GPU devices to the data center by migrating the GPU computation part to the GPU server while the GPU application is running.

In a shared GPU environment, GPU task-wide migration methods focus on selecting the physical location of the destination node for the GPU task. In full task migration, the entire GPU task is migrated; therefore, when a task is migrated to another physical node, the available GPU memory of the source node is not utilized until a new GPU task is executed. For example, if there is 100 MB GPU memory in the source node, and the GPU memory required by the GPU task is 200 MB, then the resource manager recognizes that the GPU memory is insufficient. If the entire task is migrated owing to the lack of GPU memory, then the 100 MB GPU memory at the source node will remain unutilized until a new GPU task is executed. However, the proposed partial GPGPU task migration processes selected GPGPU tasks that fit for the available GPU memory capacity at the source node, and migrates the remaining part of the GPGPU tasks to the destination node. This minimizes the idle state of the available GPU memory at the source node and the data transmission overhead to the destination node. The overhead of data transfer occurs when performing partial migration of the GPGPU tasks, however, the downside for the data transfer overhead can be resolved by using high-performance network interfaces such as InfiniBand,[24] as in the VOLC[21] and transparent accelerator migration.[22]

6.3 | VM migration for GPU resources

VM migration is one of the ways to manage numerous VMs running in cloud centers. In the VM migration method, the VM is migrated to another node when the resource usage of the physical server exceeds a threshold to minimize the performance degradation and resource contention of the VMs.[25,26] Besides GPU resources, there are various technologies that allow the VM to be paused and reexecuted as soon as it is migrated, for example, context switching, dirty pages, buffer copying.

Crane[17] proposed a pass-through technique that can be migrated using a modified library and RPC communication to a remote GPU in a pass-through environment, where the GPU migration is not possible. Crane extracts the state of the GPU while the OpenCL program is running and migrates it to a remote node so that the GPU work of the migrated VM can continue. gMig,[27] proposed a technique to migrate GPU tasks to other nodes in real-time by precopying data in the GPU memory to other nodes and migrating dirty pages. gMig uses pre- and post-migration to minimize downtime while GPU tasks are running.

Existing methods of migrating the entire VM must transfer the entire information and data of the VM such as CPU status information and data that reside in the main memory, which introduces significant overheads. However, the proposed partial GPGPU task migration method utilizes both the source and the destination nodes while processing GPGPU tasks; therefore, the overhead of data transfer can be transparent during the partial migration. Furthermore, the overhead of VM migration does not occur because the VMs are not migrated.

Herein, we support the partial migration of GPGPU work so that the user's VM is maintained in source node and only a part of the GPGPU work is migrated and, therefore, no overhead is introduced due to the full migration of the VM. Furthermore, even after some of the work has been migrated to the destination node owing to the insufficient GPU memory, the work can be started at any time if sufficient GPU memory is available. Our approach uses an RPC-based GPU virtualization to avoid GPU memory shortage effectively. Moreover, because only a part of the GPGPU work and the corresponding data are migrated, we introduce negligible network overheads.

7 | CONCLUSIONS

Herein, we propose a partial migration technique of GPGPU tasks to prevent the performance degradation of VMs due to a lack of GPU memory and resources in an RPC-based GPU virtualization environment. Our method can prevent the network overhead caused by the migration of whole VMs and does not introduce the downtime of the GPGPU operations owing to the live migration of GPGPU tasks. Furthermore, our method does not require any modifications to the OS, hypervisor, or GPU driver since the proposed method takes advantage of the RPC-based GPU virtualization. In our method, only a portion of the GPGPU work is scheduled for migration while the user's VM is run on source node thereby minimizing the occurrence of work delays while simultaneously providing the benefits of distributed processing. As shown in the experimental results, migrating only a part of the GPGPU task generates negligible network overhead and no downtime occurs for both VMs and GPGPU tasks. Moreover, as no entire VM is migrated, the incurring overhead is minimized. In other words, it does not migrate unnecessary resources such as main memory data or CPU operation states since only a part of the GPU task is migrated.

The proposed GPU resource monitoring scheme also minimizes the amount of monitoring data by adjusting the monitoring frequency reflecting the characteristics of static GPU memory usages. In addition, because we migrate only a portion of the GPGPU task and its related data, the migration time is very well reduced, thereby returning the results as soon as it finishes. The proposed migration method does not include a task size determination algorithm for predicting the migration time by analyzing the available resources of the source or the GPU usage patterns of VMs to send an appropriately sized task to the destination node. These limitations can be mitigated by adopting advanced forecast-based monitoring techniques[28,29] in existing cloud environments. If the size of clusters constituting a cloud system increases due to an extension of physical nodes, the monitoring overhead will also increase. However, the monitoring overhead can be minimized by logically dividing groups of the cluster. Furthermore, the data transfer time, the largest portion of the migration overhead, incurred during the partial migration process of GPGPU tasks can be minimized through a high-performance network interface.[24] Commercially available high-performance network interfaces can transfer data at the speeds of tens of Gb/s, thereby migrating data can be done within a few seconds, even if the data size is of the order of several GBs.

ACKNOWLEDGEMENTS

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 & communications Technology Planning & Evaluation). This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00480, Developing the edge cloud platform for the real-time services based on the mobility of connected cars).

ORCID

JiHun Kang https://orcid.org/0000-0003-4773-6157

REFERENCES

1. Amazon. Amazon GPU instance. https://aws.amazon.com/ec2/instance-types/?nc1=f ls. Accessed October 25, 2019.
2. Google. Google compute engine. https://cloud.google.com/compute/docs/gpus/. Accessed October 25, 2019.
3. Kaur P, Rani A. Virtual machine migration in cloud computing. *Int J Grid Distribut Comput*. 2015;8(5):337-342.
4. AMD. AMD accelerated parallel processing OpenCL programming guide. 2013. http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf. Accessed October 25, 2019.
5. NVIDIA NVIDIA. CUDA C programming guide. 2012. https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. Accessed October 25, 2019.
6. Barham P, Dragovic B, Fraser K, et al. Xen and the art of virtualization. *ACM SIGOPS Oper Syst Rev*. 2003;37(5):164-177.
7. Khronos Group. OpenCL: open computing language. https://www.khronos.org/opencl/. Accessed October 25, 2019.
8. NVIDIA. CUDA: compute unified device architecture. http://www.nvidia.com/object/cuda_home_new.html. Accessed October 25, 2019.
9. Suzuki Y, Kato S, Yamada H, Kono K. GPUvm: Why not virtualizing GPUs at the hypervisor? Paper presented at: Proceedings of the USENIX Annual Technical Conference 2014:109-120.
10. Tian K, Dong Y, Cowperthwaite D. A full GPU virtualization solutionwith mediated pass-through. Paper presented at: Proceedings of the USENIX Annual Technical Conference; 2014:121-113.
11. Amiri Sani A, Boos K, Qin S, Zhong L. I/O paravirtualization at the device file boundary. *ACM SIGARCH Comput Archit New*. 2014;42(1):319-332.
12. Shi L, Chen H, Sun J, Li K. vCUDA: GPU-accelerated highperformance computingin virtual machines. *IEEE Trans Comput*. 2012;61(6):804-881.
13. Duato J, Pena AJ, Silla F, Mayo R, Quintana-Ort ES. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. Paper presented at: Proceedings of the 2010 International Conference on High Performance Computing and Simulation (HPCS); 2010:224-231.
14. Tien T, Yo Y. Enabling OpenCL support for GPGPU in kernel-based virtual machine. *Softw Pract Exp*. 2014;44(5):483-510.
15. Xen Project. VGA Passthrough. https://wiki.xen.org/wiki/Xen_VGA_Passthrough. Accessed Oct 25, 2019.
16. iPerf. http://www.iperf.fr. Accessed Oct 25, 2019.
17. Gleeson J, Kats D, Mei C, de Lara E. Crane: fast and migratable GPU passthrough for OpenCL applications. Paper presented at: Proceedings of the 10th ACM International Systems and Storage Conference; 2017:1-13.
18. Kehne J, Metter J, Bellosa F. GPUswap: enabling oversubscription of GPU memory through transparent swapping. *ACM SIGPLAN Not*. 2015;50(7):65-77.
19. Gelado I, Garland M. Throughput-oriented GPU memory allocation. Paper presented at: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming; 2019:27-37.
20. Li P, Hu X, Chen D, et al. LD: Low-overhead GPU race detection without access monitoring. *ACM Trans Archit Code Optimiz (TACO)*. 2017;14(1):1-25.
21. Xiao S, Balaji P, Zhu Q, et al. VOCL: an optimized environment for transparent virtualization of graphics processing units. Paper presented at: Proceedings of Innov Parallel Comput(InPar); 2012:1-12.
22. Xiao S, Balaji P, Dinan J, et al. Transparent accelerator migration in a virtualized GPU environment. Paper presented at: Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2012); 2012:124-131.
23. Prades J, Federico S. Turning GPUs into floating devices over the cluster: the beauty of GPU migration. Paper presented at: Proceedings of the 46th International Conference on Parallel Processing Workshops(ICPPW) 2017: 129-136.
24. InfiniBand trade association. www.infinibandta.org. Accessed October 25, 2019.
25. Yang Y, Mao B, Jiang H, Yang Y, Luo H, Wu S. Snapmig: accelerating VM live storage migration by leveraging the existing VM snapshots in the cloud. *IEEE Trans Parallel Distribut Syst*. 2018;29(6):1416-1427.
26. Deshpande U, Chan D, Chan S, Gopalan K, Bila N. Scatter-gather live migration of virtual machines. *IEEE Trans Cloud Comput*. 2015;6(1):196-208.
27. Ma J, Zheng X, Dong Y, et al. Gmig: efficient gpu live migration optimized by software dirty page for full virtualization. *ACM SIGPLAN Not*. 2018;53(3):31-44.

28. Melhem SB, Agarwal A, Goel N, Zaman M. Markov prediction model for host load detection and VM placement in live migration. *IEEE Access*. 2017;6:7190-7205.
29. Syed HJ, Gani A, Nasaruddin FH, Naveed A, Ahmed AIA, Khan MK. CloudProcMon: a non-intrusive cloud monitoring framework. *IEEE Access*. 2018;6:44591-44606.

How to cite this article: Kang JH, Lim JB, Yu HC. Partial migration technique for GPGPU tasks to Prevent GPU Memory Starvation in RPC-based GPU Virtualization. *Softw Pract Exper*. 2020;1–25. https://doi.org/10.1002/spe.2801