Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.061182

ARTICLE

GPU Usage Time-Based Ordering Management Technique for Tasks Execution to Prevent Running Failures of GPU Tasks in Container Environments

Joon-Min Gil[1], **Hyunsu Jeong**[1] and **Jihun Kang**[2,*]

[1]Department of Computer Engineering, Jeju National University, Jeju-do, 63243, Republic of Korea
[2]Department of Computer Science, Korea National Open University, Seoul, 03087, Republic of Korea

*Corresponding Author: Jihun Kang. Email: k2j23h@knou.ac.kr

Received: 19 November 2024; Accepted: 31 December 2024; Published: 17 February 2025

ABSTRACT: In a cloud environment, graphics processing units (GPUs) are the primary devices used for high-performance computation. They exploit flexible resource utilization, a key advantage of cloud environments. Multiple users share GPUs, which serve as coprocessors of central processing units (CPUs) and are activated only if tasks demand GPU computation. In a container environment, where resources can be shared among multiple users, GPU utilization can be increased by minimizing idle time because the tasks of many users run on a single GPU. However, unlike CPUs and memory, GPUs cannot logically multiplex their resources. Additionally, GPU memory does not support over-utilization: when it runs out, tasks will fail. Therefore, it is necessary to regulate the order of execution of concurrently running GPU tasks to avoid such task failures and to ensure equitable GPU sharing among users. In this paper, we propose a GPU task execution order management technique that controls GPU usage via time-based containers. The technique seeks to ensure equal GPU time among users in a container environment to prevent task failures. In the meantime, we use a deferred processing method to prevent GPU memory shortages when GPU tasks are executed simultaneously and to determine the execution order based on the GPU usage time. As the order of GPU tasks cannot be externally adjusted arbitrarily once the task commences, the GPU task is indirectly paused by pausing the container. In addition, as container pause/unpause status is based on the information about the available GPU memory capacity, overuse of GPU memory can be prevented at the source. As a result, the strategy can prevent task failure and the GPU tasks can be experimentally processed in appropriate order.

KEYWORDS: Cloud computing; container; GPGPU; resource management

1 Introduction

The graphics processing unit (GPU) of a cloud environment leverages resource-sharing to provide high-performance computing capabilities to multiple users. The GPU serves as a coprocessor of the central processing unit (CPU). It does not operate independently when performing a task, rather only engaging in the GPU computation part of the overall task. Thus, it is idle when not performing GPU tasks. However, such idling contributes to poor resource utilization. This problem can be addressed by sharing a single GPU among multiple users. In the cloud environment, a single GPU can indeed be shared by many users. Several GPU tasks run simultaneously on a single GPU. This increases GPU resource utilization and minimizes idle time.

In a cloud environment, computing resources are shared by multiple users, and a separate scheduler determines the order of resource usage so that each user equally shares the computing resources. This

prevents monopolization of resources by a particular user and eliminates the problem of performance imbalance among users. However, GPUs lack intrinsic mechanisms for resource-sharing. In a cloud environment wherein GPUs are shared, two problems arise when managing GPU tasks run by multiple users. The first is that there is no guarantee of equitable resource usage time, and the second is that GPU memory does not allow excessive use.

In a containerized environment with shared GPUs [1], GPU tasks running in the containers of various users engage the GPU in multiprocessing that consumes whatever resources are needed, up to the limit of the GPU resources. At this proceeds, the task of each user does not consider GPU resource usage by other users. In addition, traditional container management systems do not separately manage information on various GPU tasks. It is impossible to know which containers are using GPU resources and to what extent. Each container performs GPU operations independently. The first task to run preempts the GPU. In addition, as more targets share the GPU, resource contention occurs among the containers. In an environment running GPU tasks, such contention is not just a performance issue. In particular, GPU memory does not allow overuse [2]. If a new task runs out of GPU memory, that task will fail to load data into GPU memory for computation and will thus fail to execute.

In this context, effective management of GPU tasks is crucial to optimize resource-sharing and avoid failures. To do this, that is, to run multiple GPU tasks in a GPU-sharing container environment, it is necessary to manage the GPU usage time of each container and to prevent overutilization of GPU memory by an individual container to ensure that multiple containers equally share the GPU without task execution failures. In this paper, we propose a technique for managing the execution order of GPU tasks executed by multiple users in a GPU-sharing Docker-based container environment. This ensures that GPU time is shared equally among the users and prevents execution failures. The main contributions of this paper are as follows:

- This paper proposes a technique for managing the execution order of GPU tasks in a GPU-sharing container environment to ensure that multiple containers share the GPU equally.
- This paper proposes a monitoring technique to track information about GPU tasks and GPU usage time for each container.
- The proposed technique supports multiple tasks running simultaneously to prevent task execution failures due to insufficient GPU memory and to complete task execution safely.
- The proposed technique does not require any modifications to the code or framework of GPU tasks and provides transparency to the container user.
- In this paper, we verify the effectiveness of the proposed method through experiments in an environment where multiple containers are executing GPU tasks simultaneously.

This paper is organized as follows. Section 2 describes the background technology of the proposed strategy and Section 3 summarizes previous relevant research. Section 4 describes the problems caused by insufficient GPU memory in current container environments as revealed by experiments, and Section 5 details our proposed strategy. In Section 6, we verify its effectiveness experimentally. Finally, Section 7 concludes the paper and discusses future work.

2 Background

2.1 GPGPU Task

GPUs can engage in massively parallel processing using thousands of computer cores exploiting general-purpose computing on graphics processing unit (GPGPU) technology. This is particularly effective when processing large amounts of data, such as those of artificial intelligence (AI), and accelerates large-scale computations. GPUs use both GPU cores and GPU memory for computation. Thousands of GPU cores

in a GPU can handle large numbers of threads simultaneously, and GPU memory holds the data required for massively parallel processing. GPU cores access GPU memory only when reading data. The cores do not load data directly into the memory. Given these characteristics, GPU operations are executed in three phases. First, the GPU memory is input, followed by GPU computation and outputting of the result. During such operations, the data to be used are first loaded into the main memory and then copied into the GPU memory [3], from which the GPU core pulls. Fig. 1 shows the flow of a GPU operation.

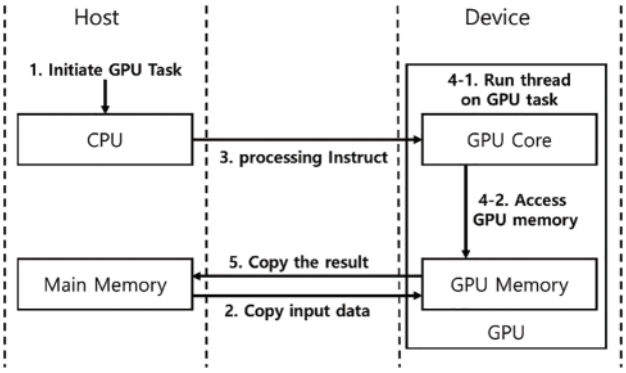

Figure 1: GPGPU task processing flow

When a GPU task runs out of GPU memory the task does not execute. To prevent such task failure in an environment wherein multiple GPU tasks run concurrently in a shared GPU environment, it is important to consider the available GPU memory.

2.2 Shared GPU in Container Environments

In a typical virtual memory (VM)-based cloud environment, GPUs are shared across multiple users using separate technologies. However, in a container-based cloud environment, it is relatively simple to share GPUs across multiple containers. In a containerized environment wherein multiple users share a GPU, any container with access to the GPU can use it to perform operations. The operations executed by each container are multi-processed on a single GPU, and no operation that can be executed by any container is forbidden.

Although multiple containers share a GPU, a single container sometimes occupies the entire GPU resource. Although the workload can be evenly handled, existing GPU resources do not allow logical multiplexing. Thus, GPU resources cannot be allocated in fixed shares to each container. Although certain commercial techniques allow GPU resource isolation in containerized environments, they work only with specific GPUs and require commercial software. This renders it impossible to prevent a particular container from monopolizing the GPU resources, the use of which may thus become contentious. If the containers are not isolated, GPU sharing is always associated with potential conflict.

Also, the container engine does not monitor the separate GPU operations. Container monitoring tools focus on container information per se. They cannot identify if a container is running GPU operations. GPU tasks can use GPU driver monitoring tools to track information about operations and resource usage, but such tools monitor only on a per-GPU basis. The tools cannot identify the container in which the GPU task is running. Currently, container and GPU tasks are monitored separately. It is necessary to integrate the monitoring information obtained by containers and GPU drivers to track the usage of GPU resources, especially the memory and usage time of each container.

3 Related Works

The sharing of GPUs across multiple users in cloud environments has been extensively studied. Container technology is associated with a minimal management layer overhead, rendering such technology appropriate for high-performance computations in a cloud environment [4]. In many cases, researchers have focused on per-device resource-sharing of GPUs in multi-GPU environments, not the sharing of a single GPU. Various resource optimization techniques have been proposed to improve resource utilization and minimize task latency when allocating GPU resources to multiple users and when controlling GPU sharing. Existing per-device GPU resource management techniques consider the multiple GPUs required to perform large-scale tasks in a clustered environment. Most GPU management techniques for multi-GPU-based cluster environments seek to solve the fragmentation problem that arises when the resources of many GPUs allocated to large tasks become scattered across the cluster.

The most traditional means of resource management in a cloud environment is resource optimization based on current state information. In the cloud environment, various forms of monitoring information can be managed. Runtime information is an important indicator of user resource needs. Such information reveals the frequency of task execution and the resource usage pattern. Resource management based on runtime information is traditionally used when managing GPU resources in a cloud environment. GPU resources are managed by deploying containers by the order of job requests [5]. This method has also been used to manage AI tasks [6–8] that actively use GPUs. In addition, techniques that employ resource-sharing and isolation [9] have been proposed. Here, we propose a technique that manages GPU usage time based on time slices. This applies to even a single GPU to ensure that users evenly share the GPU [10–13]. A similar runtime information-based approach has been used in VM-based cloud environments [14].

Existing AI-based resource management techniques include those that manage containers in Kubernetes [15–18], that schedule deep learning tasks to reduce the cost of cloud infrastructure in cloud environments [19], and that ensure that the service level objective (SLO) of each user is achieved [20,21]. Techniques that ensure quality of service (QoS) [21] in cluster environments have also been proposed. Some resource management techniques use AI to optimize GPU resource allocation. AI learns the resource usage patterns and the types of tasks solved on the server. AI-based resource allocation techniques collect runtime information and then analyze user task start and finish times, and resource usage patterns. Then they dynamically allocate resources to users.

To ensure fair sharing among multiple users in a cloud environment, some techniques support the pausing and later resumption of GPU tasks [22]. Such techniques typically modify the framework of GPU tasks and dynamically reallocate resources to paused tasks to improve throughput.

The API forwarding approach [23–25] is one of the main techniques used for GPU resource management. This optimizes memory allocation by intercepting API calls when the GPU task of a user is executed and then activating a modified API that includes resource management functions. Additionally, GPU resource fragmentation detection [26] has been studied in efforts to detect factors that do not directly manage the GPU but do compromise resource allocation and cause delays [27] attributable to resource heterogeneity.

Some GPU resource management techniques focus on prioritization rather than equal utilization of GPU resources. GPU tasks are prioritized based on the QoS [28,29] and deadlines [30] of each user. Other techniques find idle GPUs and then place tasks [31], or leverage the migration capacities of containers to access GPU servers in remote locations. GPU allocation is based on the location of the user in an edge-computing environment [32,33].

Other groups have sought to increase the GPU utilization rate with consideration of the energy required [34,35]. A runtime environment for scenarios in which GPUs are shared has been proposed [36].

Another study presented heterogeneous CPU/GPU scheduling [37]. The API-mediated GPU memory management system of TensorFlow was applied to the YARN cluster scheduling framework. Genetic algorithms have also been used to manage GPU resources [38].

Several commercial technologies manage GPU resources and tasks. Multiple GPU tasks are performed by a single GPU. Multi-Process Service (MPS) [39], a representative commercial technology, supports parallel processing of multiple GPU tasks, thereby improving the GPU utilization rate. However, this technology is a single user in nature. It cannot be applied to a multi-user GPU environment because information on the many-user tasks is lacking. Only information on overall GPU usage status is available to the cloud administrator. Other commercial methods manage GPU workloads in cluster environments [40]. Tasks are submitted to a workload manager, placed on available GPUs, and executed. However, this does not support the execution of multiple simultaneous tasks by the GPU, and lacks user transparency because all tasks are scheduled by the workload manager.

Various methods have been used to manage GPU resources in cloud environments. Most employ modifications of existing frameworks and APIs, thus increasing inter-system dependency and complexity. The methods cannot be used to manage small-scale tasks because they do not consider the sharing of a single GPU. In this paper, we seek to ensure user transparency, prevent task execution failure caused by GPU memory shortage without modifying the framework or source code of GPU tasks, and allow multiple users to equally share GPUs.

4 Motivation

In a GPU-sharing container environment, multiple GPU tasks running concurrently in multiple containers are handled as multi-processes. In this chapter, we perform two experiments in a traditional container environment with shared GPUs to understand the performance of multiple GPU tasks running concurrently and the issue of task failures due to GPU memory overflow. The first experiment measures the performance of each GPU task when multiple GPU tasks are running concurrently, and the second experiment identifies the problem of task failures that occur when the GPU runs out of memory.

In the first experiment, multiple containers execute GPU tasks in parallel on available GPU memory allows. In total, five containers are used and the execution times of all GPU tasks are measured. Each container employs 1.4 GB of GPU memory, and we increase the number of concurrent GPU tasks from 2 to 5 to identify the mutual performance impact. Each container performs a matrix multiplication operation. The experimental results are shown in Fig. 2.

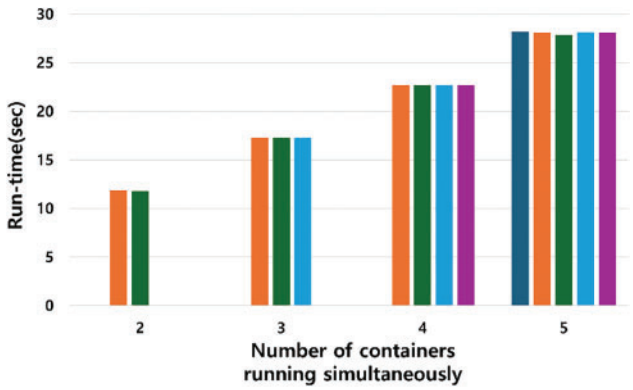

Figure 2: Performance on each simultaneous GPU task running within the allowable range of available GPU memory

When multiple GPU tasks are executed simultaneously, they are evenly processed within the limit of the GPU memory. As mentioned above, in a container environment that shares a GPU, the tasks for each container are multi-processed by the GPU and are handled fairly by the hardware scheduler. In addition, as the available GPU memory is adequate, all tasks are completed normally. There is no execution failure. Thus, when the GPU memory is not exceeded, multiple GPU tasks running concurrently do not compromise mutual performance and improve resource utilization.

The following experiment verifies the problem caused by low GPU memory. In this experiment, 19 GB of GPU memory is pre-occupied by other processes to quickly create a GPU memory shortage, and then GPU tasks are simultaneously executed in five containers. The results are shown in Table 1.

Table 1: Execution times and completion/failure rates of GPU tasks when GPU memory is overused

Container number	Execution time (sec)	Execution complete
#1	7.23	X
#2	10.95	X
#3	17.5	O
#4	7.50	X
#5	16.97	O
#6	17.3	O

The performance differences for GPU tasks that execute completely are minimal, but some containers fail to run because of inadequate GPU memory. In general, the performance of CPU-based computational tasks degrades as resources become scarce. The main memory tolerates overuse because the secondary storage serves as virtual memory. Performance degradation is apparent, but the task does not fail. However, GPU tasks do not allow excessive use of GPU memory. When such memory runs out, performance degradation is extreme. In other words, task execution fails. This is a major problem. Especially when task execution is automated, the failure of one task affects other tasks that will run later.

The simplest method that could be used to prevent GPU tasks from failing when GPU memory is low is to allocate a separate GPU to each container. However, GPU resources are not as scalable as CPU or main memory. Only a limited number of GPUs can be installed on a single server. Thus, assigning a separate GPU to each container limits the number of containers that can run on a single server. This is associated with underutilization of the GPUs per se as they sit idle when GPU tasks are not running, and also underutilization of the CPU and main memory as the number of containers running on the server decreases.

We propose a GPU task management technique that avoids GPU memory runout. The method determines the order of GPU usage based on the container GPU usage time. This solves the problems described above. The proposed technique is described in detail in the following sections.

5 Implementation

In a containerized environment wherein multiple users share a single GPU, the GPU tasks executed by each user behave as multi-processes. In addition, as the Docker engine does not individually manage information on the GPU tasks running in each container, a monitoring technique is needed to collect both the startup status and the GPU usage time of the tasks of each container. Then it becomes possible to manage the GPU usage time of each container. We propose a monitoring technique that measures the GPU usage time of each container and then manages the execution order of GPU tasks. A task execution delay technique

is employed to adjust the task execution order based on prior GPU usage time. The overall structure of the system is shown in Fig. 3.

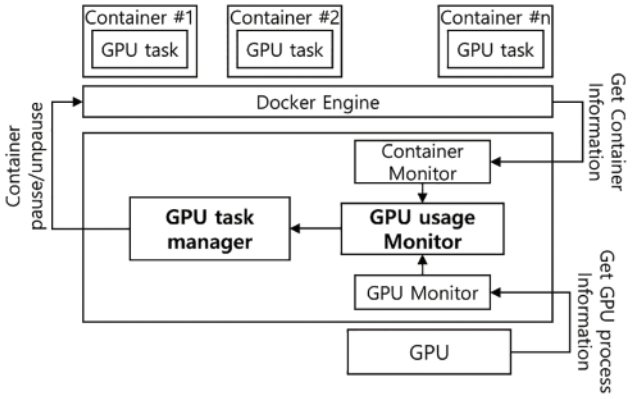

Figure 3: Overall structure

There are two main subsystems. The first is a GPU utilization monitor that detects the GPU processes running in each container, measures the usage time based on the start and end times of the tasks, and records the amount of available GPU memory based on the memory usage of each container. The second subsystem is the GPU Task Manager. This determines whether and in what order to run GPU tasks based on the monitoring information.

To measure the GPU usage time for each container, we need to determine if the container is using the GPU, that is, whether the container is running a GPU task. As described above, the container monitoring tool [41] is managed by the container engine, and the GPU monitoring tool [42] is managed by the GPU driver. This allows the container engine to monitor the tasks running in each container, but the engine cannot detect if the tasks running in the container are GPU tasks. The GPU monitoring tool performs only GPU-wide monitoring, so it cannot identify the container in which the current GPU task is running. This renders it difficult to manage the GPU usage time of each container and the execution order of GPU tasks.

Our proposed GPU utilization monitoring technique integrates the container monitoring information with the GPU driver monitoring information. As described above, the container engine can identify which tasks are running in each container. For each container, the engine looks up the process ID of the running task, and the GPU monitoring tool looks up the process ID of the GPU task that is currently running. A comparison of the process IDs collected by each monitoring tool reveals the container running the GPU task and maps the GPU process ID to the container ID. This aids management. The monitoring information identifies the containers running GPU tasks and measures the GPU memory usage of each container.

Current monitoring techniques used in traditional cloud environments collect information at regular intervals, but our technique is always-on because it must determine when GPU tasks start and end. As we do not know when a container will run a GPU task, monitoring must continue even when a GPU task is not running. When the GPU usage monitoring tool detects the execution of a GPU task, it finds the container running the GPU task and records the GPU usage time of that container and the current GPU memory usage. The method is represented by Algorithm 1.

Algorithm 1: Monitoring GPU usage time for each container

```
1  GPU_Process  gpu_list[MAX_PROCESSES];
2  int  gpu_count   = 0;
3  while (1) {
4         Time now = get_current_time();
5         GPU_Process  current_gpus[MAX_PROCESSES];
6         int  current_count   = get_current_gpu_processes(current_gpus);
7         for (int  i = 0; i < current_count; i++) {
8               int  pid   = current_gpus[i].pid;
9               int  idx   = find_process(pid, gpu_list, gpu_count);
10              if (idx   == –1) {
11                    gpu_list[gpu_count++] = {pid, current_gpus[i].name, now, NULL};
12              } else {
13                    gpu_list[idx].end_time   = NULL;
14              }
15        }
16        for (int  i = 0; i < gpu_count; i++) {
17              if (!is_in_current_list(gpu_list[i].pid, current_gpus, current_count) && !gpu_list[i].end_time)
                {
18                  gpu_list[i].end_time   = now;
19              }
20        }
21        Docker_Container  containers[MAX_CONTAINERS];
22        int  container_count   = get_running_containers(containers);
23        for (int  i = 0; i < container_count; i++) {
24              int  pids[MAX_PIDS], pid_count   = get_container_pids(containers[i].id, pids);
25              for (int  j = 0; j < gpu_count; j++) {
26                    if (is_pid_in_list(gpu_list[j].pid, pids, pid_count)) {
27                          Time end = gpu_list[j].end_time  ?  gpu_list[j].end_time: now;
28                    }
29              }
30        }
31 }
```

As shown in Algorithm 1, monitoring operates continuously, not at distinct intervals. This allows real-time determination of whether GPU tasks are being executed or not. It is impossible to know in advance when a user GPU task will be executed. In addition, as the data input operation of GPU memory is relatively fast, that input might be completed within the between-execution interval of an intermittent monitoring operation, rendering the container evaluation meaningless. The system must always run to detect, as quickly as possible, when the GPU task starts. As monitoring is continuous, some computing resources are required, but this does not significantly impact the system. This is explained in detail in the experiments below.

The information gathered by the monitoring technique is used to determine the execution order of GPU tasks. As demonstrated by the experimental results above, within the tolerance of GPU memory, the performance impact of multiple GPU tasks running concurrently on a GPU is insignificant. Therefore, to

increase GPU utilization while preventing GPU task execution failure, it is important to have as many GPU tasks as possible running concurrently while avoiding GPU memory runout.

The basic operation of our proposed method is deferred processing of GPU tasks based on the available GPU memory and the GPU usage time. The system decides whether to execute a task based on the amount of available GPU memory. If multiple tasks are in competition, the method decrees the order of GPU utilization based on the usage time of each container. To determine the order of usage via lazy processing of tasks, it must be possible to pause and unpause tasks that have commenced. Pausing tasks prevents memory runout, and unpausing adjusts the task execution order based on the usage time of each container. However, once tasks commence, their execution order cannot be changed between GPU tasks. Also, when using GPU cores for GPU task processing, tasks cannot be paused during execution. This renders it very difficult to pause the task per se. Given these considerations, we do not directly throttle the execution of such operations, but rather indirectly defer them by pausing containers. We use the cgroup [43] function freezer [44] to pause and unpause containers, and thus indirectly pause GPU tasks. The pure pause and unpause operations, excluding the waiting time caused by container suspension, do not significantly impact the operation execution time because the execution times of the two operations are only 0.06 s.

As explained above, the execution order of GPU tasks cannot be externally adjusted arbitrarily. This means that methods such as context switching and swapping are inappropriate. In addition, the cloud administrator cannot modify a user GPU task code, ensuring user transparency.

As described above, the first step of a GPU task is the input of data to be used in computation into GPU memory. During this time, the GPU task occupies GPU memory and prepares for the task. If the container is paused after the memory is occupied, the memory is wasted because it is occupied but the task is paused. Such memory remains occupied until release after the task is completed. The occupied area cannot be converted into an area available to other containers. Also, once a computational operation using GPU cores begins, control of the GPU task is completely transferred to the GPU. Thus, GPU computation continues even if the container is paused. Therefore, containers must be paused as soon as the start of a GPU operation is detected to ensure that each container minimizes its GPU memory footprint. This is why the monitoring tasks described above must always run. Our proposed method for GPU task execution order management is represented by Algorithm 2.

Algorithm 2: GPU task manager

```
1  typedef  struct  {
2  char container_id [256];
3  int  gpu_usage_time;
4  int  paused_time;
5  } Container_GPU_Status;
6  Container_GPU_Status  container_gpu_status[MAX_CONTAINERS];
7  for (int  i = 0; i < container_count; i++) {
8         if (is_container_working_on_gpu(containers[i].id)) {
9                pause_container(containers[i].id);
10                int sufficient_memory   = 0;
11                for (int  j = 0; j < gpu_count; j++)
12                      if (gpus[j].available_memory >
                                                  REQUIRED_MEMORY_THRESHOLD) sufficient_memory = 1;
13                if (sufficient_memory) resume_container(containers[i].id);
```

(Continued)

Algorithm 2 (continued)

```
14        }
15  }
16  while (1) {
17        int paused_count  = 0, min_gpu_usage_time  = INT_MAX, min_container_index  = −1;
18        for (int  i = 0; i < container_count; i++) {
19              if (is_container_paused(containers[i].id)) {
20                    paused_count++;
21                    int  effective_gpu_usage_time
                                  = container_gpu_status[i].gpu_usage_time  -
                                    container_gpu_status[i].paused_time;
22                  if (effective_gpu_usage_time  < min_gpu_usage_time) {
23                      min_gpu_usage_time  = effective_gpu_usage_time;
24                      min_container_index  = i;
25                    }
26              }
27      }
28      if (paused_count  == 0) break;
29      if (min_container_index  ! = −1) {
30          resume_container(containers[min_container_index].id);
31          container_gpu_status[min_container_index].paused_time  = 0;
32       }
33  }
```

As our method preemptively pauses GPU operations of containers when such operations start and then determines whether the operations are viable, containers become paused with very small footprints of GPU memory. Thus, we exclude the time for which a container is paused from its GPU usage time. We measure the GPU usage time of each container and determine whether the GPU is available in that container or not. When choosing between paused containers that can be used to continue GPU operations, we check the free GPU memory level and then unpause the pending container. If there is more than one pending container, we unpause them one by one in order to decrease GPU usage, with the first unpaused container completing its GPU memory allocation before the next is unpaused. This prevents task failure caused by GPU memory overuse.

As GPU usage time is compared between containers to determine which container to run only when there is more than one container waiting to use the GPU, containers with high GPU usage times will be immediately unpaused if they have no competitors, minimizing task delays. Even if a new GPU task is executed by a container with less GPU time than the restarted container, the restarted task will run until it is completed. This is because resuspension when GPU memory is occupied does not free up that memory, and the resumed GPU task will likely increase memory occupancy even if that task is later resuspended. This memory is freed after the first task is allowed to complete as quickly as possible without interfering with the execution of the second task.

Overall, this process ensures that each container fairly shares the GPU and prevents task failure by delaying the execution of tasks by reference to the availability of GPU memory. The effectiveness of this method is demonstrated through experiments as described below.

6 Evaluation

This section, we describe experiments used to verify the effectiveness of our strategy. The experimental environment is shown in Table 2.

Table 2: Experimental environment

Type	Specification
CPU	i9-10920X (3.5 GHz, 12 Cores, 24 Threads)
Memory	DDR4 128 GB
GPU	Nvidia RTX 3090 (24 GB Memory)
OS	Ubuntu 18.04
Docker version	NVIDIA Docker2

The experiments use six containers to run simultaneous multiple matrix multiplication tasks on a GPU at regular intervals. Each GPU task of each container uses 1.4 GB of GPU memory, and 19 GB of such memory is always occupied. Thus, the memory resource is rapidly exhausted. If six containers operate concurrently, the tasks of up to three containers can run, but the remaining tasks will run out of memory. The experimental results are shown in Fig. 4.

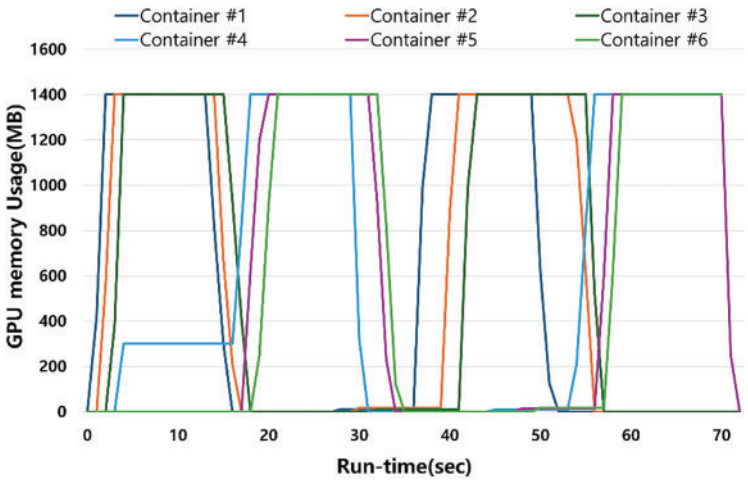

Figure 4: Delay processing of GPU tasks using our strategy to eliminate GPU memory overuse, and the GPU memory usage pattern and performance of each container

Initially, all containers exhibit zero GPU memory usage time. Their process IDs thus determine the execution order. All containers are paused once and then run when memory is available. This approach ensures that all tasks run normally, without failure. At around 1 s, container #4 exhibits relatively low memory usage because it is paused to prevent further allocation of memory via lazy processing. This helps prevent memory overuse. When container #1 completes its work and free memory thus becomes available, container #3 is unpaused and resumes allocation of memory. Thereafter, tasks are unpaused in the order of container use, and execution continues.

As the experimental results show, our method prevents task failure caused by GPU memory runout. As every container is paused once it starts a task, the execution time increases slightly but does not significantly

impact the overall performance and is an acceptable overhead because it prevents task failure. The tasks use memory in an order that is based on how long a container has been employing the GPU. All containers share the GPU equally.

Our monitoring technique tracks the usage time and memory usage of each container, integrating information collected from the container engine and the GPU driver. Monitoring is always on because it is essential to detect the execution of tasks. In the next experiment, we measure the resource use of our strategy and its impact on the overall system. The results are shown in Fig. 5.

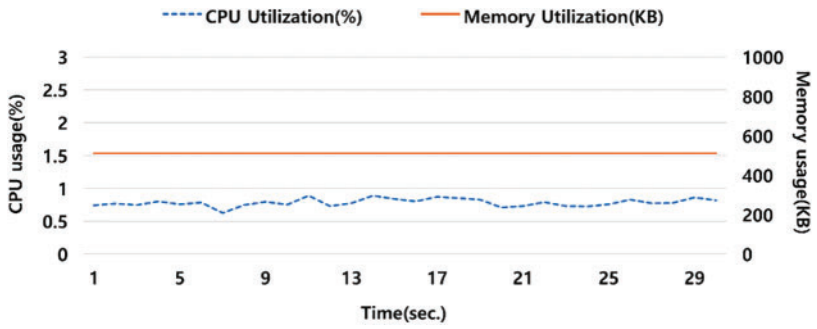

Figure 5: Resource usage by our proposed monitoring techniques

As shown in Fig. 5, our monitoring technique uses only a very small amount of computing resources, about 1% of the total resources of the experimental server, although the system is always on. The overall system impact is very small. The GPU work of the containers is not affected because the monitoring tasks do not use GPU resources.

The experiments show that our method allows each container to use the GPU in an orderly manner based on the available GPU memory, thus preventing its excessive use. In traditional container environments, GPU sharing among multiple users is not considered, so when memory is overused, new GPU tasks that are running fail to execute. However, our strategy effectively prevents such failure. We also verify that the always-on monitoring employed to measure the usage time of a container has little impact on the system. Our method prevents GPU task failure and increases GPU utilization without significantly impacting the overall system.

7 Conclusions and Future Work

We proposed a GPU task execution order management technique that determines the GPU usage order of GPU tasks. This prevents task failure because of insufficient GPU memory in a container environment that shares a single GPU. Our technique ensures that containers with less prior GPU usage time can access the GPU faster when resource demand is high, and prevents tasks from failing due to excessive memory usage by determining whether to execute GPU tasks based on prior container GPU memory usage. The proposed technique uses a deferred processing approach based on the amount of available GPU memory to prevent the overuse of such memory. If there is enough GPU memory available, all commenced GPU tasks are executed in parallel. However, if memory is over-utilized, task execution is delayed, preventing task failure caused by insufficient memory. We also verify that GPU usage can be managed relatively fairly by unpausing containers that are paused due to insufficient memory based on the prior usage time of each container at the time when GPU memory becomes available. This technique has the potential to significantly improve the efficiency

and reliability of GPU utilization in containerized environments, paving the way for more robust multi-user cloud computing systems.

Our proposed technique determines the order of GPU utilization based on the usage time of GPU resources. Usage time accumulates for each container. However, the present continuous accumulation method may unbalance resource utilization because it keeps information that is too old. In particular, as GPUs are only used when GPU operations are executed, monitoring information can be inaccurate if the frequency of operations is not consistent. To avoid this, a criterion that manages the freshness of GPU usage time is needed. Our future work will focus on formulating a cumulative criterion for GPU usage time, ensuring equitable resource allocation over extended periods to maintain such freshness. In addition, we will consider a multi-GPU environment and offload tasks to remote computing resources or use clustering techniques [45] that consider the execution patterns of each task and container locality. This will extend GPU resource management technology for a single server to an entire cluster server area.

Acknowledgement: None.

Funding Statement: This research was supported by "Regional Innovation Strategy (RIS)" through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (MOE) (2023RIS-009).

Author Contributions: The authors confirm contribution to the paper as follows: study conception and design: Joon-Min Gil, Jihun Kang; data collection: Jihun Kang; analysis and interpretation of results: Joon-Min Gil, Hyunsu Jeong, Jihun Kang; draft manuscript preparation: Joon-Min Gil, Jihun Kang. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: Not applicable.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

Abbreviations

GPU	Graphic Processing Unit
GPGPU	General Purpose Computing on Graphic Processing Unit

References

1. NVIDIA Docker [cited 2024 Nov 17]. Available from: https://github.com/NVIDIA/nvidia-docker.
2. CUDA [Online] [cited 2024 Nov 17]. Available from: https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html.
3. CUDA memory API [Online] [cited 2024 Nov 17]. Available from: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html.
4. Martin JP, Kandasamy A, Chandrasekaran K. Exploring the support for high performance applications in the container runtime environment. Hum Centric Comput Inf Sci. 2018 Jan;8(1):1–15. doi:10.1186/s13673-017-0124-3.
5. El Haj Ahmed G, Gil-Castiñeira F, Costa-Montenegro E. KubCG: a dynamic Kubernetes scheduler for heterogeneous clusters. Softw: Pract Exper. 2021;51(2):213–34. doi:10.1002/spe.2898.
6. Tsung-Tso H, Che-Rung L. Voda: a GPU scheduling platform for elastic deep learning in kubernetes clusters. In: Proceedings of the 2023 IEEE International Conference on Cloud Engineering (IC2E); 2023; Boston, MA, USA. p. 131–40. doi:10.1109/IC2E59103.2023.00023.
7. Liu Z, Chen C, Li J, Cheng Y, Kou Y, Zhang D. KubFBS: a fine-grained and balance-aware scheduling system for deep learning tasks based on Kubernetes. Concurr Comput. 2022;34(11):e6836. doi:10.1002/cpe.6836.

8. Chen HM, Chen SY, Hsueh SH, Wang SK. Designing an improved ML task scheduling mechanism on kubernetes. In: 2023 Sixth International Symposium on Computer, Consumer and Control (IS3C); 2023; Taichung, Taiwan. p. 60–3. doi:10.1109/IS3C57901.2023.00024.
9. Shen W, Liu Z, Tan Y, Luo Z, Lei Z. KubeGPU: efficient sharing and isolation mechanisms for GPU resource management in container cloud. J Supercomput. 2023;79(1):591–625. doi:10.1007/s11227-022-04682-2.
10. Gu J, Zhu Y, Wang P, Chadha M, Gerndt M. FaST-GShare: enabling efficient spatio-temporal GPU sharing in serverless computing for deep learning inference. In: Proceedings of the 52nd International Conference on Parallel Processing; 2023; New York, NY, USA. p. 635–44. doi:10.1145/3605573.3605638.
11. Liu L, Yu J, Ding Z. Adaptive and efficient GPU time sharing for hyperparameter tuning in cloud. In: Proceedings of the 51st International Conference on Parallel Processing; 2022; New York, NY, USA. p. 1–11. doi:10.1145/3545008.3545027.
12. Peng K, Bohai Zhao MB, Xu X, Nayyar A. QoS-aware cloud-edge collaborative micro-service scheduling in the IIoT. Hum Centric Comput Inf Sci. 2023;13(28):173–84. doi:10.22967/HCIS.2023.13.028.
13. Kemchi S, Zitouni A, Djoudi M. AMACE: agent based multi-criterions adaptation in cloud environment. Hum Centric Comput Inf Sci. 2018;8(26):1–28. doi:10.1186/s13673-018-0149-2.
14. Alresheedi SS, Lu S, Abd Elaziz M, Ewees AA. Improved multi objective salp swarm optimization for virtual machine placement in cloud computing. Hum Centric Comput Inf Sci. 2019;9(15):1–24. doi:10.1186/s13673-019-0174-9.
15. Harichane I, Makhlouf SA, Belalem G. KubeSC-RTP: smart scheduler for Kubernetes platform on CPU-GPU heterogeneous systems, Concurrency and Computation. Pract Exper. 2022;34(21):e7108. doi:10.1002/cpe.7108.
16. Thinakaran P, Gunasekaran JR, Sharma B, Kandemir MT, Das CR. Kube-knots: resource harvesting through dynamic container orchestration in gpu-based datacenters. In: 2019 IEEE International Conference on Cluster Computing; 2019; Albuquerque, NM, USA. p. 1–13. doi:10.1109/CLUSTER.2019.8891040.
17. Albahar H, Dongare S, Du Y, Zhao N, Paul AK, Butt AR. SchedTune: a heterogeneity-aware GPU scheduler for deep learning. In: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid); 2022; Taormina, Italy. p. 695–705. doi:10.1109/CCGrid54584.2022.00079.
18. Motavaselalhagh F, Safi Esfahani F, Arabnia HR. Knowledge-based adaptable scheduler for SaaS providers in cloud computing. Hum Centric Comput Inf Sci. 2015;5(16):1–19. doi:10.1186/s13673-015-0031-4.
19. Lim J. Versatile cloud resource scheduling based on artificial intelligence in cloud-enabled fog computing environments. Hum Centric Comput Inf Sci. 2023;13(53). doi:10.22967/HCIS.2023.13.054.
20. Lou J, Sun Y, Zhang J, Cao H, Zhang Y, Sun N. ArkGPU: enabling applications' high-goodput co-location execution on multitasking GPUs. CCF Transact High Perf Comput. 2023;5(3):304–21. doi:10.1007/s42514-023-00154-y.
21. Yeung G, Borowiec D, Yang R, Friday A, Harper R, Garraghan P. Horus: interference-aware and prediction-based scheduling in deep learning systems. IEEE Trans Parallel Distrib Syst. 2021;33(1):88–100. doi:10.1109/TPDS.2021.3079202.
22. Wang S, Gonzalez OJ, Zhou X, Williams T, Friedman BD, Havemann M, et al. An efficient and non-intrusive GPU scheduling framework for deep learning training systems. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis; 2020; Atlanta, GA, USA. p. 1–13. doi:10.1109/SC41405.2020.00094.
23. Yeh TA, Chen HH, Chou J. KubeShare: a framework to manage GPUs as first-class and shared resources in container cloud. In: Proceedings of the 29th International Symposium on High-performance Parallel and Distributed Computing; 2020; Stockholm, Sweden. p. 173–84. doi:10.1145/3369583.3392679.
24. Lee M, Ahn H, Hong CH, Nikolopoulos DS. gShare: a centralized GPU memory management framework to enable GPU memory sharing for containers. Future Gener Comput Syst. 2022;130(9):181–92. doi:10.1016/j.future.2021.12.016.
25. Benedicic L, Cruz FA, Madonna A, Mariotti K. Sarus: highly scalable docker containers for HPC systems. In: High Performance Computing: ISC High Performance 2019 International Workshops; 2019; Frankfurt, Germany. p. 46–60. doi:10.1007/978-3-030-34356-9_5.

26. Weng Q, Yang L, Yu Y, Wang W, Tang X, Yang G, et al. Beware of fragmentation: scheduling GPU-sharing workloads with fragmentation gradient descent. In: 2023 USENIX Annual Technical Conference (USENIX ATC 23); 2023 Jul; Boston, MA, USA. p. 995–1008.
27. Li K, Wang H, Mu X, Chen X, Shin H. Dynamic logical resource reconstruction against straggler problem in edge federated learning. Human-Centric Comput Inform Sci. 2024;14(25):1–22. doi:10.22967/HCIS.2024.14.025.
28. Muniswamy S, Vignesh R. DSTS: a hybrid optimal and deep learning for dynamic scalable task scheduling on container cloud environment. J Cloud Comput. 2022;11(1):33. doi:10.1186/s13677-022-00304-7.
29. Mao Y, Yan W, Song Y, Zeng Y, Chen M, Cheng L, et al. Differentiate quality of experience scheduling for deep learning inferences with docker containers in the cloud. IEEE Trans Cloud Comput. 2022;11(2):1667–77. doi:10.1109/TCC.2022.3154117.
30. Zou A, Li J, Gill CD, Zhang X. RTGPU: real-time GPU scheduling of hard deadline parallel tasks with fine-grain utilization. IEEE Trans Parallel Distrib Syst. 2023;34(5):1450–65. doi:10.1109/TPDS.2023.3235439.
31. Chen Z, Zhao X, Zhi C, Yin J. DeepBoot: dynamic scheduling system for training and inference deep learning tasks in GPU cluster. IEEE Trans Parallel Distrib Syst. 2023;34(9):2553–67. doi:10.1109/TPDS.2023.3293835.
32. Kennedy J, Sharma V, Varghese B, Reaño C. Multi-tier GPU virtualization for deep learning in cloud-edge systems. IEEE Trans Parallel Distrib Syst. 2023;34(7):2107–23. doi:10.1109/TPDS.2023.3274957.
33. Gong Y, Bian K, Hao F, Sun Y, Wu Y. Dependent tasks offloading in mobile edge computing: a multi-objective evolutionary optimization strategy. Future Gener Comput Syst. 2023;148(1):314–25. doi:10.1016/j.future.2023.06.015.
34. Filippini F, Anselmi J, Ardagna D, Gaujal B. A stochastic approach for scheduling AI training jobs in GPU-based systems. IEEE Trans Cloud Comput. 2023;12(1):53–69. doi:10.1109/TCC.2023.3336540.
35. Wang X, Li Y, Guo F, Xu Y, Lui JC. Dynamic GPU scheduling with multi-resource awareness and live migration support. IEEE Trans Cloud Comput. 2023;11(3):3153–67. doi:10.1109/TCC.2023.3264242.
36. Cai Z, Chen Z, Ma R, Guan H. SMSS: stateful model serving in metaverse with serverless computing and GPU sharing. IEEE J Sel Areas Commun. 2023;42(3):799–811. doi:10.1109/JSAC.2023.3345401.
37. Shi J, Chen D, Liang J, Li L, Lin Y, Li J. New YARN sharing GPU based on graphics memory granularity scheduling. Parallel Comput. 2023;117(6):103038. doi:10.1016/j.parco.2023.103038.
38. Jin X, Hu J, Wang J, Zhang S. A profit-aware double-layer edge equipment deployment approach for cloud operators in multi-access edge computing. Hum Centric Comput Inf Sci. 2024;14(23):1–23. doi:10.22967/HCIS.2024.14.023.
39. NVIDIA Multi-Process Service [Online] [cited 2024 Dec 02]. Available from: https://docs.nvidia.com/deploy/mps/index.html.
40. SLRUM [Online] [cited 2024 Dec 02]. Available from: https://slurm.schedmd.com/https://docs.nvidia.com/deploy/mps/index.html.
41. Docker, docker ps [Online] [cited 2024 Nov 17]. Available from: https://docs.docker.com/engine/reference/commandline/ps/ https://docs.nvidia.com/deploy/mps/index.html.
42. NVIDIA NVIDIA system management interface [Online] [cited 2024 Nov 17]. Available from: https://developer.nvidia.com/management-library-nvml.
43. The kernel development community, cgroups [Online] [cited 2024 Nov 17]. Available from: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.
44. The kernel development community, cgroups freezer [Online] [cited 2024 Nov 17]. Available from: https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt.
45. Wang C, Li L, Hao F. Deep attention fusion network for attributed graph clustering. In: IEEE Transactions on Computational Social Systems; 2024; Yinchuan, China. p. 380–5.