International Journal on Advanced Science Engineering Information Technology

Vol.14 (2024) No. 5
ISSN: 2088-5334

Pod Placement Techniques to Avoid Job Failures Due to Low GPU Memory in a Kubernetes Environment with Shared GPUs

Jihun Kang [a], Hwamin Lee [b] Daewon Lee [c,*]

[a] *Department of Computer Science, Korea National Open University, 86, Daehak-ro, Jongno-gu, Seoul, Republic of Korea*
[b] *Department of Biomedical Informatics, Korea University College of Medicine, 46, Gaeunsa 2-gil, Seongbuk-gu, Seoul, Republic of Korea*
[c] *Department of Electronics Computer Engineering, Seokyeong University, 124, Seogyeong-ro Seongbuk-gu, Seoul, Republic of Korea*
[*] *Corresponding author: daelee@skuniv.ac.kr*

***Abstract*— In a container-based cloud environment, GPUs have the advantage of providing high-performance computation to multiple users, and through GPU sharing, many GPU container users can be accommodated over the number of physical GPUs. This increases resource utilization and minimizes idle time. However, extended resources used to share GPUs in Kubernetes do not partition GPU resources or limit usage and only logically increase the number of GPUs that can be recognized. Therefore, usage limits and equal use of GPU resources cannot be guaranteed among pods sharing GPUs. Additionally, GPU memory generally does not allow for overuse. As a result, if a pod with high GPU memory usage runs a GPU task, it will be unable to limit GPU memory usage and free up available GPU memory. Data must be loaded into the GPU memory to perform the GPU task. However, if the data to be used for computation cannot be loaded into the GPU memory due to insufficient GPU memory, the pod will not be able to start the task and will fail to execute. This paper proposes a pod placement technique to avoid GPU memory shortage when sharing GPUs between pods in the Kubernetes environment. The proposed technique monitors the GPU memory usage and usage frequency of each worker node that makes up the cluster and places the pod on the worker node with the most available GPU memory based on the monitoring information.**

***Keywords*— Cloud; GPU computing; resource management; pod placement.**

Manuscript received 11 Dec. 2023; revised 29 Mar. 2024; accepted 18 Aug. 2024. Date of publication 31 Oct. 2024.

I. INTRODUCTION

In a GPU-based high-performance cloud environment, the environment has the advantage of providing high-performance computing services to users over a network. Based on the characteristics of cloud environments where multiple users share computing resources, sharing GPUs with multiple users enables a single GPU to be provided to various users, thereby increasing the resource utilization rate of the GPU and minimizing idle time. In container-based cloud environments [1], containers sharing a GPU can utilize the full capabilities of the GPU simultaneously without the need for additional commercial technology.

In a container environment, the ability of each container to utilize the full capabilities of the GPU provides the advantage of not being limited in the use of GPU resources. However, in a GPU-sharing environment, it is not possible to limit the GPU resource usage of each container resulting in resource contention. GPUs have minimal resource scaling, unlike CPUs and memory. Because the number of GPUs that can be installed on a server is physically restricted, it is relatively complex to acquire additional resources, sometimes requiring the addition of the server itself.

When performing GPU computations, GPU tasks can be waited by the scheduler if no GPU cores are available, but GPU memory is generally not allowed to be overused [2]. Also, due to the nature of using GPU cores to process tasks, data must be ready in GPU memory, so if a new task runs out of GPU memory, it will fail to load data into GPU memory, and the task will fail to run. When sharing GPU resources, Kubernetes [3], a container-based cluster system that manages available GPUs as a virtual number of resources through extended resources manages GPUs as a virtual number of resources without partitioning them, and places pods on servers with the highest number of available virtual resources.

In Kubernetes, when a Pod is allocated resources, it can be logically partitioned to allocate only a portion of the resources, such as CPU or memory. GPUs, however, can only be allocated on a per-device basis unless use a separate commercial technology, and it is not possible to partially

allocate a single GPU a fraction of its total resources. Virtual GPUs recognized as multiple via extended resources do not logically partition resources and allocate a portion of the GPU's resources but only provide the ability to limit the number of pods that can share the GPU simultaneously. This method alone cannot limit each Pod's GPU usage or allocate only a portion of GPU resources. As a result, if a particular Pod uses most of the GPU memory, new Pods can still be deployed if there are enough extended resources. However, the Pods deployed on the server will fail to execute GPU jobs because no actual GPU memory is available.

In this paper, we propose a pod placement technique that determines the server where a pod is placed based on the GPU memory usage of each server in the cluster and the frequency of GPU usage when placing a pod requesting GPU resources in a Kubernetes environment. The proposed technique solves the GPU memory shortage problem caused by the existing pod placement technique in Kubernetes, which only considers the number of extended resources without considering the GPU memory usage. It prevents the pods from failing to perform GPU tasks.

II. Material And Method

A. Motivation

With extended resources, Kubernetes recognizes a single GPU as if it were multiple GPUs. GPU sharing technology allows multiple users to use a GPU simultaneously, and users using the GPU concurrently share GPU resources to accomplish their tasks. However, unlike CPUs or main memory, GPUs cannot partition their resources logically. As a result, each user sharing a GPU is not allocated a portion of the GPU's resources but instead uses the entire GPU without limits. Limiting each user's GPU usage in a cloud environment requires the use of commercial products, which have limitations that only work with specific GPUs. These characteristics can lead to job failures due to contention for GPU resources, depending on the size of the jobs of the users sharing the GPU.

As explained earlier, extended resources are used when sharing GPUs in the Kubernetes environment. Extended resources do not divide the GPU but increase the number of devices recognized by the system and, as a result, only limit the maximum number of pods that can be used simultaneously by sharing the GPU. Because of this, even if a specific pod sharing the GPU uses most of the GPU resources, a new pod can be deployed if the number of resources divided as extended resources remains. Because of this, if the pod deployed first uses most of the GPU, a new pod can be deployed. Even if we try to run a task using GPU, the task execution will fail because there are no available GPU resources.

GPU task execution failures due to insufficient GPU resources occur primarily due to inadequate GPU memory. GPU cores, where multiple tasks can be executed simultaneously, and their execution order can be adjusted by the GPU's internal scheduler depending on the availability of GPU cores. GPU memory does not operate in the same way. To perform computations using GPU cores, the data to be processed must be loaded into GPU memory. Additionally, GPU memory typically does not allow for overuse. While primary memory may utilize secondary storage in case of overuse, GPU memory does not have this capability.

While there exists functionality to utilize main memory as auxiliary storage to expand GPU memory, this feature needs to be added at the level of GPGPU programming code for allocating GPU memory [4]. However, AI frameworks like TensorFlow, which provides GPU computation capabilities, typically do not offer this feature by default. As a result, if a new GPU task is initiated while GPU memory is already in use beyond capacity, the task execution fails due to the inability to allocate GPU memory.

Fig. 1 shows the performance of the inference task on the pods. The experiments measured the performance and success of task execution of inference tasks implemented with TensorFlow [5]. The dataset was MNIST with a batch size set to 32, and 60,000 data points were used for inference tasks. In Fig. 1, the leftmost bar shows the performance of a single run for performance comparison. As shown in Fig. 1, as the number of pods running inference tasks increases, the performance of inference tasks deteriorates. The inference task in the experiment causes performance degradation when run simultaneously on up to 23 pods but can be completed typically.

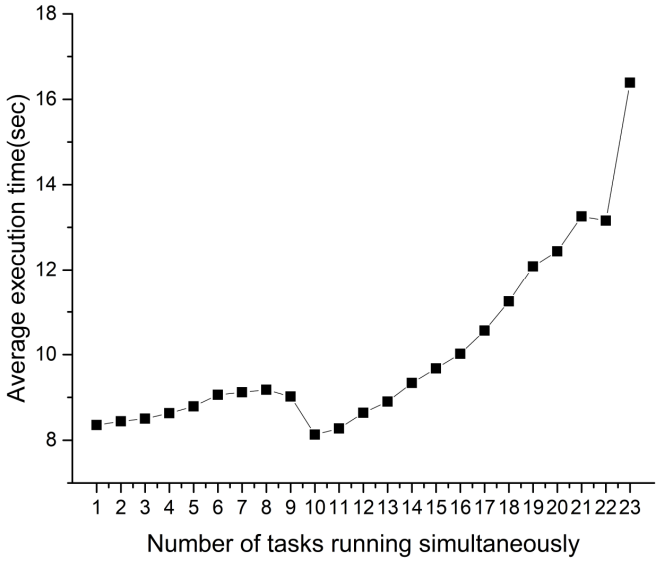

Fig. 1 Task failure due to insufficient GPU memory.

If the number of pods running inference tasks increases and available GPU memory becomes insufficient, multiple pods will fail to execute the task. When each inference task starts simultaneously, some containers cannot input all data into the GPU memory due to competition for GPU memory input tasks, the libraries to be executed on the GPU are not initialized, and the task fails to start.

Fig. 2 shows the individual performance when inference tasks are run simultaneously in 23 pods. As shown in the experimental results, when the number of concurrently running containers increases, tasks with large performance deviations occur due to competition for GPU resources. up to 23 pods executing inference tasks result in some degree of performance degradation while completing the inference tasks successfully. However, if the number of concurrently executing tasks increases, some tasks fail to execute. Table 1

shows the task execution failure when 24 and 25 pods are running inference tasks.

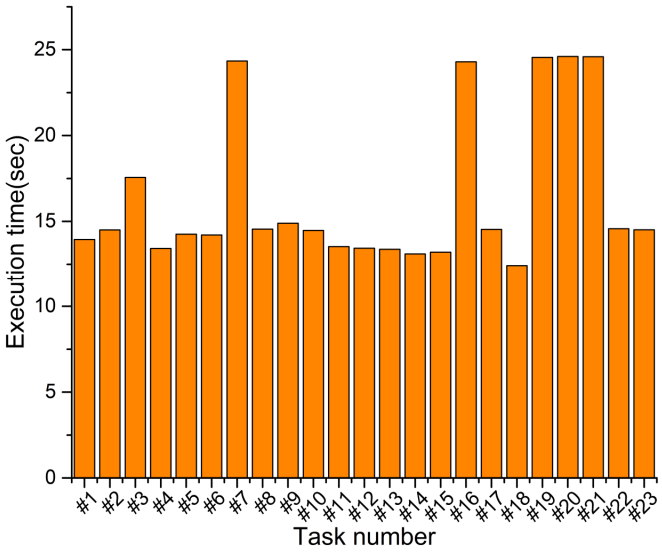

Fig. 2 Individual performance on inference tasks.

TABLE I
TASK EXECUTION FAILS DUE TO INSUFFICIENT GPU MEMORY

Number of tasks	Number of concurrently running tasks			
	24		25	
	run time	Execution completion	run time	Execution completion
1	11.728	x	13.8	o
2	15.662	o	13.788	o
3	24.017	o	11.044	x
4	15.65	o	13.413	o
5	15.126	o	11.322	x
6	18.208	o	11.187	x
7	24.174	o	13.022	o
8	13.838	o	11.118	x
9	14.322	o	11.209	x
10	15.724	o	11.149	x
11	13.803	o	12.993	o
12	15.354	o	11.169	x
13	15.305	o	11.507	x
14	15.756	o	11.366	x
15	25.498	x	13.452	o
16	18.147	o	11.089	x
17	25.678	x	11.287	x
18	13.992	o	11.068	x
19	25.569	x	13.365	o
20	15.81	o	13.182	o
21	25.592	x	11.027	x
22	15.74	o	10.75	x
23	15.447	o	13.924	o
24	25.606	x	13.847	o
25	-	-	11.385	x

As shown in the experimental results in Table 1, the number of pods executing inference tasks increased by only one and two, respectively. Still, a much larger number of pods terminated without completing execution. This is because, as competition for GPU memory intensifies, the number of tasks where only part of the data has been uploaded to GPU memory and the data upload required for the task cannot be completed due to lack of available capacity increases. As a result, some containers continue to compete for GPU memory while uploading only part of the data to GPU memory, and due to insufficient GPU memory, the GPU library used in TensorFlow cannot be initialized, and the task fails to start.

In this paper, we aim to prevent excessive use of GPU memory and the resulting task execution failure caused by the increase in the number of GPUs that can be recognized through extended resources without considering the actual GPU memory usage in the existing Kubernetes environment. To this end, we propose a pod placement technique based on each worker node's GPU memory usage and GPU usage frequency. The method proposed in this paper recognizes the availability of GPU resources based on information about actual GPU memory usage and GPU usage frequency and manages a list of candidates for servers where new pods can be deployed.

B. Implementation

The technique proposed in this paper utilizes extended resources to increase the number of virtual GPUs. However, the number of extended resources is independent of the decision of which server a pod will be placed on. In traditional Kubernetes, pods were placed on servers with the highest available GPUs based on the number of extended resources. However, as described earlier, the available number of extended resources is unrelated to the available ones. Therefore, the decision of where a new pod will be placed is based on the actual GPU memory usage and usage frequency rather than the number of available extended resources.

The technique proposed in this paper involves obtaining information from monitoring GPU resources on each server and the execution status of tasks. Based on this information, the least busy server is selected. To minimize transmission overhead, monitoring information is not transmitted periodically but rather tracked based on the execution status of GPU tasks using an event-based monitoring approach.

In our approach, the most critical information is the GPU memory and the execution status of tasks on each server. As described earlier, the technique proposed in this paper uses an event-based monitoring approach rather than the conventional method of periodically sending monitoring information to control nodes, commonly used in cloud environments. Initially, each worker node minimizes computing resource usage due to monitoring tasks by detecting only the start of GPU tasks. When the control node deploys a new pod, and GPU tasks commence, it verifies the execution of GPU tasks and monitors the GPU memory usage of those tasks. If the GPU memory usage of the pod remains constant after increasing GPU task execution, it is assumed that the input tasks for GPU memory have been completed. Subsequently, the control node sends the server's GPU memory usage information.

When a new GPU task is initiated, the worker node sends 3 pieces of information regarding that task: 1) when the GPU task starts, 2) when GPU memory input is completed, and 3) when the task is completed. Suppose the GPU memory usage for the specific GPU task remains constant and does not increase after the task has started. In that case, it is considered that the data input operation into the GPU memory has been completed. Therefore, monitoring information is transmitted when the GPU task starts and when GPU resource occupancy is completed. The control node receiving the GPU memory

usage information selects the most suitable worker node based on the GPU memory usage information and the frequency of GPU usage when deploying the next new pod. The operational mechanism of the monitoring technique proposed in this paper is shown in Fig. 4.

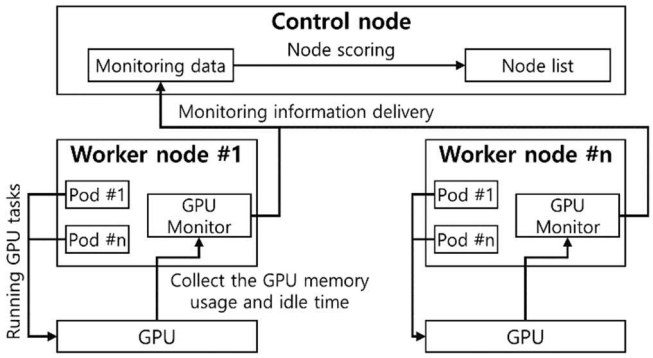

Fig. 3 Proposed GPU memory and task monitoring system

As shown in Fig. 3, sending monitoring information to the control node starts after initiating a new GPU task. Additionally, since our approach utilizes information regarding GPU memory usage and usage frequency, there is no need to know which task each pod executes and how much GPU memory it uses. It is sufficient only to monitor the current GPU memory usage and how frequently the GPU is being used. Therefore, no separate collection of information about the pods is conducted.

Algorithm 1: Pod placement

```
1    if (monitoring information has been transmitted) {
2       gpu_memory_usage = memory_usage;
3       gpu_idle_time =  idle_time;
4       for (0 < node_info[n])
5        node_score = (1 - gpu_memory_usage /
         total_gpu_memory) * (1 - gpu_idle_time / 100);

6       node_info[n] = node_score;
7    }
8    sorted_node[n] = sorted (node ID, node_score);
9    if (Pod creation requested);
10      for (0 < sorted_node[n])
11         select_node(sorted_node[0]);
12         if (node satisfies the CPU and memory required)
13            select_node_for_pod
14            break;
15         else
16            wait_for_pod_scheduling
```

The control node asynchronously receives monitoring information from each worker node and manages the node list in the order of nodes most suitable for deploying new pods based on this information. When deploying pods, it is necessary to check the available CPU and memory capacity, GPU memory usage, and use frequency to identify suitable nodes in two steps. The node list is sorted based on available GPU memory capacity in the first step. In the second step, a node with appropriate CPU and main memory requirements is selected and determined as the node for deploying the pod. Our proposed pod placement technique is described in Algorithm 1.

As shown in Algorithm 1, the approach proposed in this paper manages the node list based on current GPU memory usage information and GPU usage frequency. A high GPU usage frequency on a worker node implies that the pod frequently utilizes the GPU. We do not differentiate between pods that repeatedly execute GPU tasks over an extended period on worker nodes and those that execute GPU tasks only once and then terminate; instead, we assess the busy state of worker nodes based on GPU usage frequency. GPU usage frequency is determined by the number of times GPU tasks are executed and by GPU idle time. Since GPU tasks executed on worker nodes have varying execution times, a GPU task with a longer execution time will result in a lower GPU usage frequency measurement.

In the implementation of this paper, the idle time of the GPU is measured for 1 minute for each worker node, the weight of the previous information is calculated as half of the latest information, the average idle time information is managed, and monitoring information is delivered to the control node whenever a GPU task is executed. do. When the creation of a new pod is requested, the node list is checked from the beginning, and the worker node that matches the CPU and memory requirements is selected among the worker nodes with the lowest GPU memory usage and GPU usage frequency.

In this paper, we modify the existing Kubernetes pod placement technique and propose a method to solve the problem of not being able to limit GPU memory usage in Extended Resources used when sharing GPUs. Our approach considers the characteristics of GPU devices, where it is impossible to allocate only a portion of resources and uses a method to select worker nodes with the lowest probability of failing to execute tasks based on GPU memory usage and GPU usage frequency information. In particular, the proposed technique suggests that the node with the lowest GPU memory usage obtains the highest score, thus increasing the likelihood of having sufficient available GPU resources on the worker node even if the resource demands of newly launched GPU tasks are high.

Additionally, since monitoring information is transmitted only when the GPU task starts and when GPU memory allocation is completed, unnecessary transmission can be minimized. In addition, this method requires monitoring information to be transmitted more frequently when GPU tasks are commonly executed and terminated. Still, it has the advantage of checking the execution status of tasks in real-time that cannot be detected by sending monitoring information periodically. The efficiency of the technique proposed in this paper is verified through experiments in the next chapter.

III. Results and Discussion

A. Evaluation

In this chapter, an experiment is performed to verify the efficiency of the technique proposed in this paper. Our experiments use an inference task using the MNIST data set. Each pod that performs inference tasks performs inference tasks using a different amount of data. Because the number of

data used in inference tasks differs, GPU memory usage is also different. The inference task used in the experiment was implemented in TensorFlow, and the inference task prevents a single task from occupying the entire GPU through a TensorFlow function [6] that limits GPU memory usage to actual usage.

In the experiments in this chapter, we use the proposed technique in an environment where multiple inference tasks are executed simultaneously to check whether newly executed inference tasks are placed on appropriate worker nodes and completed without task failure. Also, measure the resource usage of proposed monitoring techniques and analyze their impact on the overall system. The experimental environment is shown in Table 2.

TABLE II
EXPERIMENT ENVIRONMENT

Type	Control node	Worker node × 2
CPU	AMD Ryzen 9 7900X	
Memory	128 GB	
GPU	-	RTX 4090
OS	Ubuntu 20.04	
Kubernetes version	1.24	

The first experiment verifies the efficiency of deploying pods using the technique proposed in this paper when pods using various capacities of GPU memory are executed simultaneously. In the experiment, pods were divided into two groups to vary the GPU memory usage, and each group used 20000 and 60000 data when performing inference tasks. Table 3 shows GPU memory usage and individual execution performance depending on the number of data to be used in the inference task.

TABLE III
INDIVIDUAL PERFORMANCE OF TASKS BASED ON THE NUMBER OF DATA

Number of data	execution time(sec)	GPU memory usage (MB)
40,000	5.09	821
60,000	8.632	1,205

When a pod is created, it is placed on one of three worker nodes. In the case of existing Kubernetes, pods are placed based only on the number of extended resources regardless of GPU memory usage, so if pods with high GPU memory usage are concentrated in a specific worker node, the probability of job execution failure increases.

To create a situation where GPU memory is insufficient in the experiment, the extended resource was set to 30, and 23 inference tasks using 60,000 pieces of data were placed on worker node 1, and 25 inference tasks using 40,000 pieces of data were placed on worker node 2. The inference task was arranged to run repeatedly and continuously occupy the GPU memory. In this situation, two new pods were created for the inference task using 60,000 pieces of data, and task failure due to the new pods was confirmed.

As shown in Table 4 in the experimental results, in the existing environment, the node on which a new pod is to be placed is selected based only on the available number of expansion resources, so when pods that use a lot of GPU memory are concentrated, pods that fail to execute tasks occur.

TABLE IV
TASK EXECUTION FAILS DUE TO NEW PODS WHEN GPU MEMORY IS LOW

Node Number	Number of pods placed	Number of pods that failed to run tasks
1	2	14
2	0	0

Table 5 shows the results when using the pod placement technique proposed in this paper in the same situation.

TABLE V
POD PLACEMENT USING THE PROPOSED TECHNIQUE

Node Number	Number of pods placed	Number of pods that failed to run tasks
1	0	0
2	2	0

As shown in the experimental results in Table 5, the technique proposed in this paper deploys pods using GPU based on GPU memory usage regardless of the available number of extended resources. As a result, new pods are not placed on worker nodes that have more available extended resources but use more GPU memory, but new pods are placed on worker nodes that have fewer available extended resources but lower GPU memory usage.

Due to this way of working, even without knowing the GPU memory usage of the newly created pod, the pod is placed on the worker node with the least busy GPU, so there is a low probability of job failure due to insufficient GPU memory. However, our approach does not fundamentally solve the GPU memory shortage problem. Because our approach is to place new pods on the least busy worker nodes, we are limited in our ability to respond to tasks that require full GPU memory. However, for GPU tasks with a relatively small computational amount, such as inference tasks or transfer learning used in the experiment, new pods are placed based on the GPU memory usage of the worker node, so the probability of task failure due to insufficient GPU memory can be reduced.

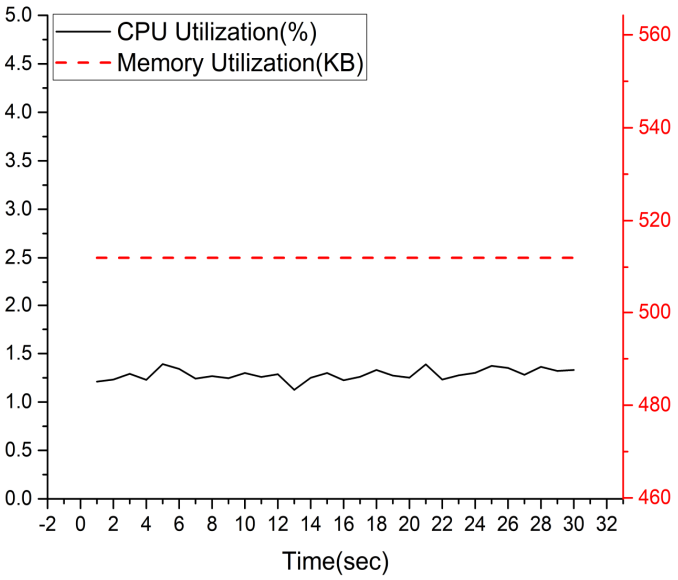

Fig. 5 The resource usage of the proposed monitoring technique

In this experiment, we measure the resource usage of the proposed GPU monitor to collect basic information when selecting a node to deploy a pod and analyze the impact of the proposed monitoring technique on the overall system. The experimental results are shown in Fig. 5. As shown in Fig. 5, the monitoring technique proposed in this paper uses very little CPU and main memory. For monitoring information collection work, the CPU uses only about 1 to 2%, and in the case of main memory, only a small, fixed capacity is used. The proposed monitoring technique runs inside a worker node and uses the level of computing resources that do not affect the performance of the pod that performs actual user tasks.

The pod placement technique proposed in this paper solves the GPU memory overuse problem that can occur in Kubernetes, which only checks the available number of extended resources without considering the available GPU memory capacity when allocating GPU resources to pods. As explained earlier, Kubernetes does not provide isolation for the GPU allocated to each pod when managing GPU resources, so it is not possible to limit the GPU resource usage of each pod. Because of this, if a specific pod uses a lot of GPU memory, available resources for actual GPU work may not be secured even if the number of expansion resources is sufficient. This leads to execution failure of GPU tasks due to insufficient GPU memory. However, in this paper, new pods are placed based on the actual amount of available GPU memory and GPU usage frequency regardless of the number of expansion resources, thereby alleviating the problem of insufficient GPU memory and preventing execution failure of GPU tasks.

The technique proposed in this paper targets tasks that require relatively small GPU resource usage, such as inference tasks or transfer learning. In the case of large-scale tasks that fully use the GPU, such as learning tasks, the GPU must be allocated exclusively, but the proposed technique evenly distributes pods to multiple worker nodes based on GPU memory usage and GPU usage frequency. For jobs that use all of them, there is a high probability that the batch will fail. This limitation will be addressed through future research to develop pod relocation techniques to prevent partial fragmentation of GPU memory. In future research, we will extend the technique proposed in this paper to secure available GPUs through relocation technology, enabling scalability to handle large-scale computations.

B. Related Works

Various research has been conducted to manage GPU resources in the Kubernetes environment. Existing research focuses on managing device-level GPU resources in a multi-GPU environment where two or more GPUs exist on a single node in a cluster environment. Existing research on managing GPUs on a per-device basis deploys tasks in a cluster environment and manages GPU devices for large-scale tasks using multiple GPUs. The main goal of existing research is to propose techniques for scheduling between each task and preventing fragmentation of available GPU resources scattered throughout the cluster [7], [8], [9], [10].

Additionally, to alleviate performance degradation due to GPU resource competition, a technique was also proposed to improve throughput in an environment where multiple GPU tasks share the GPU by pausing and then restarting the GPU task. These techniques typically support pausing and restarting tasks and reallocating GPUs through framework-level modifications for GPU tasks [11], [12], [13]. Another method of managing GPU resources in a container-based environment is a technique that dynamically allocates resources by analyzing and predicting the completion time and pattern of tasks [14], [15], [16], [17], [18], [19], [20], [21]. These techniques collect runtime information of tasks and achieve optimization of task placement based on this. To improve the performance of tasks running on GPUs, a technique was also proposed to dynamically select the GPU access method of a virtual instance according to resource requirements and container parameters [22]. This method optimizes GPU resources by selecting various GPU resource access technologies, such as passthrough and API forwarding, according to requirements.

A technique was also proposed to optimize data communication for tasks by caching models used when multiple containers perform AI tasks simultaneously in GPU memory [23]. This method proposes a locality-aware scheduler using global management techniques of GPU memory. In addition, in order to reduce fragmented GPUs scattered in the cluster, a technique for scheduling tasks based on GPU fragmentation measurement technique was proposed [24]. To efficiently allocate GPUs when performing distributed learning, a technique was also proposed that uses AI to select nodes that satisfy multiple conditions [25]. This technique uses an AI-based scheduling technique to determine which nodes to place tasks in the Kubernetes environment.

There is existing research on priority-based management of GPU resources. The proposed research prioritizes GPU jobs based on user-specific job deadlines [26], [27] or QoS [28] and provides GPUs based on priority. In addition, methods proposed for providing GPUs that exist on other servers in the cluster based on GPU usage information [29] and techniques for finding idle GPUs and placing [30].

IV. CONCLUSION

In this paper, we propose a pod placement technique that can select appropriate worker nodes to prevent task failure due to insufficient GPU memory when multiple pods share GPUs in a Kubernetes environment. The extended resource used to support GPU sharing among multiple pods in existing Kubernetes makes it appear that there are multiple GPUs, but the GPU resources are not divided. Because of this, the GPU resource usage of each pod cannot be limited. In a situation where a specific pod uses most of the GPU memory if the number of available extended resources of the worker node is large, a new pod is deployed regardless of the actual GPU memory usage. In this paper, to solve this problem, worker nodes where new pods will be placed are selected based on actual GPU memory usage and GPU resource usage frequency rather than the number of extended resources. This allows newly created pods to be placed on worker nodes where the GPU is not busy, alleviating GPU memory contention and preventing tasks from executing due to insufficient GPU memory. As confirmed through experiments, the proposed technique effectively solves the GPU memory shortage problem caused by the existing method of placing pods by considering only the number of

available extended resources. Even if it does not know the GPU memory usage of a newly executed pod, when selecting a worker node on which a new pod will be placed, the least busy worker node is selected, thereby preventing task failure with a high probability.

Our proposed method effectively prevents task execution failure due to GPU memory contention in situations where different GPU memory usage of simultaneously running pods. However, the technique proposed in this paper determines the worker node where the GPU is least busy as the placement node for the new pod, so it operates using a bean-packing method based on the state of GPU resources. As a result, there is an extremely high probability that pods using GPUs are evenly distributed to each worker node. Because of this, when a pod is created that runs a task that requires all GPU resources, such as an AI learning task, it is hard to prevent task failure even if the most appropriate worker node is selected in the current situation. To solve this problem, this paper plans to study a pod relocation technique based on the status of GPU memory I/O operations of other pods sharing the GPU as a future study.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea funded (NRF) by the Ministry of Education (2022R1I1A1A01063551 and NRF-2021R1F1A1063121).

REFERENCES

[1] Docker, Docker engine [Online]. Available: https://docs.docker.com/engine/

[2] CUDA C Programming Guide, NVIDIA Corporation, CA, USA, 2024.

[3] Linux Foundation, Kubernetes, [Online]. Available: https://kubernetes.io/docs/setup/

[4] CUDA API Reference Manual, CA, USA, pp. 59, 2012.

[5] M. Abadi *et al*., TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, arXiv preprint arXiv:1603.04467, 2016.

[6] TensorFlow, Use GPU, [Online]. Available: https://www.tensorflow.org/guide/gpu.

[7] T.-A. Yeh, H.-H. Chen, and J. Chou, "KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud," *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, vol. 2014, pp. 173–184, Jun. 2020, doi: 10.1145/3369583.3392679.

[8] I. Harichane, S. A. Makhlouf, and G. Belalem, "KubeSC-RTP: Smart scheduler for Kubernetes platform on CPU-GPU heterogeneous systems," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 21, Jun. 2022, doi: 10.1002/cpe.7108.

[9] G. El Haj Ahmed, F. Gil-Castiñeira, and E. Costa-Montenegro, "KubCG: A dynamic Kubernetes scheduler for heterogeneous clusters," *Software: Practice and Experience*, vol. 51, no. 2, pp. 213–234, Sep. 2020, doi: 10.1002/spe.2898.

[10] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters," *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2019, doi: 10.1109/cluster.2019.8891040.

[11] S. Wang et al., "An Efficient and Non-Intrusive GPU Scheduling Framework for Deep Learning Training Systems," *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, Nov. 2020, doi:10.1109/sc41405.2020.00094.

[12] N. Zhou et al., "Container orchestration on HPC systems through Kubernetes," *Journal of Cloud Computing*, vol. 10, no. 1, Feb. 2021, doi: 10.1186/s13677-021-00231-z.

[13] J. Shi, D. Chen, J. Liang, L. Li, Y. Lin, and J. Li, "New YARN sharing GPU based on graphics memory granularity scheduling," *Parallel Computing*, vol. 117, p. 103038, Sep. 2023, doi:10.1016/j.parco.2023.103038.

[14] T.-T. Hsieh and C.-R. Lee, "Voda: A GPU Scheduling Platform for Elastic Deep Learning in Kubernetes Clusters," *2023 IEEE International Conference on Cloud Engineering (IC2E)*, vol. 27, pp. 131–140, Sep. 2023, doi: 10.1109/ic2e59103.2023.00023.

[15] H. Albahar, S. Dongare, Y. Du, N. Zhao, A. K. Paul, and A. R. Butt, "SchedTune: A Heterogeneity-Aware GPU Scheduler for Deep Learning," *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2022, doi:10.1109/ccgrid54584.2022.00079.

[16] G. Yeung, D. Borowiec, R. Yang, A. Friday, R. Harper, and P. Garraghan, "Horus: Interference-Aware and Prediction-Based Scheduling in Deep Learning Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 88–100, Jan. 2022, doi: 10.1109/tpds.2021.3079202.

[17] J. Gu, Y. Zhu, P. Wang, M. Chadha, and M. Gerndt, "FaST-GShare: Enabling Efficient Spatio-Temporal GPU Sharing in Serverless Computing for Deep Learning Inference," *Proceedings of the 52nd International Conference on Parallel Processing*, pp. 635–644, Aug. 2023, doi: 10.1145/3605573.3605638.

[18] Z. Liu, C. Chen, J. Li, Y. Cheng, Y. Kou, and D. Zhang, "KubFBS: A fine-grained and balance-aware scheduling system for deep learning tasks based on kubernetes," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 11, Jan. 2022, doi: 10.1002/cpe.6836.

[19] I. Harichane, S. A. Makhlouf, and G. Belalem, "A Proposal of Kubernetes Scheduler Using Machine-Learning on CPU/GPU Cluster," *Intelligent Algorithms in Software Engineering*, pp. 567–580, 2020, doi: 10.1007/978-3-030-51965-0_50.

[20] L. Liu, J. Yu, and Z. Ding, "Adaptive and Efficient GPU Time Sharing for Hyperparameter Tuning in Cloud," *Proceedings of the 51st International Conference on Parallel Processing*, pp. 1–11, Aug. 2022, doi: 10.1145/3545008.3545027.

[21] J. Lou, Y. Sun, J. Zhang, H. Cao, Y. Zhang, and N. Sun, "ArkGPU: enabling applications' high-goodput co-location execution on multitasking GPUs," *CCF Transactions on High Performance Computing*, vol. 5, no. 3, pp. 304–321, May 2023, doi:10.1007/s42514-023-00154-y.

[22] W. Shen, Z. Liu, Y. Tan, Z. Luo, and Z. Lei, "KubeGPU: efficient sharing and isolation mechanisms for GPU resource management in container cloud," *The Journal of Supercomputing*, vol. 79, no. 1, pp. 591–625, Jul. 2022, doi: 10.1007/s11227-022-04682-2.

[23] M. Zhao, K. Jha, and S. Hong, "GPU-enabled Function-as-a-Service for Machine Learning Inference," *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, vol. 11, pp. 918–928, May 2023, doi: 10.1109/ipdps54959.2023.00096.

[24] Q. Weng, L. Yang, Y. Yu, W. Wang, X. Tang, G. Yang, et al., "Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent", *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 995-1008, 2023, [online] Available: https://www.usenix.org/conference/atc23/presentation/weng.

[25] D. Jorge-Martinez et al., "Artificial intelligence-based Kubernetes container for scheduling nodes of energy composition," *International Journal of System Assurance Engineering and Management*, Jul. 2021, doi: 10.1007/s13198-021-01195-8.

[26] M. Saravanan and R. Vignesh, "DSTS: A hybrid optimal and deep reinforcement learning for dynamic scalable task scheduling on container cloud environment," Mar. 2022, doi: 10.21203/rs.3.rs-1431790/v1.

[27] Y. Mao et al., "Differentiate Quality of Experience Scheduling for Deep Learning Inferences With Docker Containers in the Cloud," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1667–1677, Apr. 2023, doi: 10.1109/tcc.2022.3154117.

[28] A. Zou, J. Li, C. D. Gill, and X. Zhang, "RTGPU: Real-Time GPU Scheduling of Hard Deadline Parallel Tasks With Fine-Grain Utilization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 5, pp. 1450–1465, May 2023, doi:10.1109/tpds.2023.3235439.

[29] Z. Chen, X. Zhao, C. Zhi, and J. Yin, "DeepBoot: Dynamic Scheduling System for Training and Inference Deep Learning Tasks in GPU Cluster," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 9, pp. 2553–2567, Sep. 2023, doi: 10.1109/tpds.2023.3293835.

[30] J. Kennedy, V. Sharma, B. Varghese, and C. Reaño, "Multi-Tier GPU Virtualization for Deep Learning in Cloud-Edge Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 7, pp. 2107–2123, Jul. 2023, doi: 10.1109/tpds.2023.3274957.