API Interception-Based GPU Virtualization for Containerized HPC Workloads in Cloud Environments

Ming Lu[1,*], Heng Wu[2], Rongzhou Luo[2]
[1]Henan Open University School of Information Engineering and Artificial Intelligence, Zhengzhou 450008, China
[2]Institute of Software Research, Chinese Academy of Sciences, Beijing 100190, China
E-mail: hndd2025@163.com, wuheng@iscas.ac.cn, luorongzhou@iscas.ac.cn
[*]Corresponding author

Keywords: computing power, GPU virtualization, containers, artificial intelligence

Received: September 25, 2025

In recent years, with the development of cloud-native technologies such as containers and Kubernetes, high-performance computing (HPC) tasks using GPUs have gradually migrated to container cloud environments, introducing new challenges in fine-grained GPU resource management. Currently, the native Kubernetes framework lacks support for allocating fractional GPU resources to containers, permitting only exclusive access to entire physical GPUs, which results in low cluster-wide GPU utilization. To enable efficient GPU sharing for HPC tasks in containerized environments, GPU virtualization — allocating precise amounts of GPU compute and memory resources to different containers with isolation guarantees — becomes essential. However, current GPU virtualization technologies remain in their infancy and exhibit the following limitations: (1) NVIDIA GPUs enforce strict closed-source policies at the driver and lower layers, forcing existing solutions to rely on reverse engineering approaches for virtualization; (2) current implementations fail to address GPU idle cycles during HPC task execution, leading to computational resource wastage. To address these issues, this paper proposes a GPU virtualization system for container cloud environments. The key contributions include: (a) profiling the workflow and invocation mechanisms of CUDA-based HPC tasks (e.g., deep learning training) to characterize their GPU memory and compute usage patterns; (b) developing formal models for HPC resource utilization processes; and (c) implementing a resource isolation and quota mechanism via API interception and forwarding. Experimental results demonstrate that our system achieves superior virtualization efficiency with lower overhead. The proposed adaptive elastic GPU allocation method yields 37% higher average GPU utilization and 26% greater cluster throughput under heavy loads compared to static allocation in KubeShare.

Povzetek: Prispevek predstavi sistem za virtualizacijo GPU-jev v kontejnerskih oblačnih okoljih, ki omogoča natančnejšo in prilagodljivo delitev GPU-virov ter s tem izboljša izkoriščenost in zmogljivost pri HPC-nalogah.

1 Introduction

Currently, the use of GPUs to provide arithmetic power in machine learning training has become very common, and the research on GPU-based AI system is also in full swing. Resource utilization is one of the hot spots in current researches. This paper focuses on the API-based intercept-and-forward resource isolation and quota algorithm, which can dynamically select GPU virtualization to deploy containers to meet container runtime performance requirements, improve GPU utilization and overall system throughput. The first part analyzes the hardware and software hierarchy of Nvidia GPU, explains the GPU virtualization approach based on API interception and forwarding adopted in this paper, and emphasizes the two main indicators to consider, namely, efficiency and fairness. The second part analyzes the flow of HPC tasks calling CUDA APIs, finds out how HPC tasks use device memory (HBM/GDDR) resources and compute resources, models the use of device memory (HBM/GDDR) resources and compute resources by HPC tasks, and implements the resource isolation and quota algorithms for device memory (HBM/GDDR) and compute resources, respectively. In the third part, relevant experiments are conducted, and the experimental results are compared and analyzed to illustrate the effectiveness of the resource isolation and quota algorithms implemented in this paper.

"GPU virtualization" is a broad term encompassing a range of technologies designed to enable multiple workloads (such as containers, virtual machines) to share physical GPU resources. However, there are significant differences in their implementation methods and levels of abstraction, which can be primarily categorized as follows:

Hardware Partitioning: Exemplified by NVIDIA's Multi-Instance GPU (MIG). This technology physically

partitions the computing units and memory of a high-end GPU (such as the A100) into multiple independent, physically isolated instances. Each instance possesses its own parallel computing engines and memory controllers, providing strong fault and performance isolation. However, MIG requires specific hardware support, partitioning is static, and the granularity is relatively coarse.

Device Passthrough & Hardware Virtualization: This includes technologies like SR-IOV and NVIDIA's GPU solution. They create multiple Virtual Functions (VFs) within the GPU hardware and directly assign each VF to a virtual machine. This offers a GPU experience close to native performance with good isolation. However, it typically relies on enterprise-grade GPU hardware and specific software licensing, and lacks elasticity after resource allocation.

Time-Slicing: This is the baseline sharing scheme provided by the NVIDIA Container Runtime in conjunction with groups. It shares the GPU's computing engine among multiple processes in a round-robin fashion, achieving concurrency at a macroscopic level. However, it is a coarse-grained method for sharing computing resources and completely lacks the ability to limit and isolate memory usage per container, posing a risk of memory overflow.

Multi-Process Service (MPS): MPS allows the computational workloads of multiple CUDA processes to be interleaved for execution on the GPU, aiming to improve the utilization of compute units. However, its design goal is not strong isolation; conversely, it mixes the workloads of different processes, and an error in one process can cause the entire GPU context to crash, offering the poorest isolation.

API-level Virtualization: This is the approach adopted in this paper. Its core lies in the technique of API Interposition and Forwarding. It intervenes at the CUDA Driver API layer, transparently intercepting GPU resource requests (such as memory allocations, kernel launches) issued by applications within containers, and manages and isolates them based on predefined quotas. This method does not require specific hardware support and can simultaneously achieve fine-grained, software-defined isolation and quotas for both memory and compute resources.

Remote GPU: Examples include CUDA, where the GPU is located on a remote server and shared by multiple clients over a network. This introduces significant network communication overhead, and its challenges and objectives are fundamentally different from those of local GPU virtualization.

Positioning of this work:

The GPU virtualization system implemented in this paper primarily follows the API-level virtualization technical route described above. By intercepting CUDA Driver API calls in user space, we have implemented a quota-based memory resource isolation algorithm and a monitoring-based compute resource regulation algorithm. The advantages of this scheme are its hardware agnosticism (no need for specialized MIG or GPU hardware), fine-grained resource control (simultaneously managing memory and compute), and deployment flexibility.

2 Problem analysis

The purpose of virtualizing GPUs in cloud environments is to allow containers to share physical GPUs among themselves, and to allocate each container its corresponding quota according to the computational and device memory (HBM/GDDR) resources requested by the container, so as to ensure that containers remain isolated from each other and do not interfere with each other while running [1,2]. Therefore, sharing GPUs among containers is essentially concurrent execution of different processes on GPUs.

In order to establish a virtualization mechanism for GPUs in cloud environments, an in-depth understanding of the hardware and software call stack of GPUs for HPC tasks is required. In this paper, we study the virtualization of GPUs based on the CUDA computing platform of Nvidia, which is currently the mainstream company. It is found that the hardware and software call stacks of Nvidia GPUs for HPC tasks are clear, and the top-to-bottom hardware and software call stacks are shown in Figure 1.

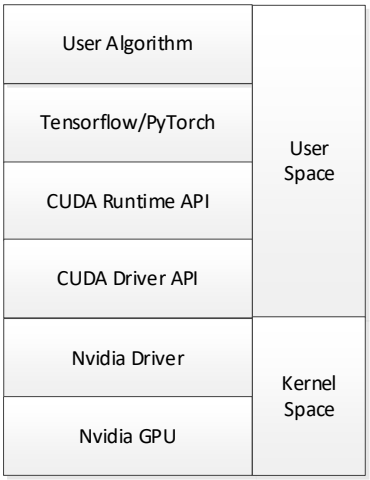

Figure 1: Hardware and software call stack for high performance computing tasks on Nvidia GPUs.

The topmost layer is the user algorithm program which is in the user state. User algorithmic programs are generally developed using deep learning programming frameworks, such as PyTorch, TensorFlow, etc., which are developed using Python language, by building operators or neural networks, and finally building a deep learning model and using deep learning programming frameworks for training or inference logic [3,4].

The middle layer is the CUDA API layer in the user state. Deep learning programming frameworks such as PyTorch, TensorFlow, etc, ultimately access the GPU to perform logic such as data copying and computation by calling the CUDA runtime APIs and CUDA driver level APIs through calls to the lower programming interfaces such as C, C++, and so on.

At the bottom is the Nvidia driver and physical hardware circuitry in the kernel state and physical layer. The CUDA driver-level APIs ultimately call the Nvidia

driver, which in turn uses Nvidia's physical GPUs to execute the final GPU-accelerated algorithms [5-7].

Since GPU manufacturers such as Nvidia have implemented a strict closed-source policy at the driver and physical layers, a feasible GPU virtualization solution is to virtualize at the CUDA API layer, specifically by using an API-based intercept-and-forward approach. Therefore, we need to analyze and summarize the related CUDA APIs, which are divided into runtime APIs and driver APIs. Since the runtime APIs are just a higher-level encapsulation of the driver APIs, and the driver APIs are more flexible, this paper intercepts and forwards the driver APIs. Due to the closed-source strategy, it is impossible to control the execution of different processes on the GPU, so the core goal of virtualization in this paper is to limit and isolate the device memory (HBM/GDDR) and computing resources applied by containers.

Nvidia provides a large number of CUDA driver APIs, which are mainly divided into three categories: APIs related to device memory (HBM/GDDR) resources, APIs related to compute resources, and APIs related to device control information, and some typical CUDA driver APIs are shown in Table 1.

Table 1: Some CUDA driver-level APIs.

Category	**API Name**	**Core Parameter / Return Value Unit**	**Description**
Memory Resources	cuArray3DCreate	Bytes	Creates a 3D array memory space
	cuMemAlloc	Bytes	Allocates device memory
	cuMemFree		Deallocates device memory
	cuArrayDestroy		Destroys an array memory space
Compute Resources	cuLaunchKernel	count, Bytes	Launches a CUDA kernel function
	cuLaunch	count,	Launches a kernel (simplified interface)
	cuLaunchGrid	count	Launches a kernel with grid-level configuration
	cuLaunchCooperativeKernel		Launches a cooperative kernel (multi-block synchronization)
Device Control	cuInit	Count/	Initializes the CUDA driver API
	cuDeviceGetAttribute	Bytes, kHz	Queries specific device attributes
	cuDeviceTotalMem	Bytes	Retrieves the total available device memory
	cuMemGetInfo	Bytes	Queries free and total memory statistics

Virtualizing GPUs in a cloud environment to enable container sharing of GPUs requires two evaluation metrics, efficiency and fairness, to be maximized.

Efficiency refers to the difference between the Job Completion Time (JCT) of a container running through virtualized logic, limiting the computational and memory resources of a task, and the JCT of a container running directly without virtualized logic, with a larger difference indicating lower efficiency.

Fairness refers to whether the container actually gets the specified number of resources after limiting the computational and device memory (HBM/GDDR) resources of the task, reflecting the performance of the task. Further fairness can be categorized into strong and weak fairness, where strong fairness refers to the fact that task performance can be guaranteed to be fair within a short period of time and the performance is continuously guaranteed to be stable, while weak fairness refers to the fact that the task performance may constantly fluctuate, but the average value of the performance can be guaranteed to be a more stable value over a longer period of time [8].

Therefore, this paper adopts the GPU virtualization method based on API interception and forwarding, by intercepting the CUDA driver-level APIs and adopting different resource quota algorithms for the device memory (HBM/GDDR) resources and computing resources, to achieve the effect of sharing GPUs among containers and to improve the efficiency and fairness as much as possible [9].

In the modeling of the process of HPC tasks invoking device memory (HBM/GDDR) and computing resources, the process of accelerating HPC tasks using the CUDA computing platform is roughly divided into the following stages:

(1) Initialize the CUDA operating environment, obtain device information and other related logic, at which

time APIs such as cuInit and cuDeviceGetAttribute will be called.

(2) Claim and allocate memory space on the device side, which will call APIs such as cuMemAlloc.

(3) Copy the data from the host side to the device side, which will call APIs such as cuMemcpyHtoD.

(4) Call a custom CUDA kernel function (Kernel) to start a thread on the device side to execute the computation logic, which will call, for example, the API of cuLaunchKernel.

(5) Copy the data on the device side back to the host side, which calls the API of cuMemcpyDtoH for example.

(6) Release the unneeded memory space on the device side, which will call the API of cuMemFree for example.

Based on the above process, the flowchart of the HPC task accelerated using the CUDA computing platform is shown in Figure 2

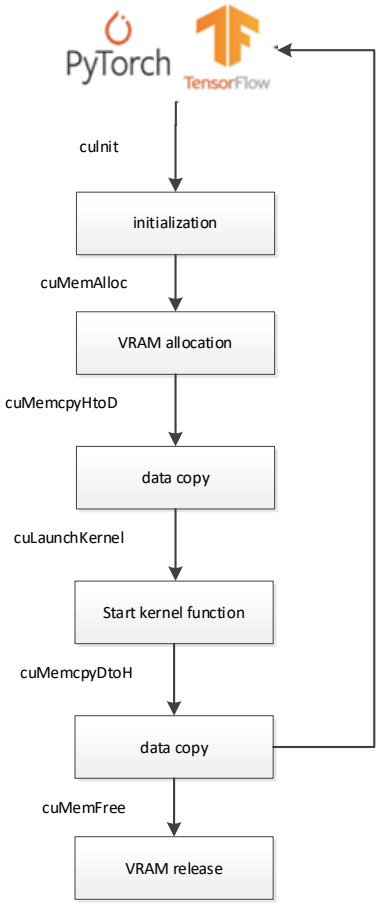

Figure 2: Flowchart of high-performance computing tasks using CUDA computing platforms.

Analyzing the process of calling CUDA acceleration libraries for HPC tasks, we find that the phases involving GPU resources are device memory (HBM/GDDR) request and release, and starting CUDA kernel functions.

(1) Modeling of Device memory (HBM/GDDR) Resources

The invocation of device memory (HBM/GDDR) resources by HPC tasks is relatively simple. Currently, the mainstream GPUs do not enable technical solutions such as device memory (HBM/GDDR) exchange due to the limitations of IO transfer performance, so we can consider that the total amount of device memory (HBM/GDDR) resources of a GPU is fixed, and its size is related to the model of the GPU, for example, the available device memory (HBM/GDDR) of Nvidia RTX 3090Ti is 24GB. When containers share a GPU, all containers will obtain and use device memory (HBM/GDDR) resources from the GPU, and the total amount of device memory (HBM/GDDR) used by all containers cannot exceed the total amount of available device memory (HBM/GDDR) of the GPU. The containers share the device memory (HBM/GDDR) resources of the GPU as shown in Figure 3.

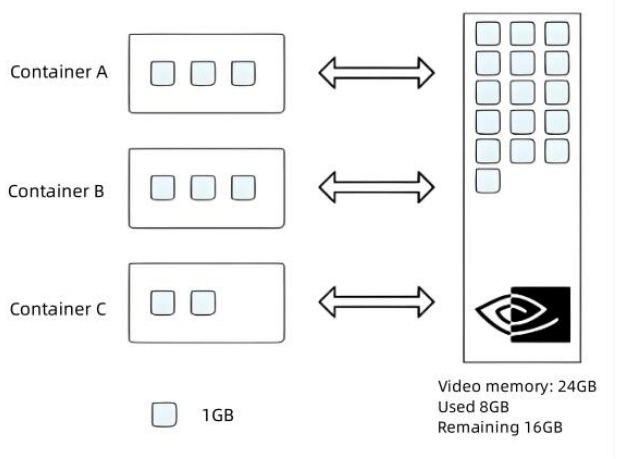

Figure 3: Containers share device memory (HBM/GDDR) resources using GPUs.

From the characteristics of GPU device memory (HBM/GDDR) resources, it is known that the allocation and reclamation requests of device memory (HBM/GDDR) resources accompany the entire life cycle of HPC tasks. Assuming that task i is a container in the cluster running a HPC task, and the container requests and releases the device memory (HBM/GDDR) resources several times throughout the lifecycle, the device memory (HBM/GDDR) resource usage of task i after startup and running until time $TMemoryi$, T can be expressed in Equation 1. Where $Malloci$, t denotes the amount of device memory (HBM/GDDR) resources requested and allocated by the container at time t, $Mfreei,t$ denotes the amount of explicit memory resources released by the container at time t.$Mfreei,t$ denotes the amount of device memory (HBM/GDDR) resources released by the container at time t.

$$Memory_{i,T} = \sum_{t=0}^{T}(Malloc_{i,t} - Mfree_{i,t}) \quad (1)$$

In order for the containers to run properly, it is necessary to strictly ensure that the sum of the graphics resources used by all containers on a single GPU at a certain moment cannot be higher than the actual amount of graphics resources available on that GPU, as shown in Equation 2. Where n denotes the number of containers on the GPU and $GMemory$ denotes the actual amount of device memory (HBM/GDDR) resources available on the GPU.

$$0 \le \sum_{i=0}^{n-1} Memory_{i,T} < GMemory \quad (2)$$

When performing the algorithm of device memory (HBM/GDDR) resource isolation and quota for GPUs, assuming that the size of the device memory (HBM/GDDR) resource quota for container i is $LimitMemoryi$, then the conditions to be satisfied are shown in Equation 3.

$$0 \le Memory_{i,T} < \text{Limit } Memory_i \quad (3)$$

Unlike device memory (HBM/GDDR) resources, GPU compution resources used for deep learning tasks are not a simple metric and require abstraction and modeling.

(2) Modeling of computing resources

It is found that HPC tasks are accelerated using the computational resources of the GPU, and the main process is to start the CUDA kernel functions by calling computational APIs such as cuLaunchKernel, and start a large number of computational threads on the parallel computational units of the GPU to execute the computational logic.

According to the architecture of Nvidia GPUs, the parallel computing units inside Nvidia GPUs of different architectures are divided into different levels. The current mainstream Nvidia GPUs are composed of a series of Streaming Multiprocessors (SMs), each of which can run a large number of parallel computing threads. For example, the Nvidia Tesla T4 GPU has 40 SMs with up to 1024 threads running on each SM.

In terms of the programming model of the CUDA computing platform, when a computing API such as cuLaunchKernel is called, a CUDA kernel is launched on the GPU to perform computation. A Kernel corresponds to a Grid of threads, a Grid contains multiple Blocks of threads, and a Block contains multiple Threads. In short, one Block of the programming model will be computed on one SM of the GPU hardware, as shown in Figure 4. Therefore, the more SMs there are, the more threads the GPU can parallelize and the more efficient the computation will be.

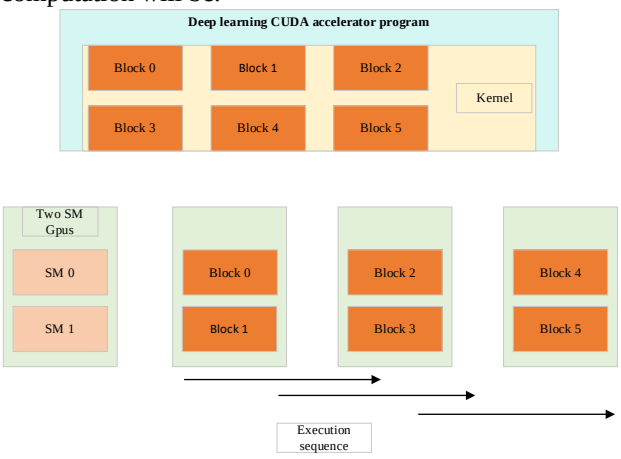

Figure 4: Schematic diagram of Block execution on the SM of the GPU.

High-performance computing tasks using GPUs such as deep learning are generally use computational graph models developed by programming frameworks. In the computational graph model, each node represents an operator, which contains some data operations, such as addition, convolution, matrix multiplication, etc., which are actually realized by the corresponding CUDA kernel functions written by the framework. Therefore, the process of accelerating HPC tasks using GPUs is the process of launching a large number of CUDA kernel functions at the bottom.

According to the concurrent multitasking design of Nvidia GPUs [10], multitasks are executed concurrently between tasks on a macro level when sharing GPUs, but serially between tasks on a micro level. Specifically, the Nvidia driver maintains a process context for each task and performs frequent context switches between tasks, which execute their own CUDA core functions in their respective time slices and only one CUDA core function can be executed on the GPU at the same time [11].

In summary, a HPC task using the computational resources of the GPU can be abstracted as running a CUDA core function on the GPU, and the longer the computation time of the CUDA core function, the longer it occupies the GPU, and the more computational resources are used by the GPU.

Because executing CUDA core functions is a short process, Nvidia uses GPU Utilization to indicate current GPU compute resource usage. According to Nvidia's official definition, GPU Utilization is the percentage of time that one or more CUDA core functions have been running on the GPU for the past sampling period (1/6th of a second to 1 second). In practice, the GPU utilization is typically viewed using the command line nvidia-smi, or you can programmatically view the GPU utilization of different processes using the corresponding APIs. Therefore, the compute resources used by a container can be represented by the GPU utilization of the processes within that container. The higher the GPU utilization of a container, the more time the container has spent on GPU computing units in the current time period.

Assuming that task $i$is a container in the cluster running a HPC task, task i is started after

runs until time T when the computational resource usage $Computationi,T$ can be expressed in Equation 4. where Δt denotes the sampling period and $Kerneli,t$ denotes the execution time of the CUDA kernel function initiated by task i at time t.

$$Computation_{i,T} = \frac{\sum_{t=T-\Delta t}^{T} Kernel_{i,t}}{\Delta t} \times 100\% \quad (4)$$

In this paper, we define that the computational resource usage of a container at a given moment in time is equal to the GPU utilization of that container. Since the GPU utilization rate is based on the sampling period in the past time period, there is some error, so it may happen that the sum of the computational resource usage of all the HPC tasks running on a certain GPU is greater than 100% at a certain moment [12].

When performing the computational resource isolation and quota algorithm for GPUs, strong fairness cannot be achieved due to the errors in the official GPU utilization sampling interface provided by Nvidia, only weak fairness can be guaranteed, i.e., the difference

between the average value and the limit value of the GPU utilization of the container i over the past time period is lower than a certain threshold, which can be expressed in Equation 5. where ΔT denotes the sampling time period for assessing the fairness of the GPU's computational resource segregation and quota algorithm, $LimitComputation_i$ denotes the size of the computational resource limit for container i, and δ denotes the maximum value allowed by the difference between the average utilization of the container's actual GPUs and the limited utilization.

$$\left| \frac{\sum_{t=T-\Delta T}^{T} Computation_{i,t}}{\Delta T} - LimitComputation_i \right| \leq \delta \quad (5)$$

3 Resource segregation and quota algorithm based on API intercept forwarding

After modeling the HPC tasks in terms of explicit memory resources and computational resources, it is necessary to implement resource limitation and isolation algorithms on the basis of the model of resource utilization [13]. The meaning of resource isolation and limitation is to allocate a specified amount of explicit memory resources and computational resources to the container, and to ensure that the total amount of resources actually used in the container operation is approximately equal to the specified amountions of resources within a certain margin of error.

In this paper, we virtualize the CUDA driver API layer, and adopt a GPU resource isolation and quota algorithm based on API interception and forwarding. Because the container uses different models for memory resources and computation resources, this paper adopts a quota-based regulation algorithm for memory resources and a monitoring-based regulation algorithm for compute resources, as shown in Figure 5.

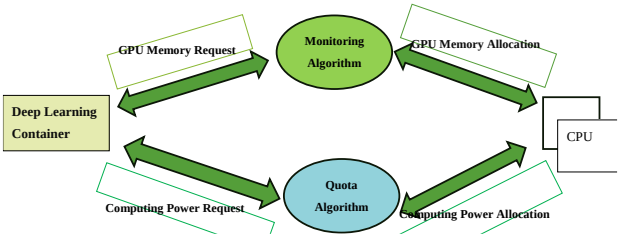

Figure 5: Different segregation and quota algorithms for device memory (HBM/GDDR) and compute resources.

(1) Quota-based algorithm for GPU device memory (HBM/GDDR) resource isolation and limitation

Segregation and quota of GPU memory resources is relatively simple. As mentioned in the previous section, containers will continuously request and release GPU memoryresources throughout their lifecycle. Therefore, this paper maintains a memory quota for each container, which records the current size of the memory used by the container, and the memory quota is set to 0 when the container is created.

When the process in the container calls the CUDA driver APIs related to memory requestions such as cuArray3DCreate, cuMemAlloc, etc., the CUDA library intercepts the call, gets the size of the requested memory, and adds the requested memory value to the container's current memory quota and determines whether it exceeds its specified quota size. If the specified amount is exceeded, the CUDA library reports an exception for insufficient memory, and the CUDA library does not continue to call the real CUDA library; if the specified amount is not exceeded, the CUDA library continues to call the real CUDA library, executes the API for allocating memory, and returns the real result of its call.

When a process in a container calls a CUDA driver API related to memory freeing, such as cuArrayDestroy, cuMemFree, etc., the CUDA wrapper intercepts the call, gets the size of the freed memory via a pointer, and subtracts the value of the freed memory from the container's current memory quota. API returns the true result of its call.

In summary, the quota-based GPU memory resource isolation and quota algorithm is shown in Algorithm 1, which contains two processes: memory resource request and memory resource release. Lines 1 to 12 of the algorithm are the memory request part, which inputs the memory request size and quota size as well as the information of the current container, and when the memory resource quota is insufficient, it returns the memory insufficiency error as shown in line 4 of the algorithm, otherwise, it forwards and calls the real memory request API as shown in lines 6 to 11 of the algorithm. Lines 14 to 25 of the algorithm are the memory release process, which is entered and executed in the opposite flow to the memory request.

Table 2: Algorithm 1: Quota-based GPU Memory Resource Isolation and Limit Algorithm.

Algorithm 1	**Quota-based GPU device memory (HBM/GDDR) resource isolation and quota algorithm**
Input:	Container number i, application or release of device memory (HBM/GDDR) size req, specified device memory (HBM/GDDR) resource quota limit
Output:	The result of the call for or release of device memory (HBM/GDDR) resources
1:	function cuMemAlloc(i,req,limit)
2:	quota ← quotaMap[i]
3:	if req + quota > limit then
4:	return CUDA_ERROR
5:	end if
6:	realFunction ← realFunctionMap[cuMemAlloc]
7:	result ← realFunction(req)
8:	if result == CUDA_SUCCESS then
9:	quotaMap[i] ← quotaMap[i] + req
10:	end if

11:	return result
12:	end function
13:	
14:	function cuMemFree(i,req)
15:	quota ← quotaMap[i]
16:	if req > quota then
17	return CUDA_ERROR
18:	end if
19:	realFunction ← realFunctionMap[cuMemFree]
20:	result ← realFunction(req)
21:	if result == CUDA_SUCCESS then
22:	quotaMap[i] ← quotaMap[i] - req
23:	end if
24:	return result
25:	end function

(2) Monitoring-based Segregation and Quota Algorithm for GPU Computing Resources.

Implementing a segregation and quota algorithm for GPU compute resources is more complex. The amount of GPU computing resources used by a container can be measured by the GPU utilization rate. When the container's GPU utilization rate is higher, the more time the container occupies GPUs in a time cycle, and the more CUDA core functions and execution time are initiated by the high-performance computing tasks in the container [14]. Therefore, the idea of this paper for the isolation and quota of GPU computing resources is based on the monitoring and adjustment algorithm, through the use of Nvidia related tools, real-time monitoring of the container's GPU utilization rate, dynamically adjusting the rate of the container's high-performance computing tasks to start the CUDA core function, so that the actual GPU utilization rate is roughly equal to the size of the container's specified computing resource quota.

During the runtime of a high-performance computing task such as deep learning, the framework executes a computational graph model, runs a large number of operators serially, and launches a large number of CUDA kernel functions on the bottom layer. Usually deep learning tasks use a large number of operators, which are of different numbers and types, and the execution time of different operators is also different. Through experimental analysis, the execution time of a CUDA kernel function is related to the time complexity of the algorithm implemented by the kernel function and the number of Blocks and Threads in the programming model.

In the isolation and quota algorithm for GPU computing resources, the startup rate of CUDA kernel functions is adjusted according to the real-time GPU utilization rate to ensure that the actual GPU utilization is approximately equal to the size of the computing resource quota specified by the container. In order to make this process smoother and the GPU utilization fluctuation smaller, this paper quantifies the execution time of the kernel function and the available time of the GPU.

The monitoring-based algorithm for GPU compute resource isolation and quota consists of two main parts. The first part is a real-time GPU utilization monitoring program, which monitors the GPU utilization and compares the difference between the utilization and the specified quota, and increases or decreases the available time quota of the GPU according to the difference. The second part is the rate limiting program of the core function, which calculates the execution time quota of the core function by obtaining the size of its Grid and Block parameters, and then compares it with the available time quota to determine whether to start the core function immediately or not.

The GPU's available time allowance can be viewed as a globally maintained resource that operates in a Producer Consumer Pattern. The real-time GPU utilization monitor is the producer, increasing the available time quota when the GPU utilization falls below a specified value; the core function is the equivalent of the consumer, decreasing the time quota when the core function is started. In order to ensure that the effect of GPU resources and limits can be achieved, it is necessary to strictly ensure that the production rate of the producer is not lower than the consumption rate of the consumer, or else starvation phenomenon will occur, so that the GPU utilization rate can not reach the target value.

In summary, the monitoring-based GPU computing resource isolation and quota algorithm is shown in Algorithm 2, which contains two parts of the algorithm, the monitoring program for real-time GPU utilization and the rate-limiting program for the core function. Lines 1 through 12 of the algorithm are the real-time GPU utilization monitor section, which inputs the container information and the limit amount, and increases the amount of available time when the current GPU utilization is lower than thelimit amount, as in lines 5 through 6 of the algorithm, and decreases the amount of available time otherwise, as in lines 7 through 8 of the algorithm. Lines 14 to 28 of the algorithm are the rate-limiting procedure part of the core function, which inputs the size of the Gird and Block parameters of the core function and calculates the amount of time consumed by the core function, compares it with the amount of available time, and performs a sleep operation when the resource amount is not enough, as in lines 16 to 21 of the algorithm, otherwise calls the real core function startup API, as in lines 25 to 27 of the algorithm.

Table 3: Algorithm 2: Monitoring-based GPU Computing Resource Isolation and Quota Algorithm.

Algorithm 2	**GPU computing resource isolation and quota algorithm based on monitoring**
Input:	Container number i, Grid &Block of the kernel function, and the specified computing resource limit
Output	The call result of the kernel function startup
1:	**function** utilizationMonitor(i,limit)
2:	quota ← global()
3:	**while** True **do**
4:	utilization ← getUtilization(i)
5:	**if** utilization < limit **then**

6:	quota ← addQuota(limit – utilization)
7:	**else:**
8:	quota ← subQuota(limit – utilization)
9:	**end if**
10:	sleep()
11:	**end while**
12:	**end function**
13:	
14:	**function** cuLaunchKernel(grid,block)
15:	quota ← global()
16:	exeTime ← getExeTime(grid,block)
17:	**while** True **do**
18:	**if** exeTime > quota **then**
19:	sleep()
20:	**continue**
21:	**end if**
22:	quota ← quota - exeTime
23:	**break**
24:	**end while**
25:	realFunction ← realFunctionMap[cuLaunchKernel]
26:	result ← realFunction(req)
27:	**return** result
28:	**end function**

The design objectives and evaluation of the three algorithms discussed in this summary are summarized as follows: 1. **Video Memory Resource Modeling**: The primary goal is to establish an accurate video memory usage tracking model, providing a theoretical foundation for achieving strong isolation and fairness. The core objective is to ensure that both global and individual container video memory usage never exceeds limits. Evaluation methods and metrics: - **Accuracy**: The model accurately reflects real-time video memory consumption of containers. - **Strictness**: Formula (2) is validated through stress testing to ensure no GPU global video memory overflow occurs. - **Isolation**: Video memory operations between containers are mutually exclusive (indirectly verified in Section 4.3 of the experiment).

2. The quota-based video memory isolation and limiting algorithm is designed to enforce strict isolation and fairness through a time-based mandatory policy. It ensures containers cannot breach their video memory quotas in any way, providing predictable exclusive memory space. Evaluation metrics: Quota accuracy: The actual peak video memory usage of containers consistently remains below their LimitMemory_i quota (Formula 3). Functional correctness: Tasks within the quota execute successfully, while those exceeding the quota fail immediately. Performance impact: The algorithm introduces minimal virtualization overhead (evaluated in Section 4.1). Non-interference: When video memory is sufficient, quota size has no impact on task performance (Section 4.2, Figure 9).

3. Monitoring-Based Computing Resource Isolation and Quota Allocation Algorithm (Algorithm 2). The design objective is to develop a dynamic feedback control system that achieves long-term weak fairness while tolerating inherent NVIDIA interface errors. Through monitoring and adjustment, the average GPU utilization of containers converges to their allocated quotas. Evaluation metrics: Average compliance rate: During prolonged operation, the average GPU utilization of containers approaches their quota LimitComputation_i (Formula 5). Dynamic response and stability: The system can rapidly stabilize utilization rates during load fluctuations, with controlled volatility (δ). Isolation effect: The extent to which load fluctuations in one container impact another's performance (Experiment 4.3, Figures 10,11, Table 7). Efficiency: Compared to static allocation, improvements in cluster throughput and average GPU utilization (see abstract).

4 Experimental validation

In this subsection, the resource isolation and quota algorithm based on API interception and forwarding proposed in this paper is experimentally validated. To illustrate the effectiveness of the resource isolation and quota algorithm, three experiments are conducted to evaluate the performance of virtualizationoverhead, resource quota, and resource isolation respectively. Some of these experiments are compared with the latest related work KubeShare [15,16].

4.1 Virtualization overhead

In this paper, we define the virtualization overhead as the ratio of the additional running time of the container after resource isolation and quota to the time of the container running directly by executing the same HPC task, as shown in Equation 6. Where $native\_time$ refers to the time spent by the container running directly, and $virtual\_time$ refers to the time spent using all the device memory (HBM/GDDR) resources and compute resources after the virtualization resource isolation and quota algorithm.

$$Overhead = \frac{virtual_{time} - native_{time}}{native_{time}} \times 100\%$$

The experimental environment is shown in Table 4.

Table 4: Virtualization overhead and resource limits for experimental environments

	Kubernetes Worker Node information
Operating system	Ubuntu 18.04
Memory size	128GB
CPU model	Intel(R) Core(TM) i9-10900X,3.70GHz
GPU model	Nvidia RTX2080Ti,11GB
Nvidia driver version	515.65.01
CUDA version	11.7
Test container information	
CUDA version	10.0
Deep learning programming framework	PyTorch 1.2
Deep learning task	Training tasks such as ResNet18

The test container adopts the training task of ResNet18 model, which calls the CUDA-related API of GPU for acceleration. The deep learning training process is usually divided into a number of rounds (epoch), the execution time of each epoch can reflect the performance of the current deep learning task, the shorter the epoch time, the higher the performance of the deep learning task.

Therefore, the corresponding virtualization overhead can be calculated by testing and comparing the average epoch times in the three cases of runningthe test container directly, running the test container using the algorithm in this paper, and running the test container using the algorithm in KubeShare. The experiments in this section run the deep learning training container, in which the number of epochs is set to 60, and the experimental results are shown in Figure 6.

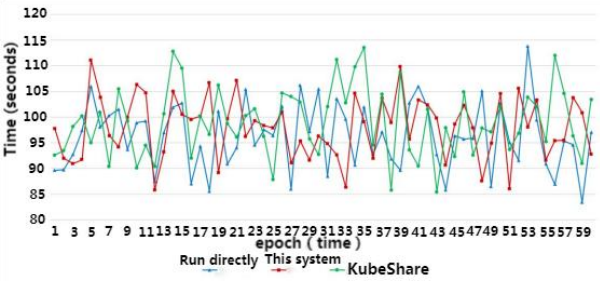

Figure 6: Execution time of container epochs after virtualization.

The experiments revealed that there are some fluctuations in the running time of the test container for each epoch. When running the test container directly, the average execution time of the epoch is 96.26 seconds, i.e., $native\_time$ = 96.26 seconds.

When running the test container using the algorithm in this paper, the execution time of the average epoch is 97.94 seconds, i.e., $virtual\_time$ this system = 97.94 Seconds. When running the test container using KubeShare's algorithm, the average epoch's execution time is 98.96 seconds, i.e. $virtual\_timeKubeShare$ = 98.96 seconds. The virtualization overhead of the system implemented in this paper and the resource isolation and quota algorithm of KubeShare is calculated based on Equation 6 as shown in Table 5.

Table 5: Virtualization overhead for this system and the KubeShare algorithm.

	native_time	virtual_time	Virtualization overhead
This system	96.26 seconds	97.94 seconds	1.75%
KubeShare	96.26 seconds	98.96 seconds	2.80%

Based on the above experimental results, we can get the experimental conclusion that the overhead of resource isolation and quota algorithms of both this paper and KubeShare is less than 3% within the error range, and the container performance loss caused by virtualization overhead is acceptable.

4.2 Resource quota

After allocating a specified amount of device memory (HBM/GDDR) resources and computation resources for a container, a resource limit is required to ensure that, within a certain fair range, the container receives the corresponding number of resources. The following two experiments test the impact of compute resource quota and device memory (HBM/GDDR) resource quota on the performance of containers respectively. The experimental environment is shown in Table 2.

The following experiments test the impact of the computational resource limit on the performance of the containers for the training task of the ResNet18 model used for the test containers.

Before conducting the test, the performance of the test container is first evaluated. The GPU utilization of the test container is viewed using the nvidia-smi tool, and the experiment reveals that the GPU utilization of the test container does not reach 100%, and the GPU utilization of the test container within 100 seconds is shown in Figure 7

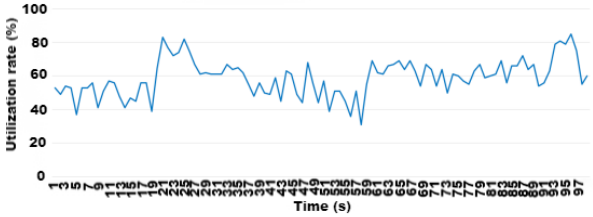

Figure 7: Test container GPU utilization for 100 seconds.

The average GPU utilization rate is calculated to be 60%. The test containers are allocated different amounts of computing resources from 10% to 100%, and ensure sufficient device memory (HBM/GDDR) resources, test the execution time of three epochs, and take the average value, reflecting the impact of computing resource allocation on the performance of containers. The results of the system implemented in this paper and KubeShare are shown in Figure 8.

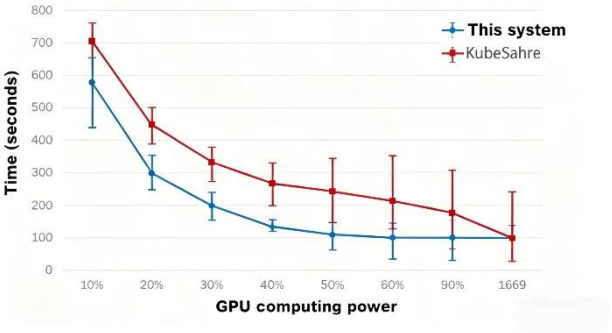

Figure 8: Impact of computing resource allocation on container performance.

Experiments found that the performance of the test container operation is related to the amountions of computational resources allocated. Using the computation resource quota algorithm in this paper, when the allocated compute resources are less than or equal to 60%, the epoch execution time and compute resources are strictly

inversely proportional to each other, and the performance of the container remains stable when the computation resources are in excess. With KubeShare's algorithm, when the allocated computing resources are less than or equal to 60%, the epoch execution time and computing resources are also roughly inversely proportional to each other, but the epoch execution time is higher than that of this paper's algorithm for the same computing resources, and when the allocated computing resources are greater than 60%, i.e., when the computing resources are in excess, the epoch execution time is still inversely proportional to the computing resources. Epoch execution time still decreases with the increase of computing resources.

The following experiments test the impact of the device memory (HBM/GDDR) resource limit on the performance of containers, and the test containers use the training tasks of ResNet18, Transformer, and GRU models. Different amounts of device memory (HBM/GDDR) resources are allocated for the test containers respectively, and ensure sufficient computational resources, all of which are 100%, and test the execution time of three epochs and take the average value to reflect the impact of device memory (HBM/GDDR) resource allocation on the performance of the containers. The experimental results of the system realized in this paper are shown in Fig. 9.

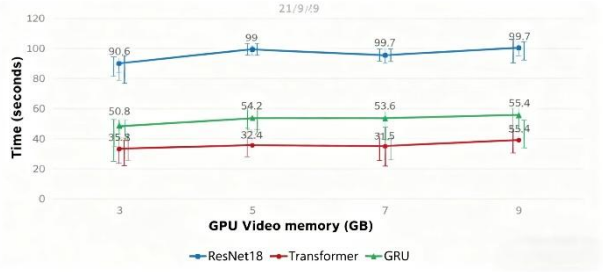

Figure 9: Impact of device memory (HBM/GDDR) resource allocation on container performance.

The experiments found that the performance of the test container is independent of the allocation of memory resources, and that when the memory resources are less than a certain value, the program fails to run and throws an exception for insufficient memory. The test found that KubeShare also has similar experimental phenomena.

Based on the above experimental results, it can be concluded that the resource limit algorithms of this paper and KubeShare are both effective. In terms of computation resource quota, the monitoring-based GPU compute resource isolation algorithm in this paper is better than the time slice rotation algorithm in KubeShare. In terms of device memory (HBM/GDDR) quota, both this paper's and KubeShare's algorithms have good quota effects and do not affect the performance of containers.

4.3 Resource isolation

Resource isolation between containers refers to the extent to which allocating resources to containers has a performance impact on other containers. A good resource isolation means that no matter how many containers are started at the same time, and no matter how much GPU resources are allocated between containers, there will not be any performance impact. According to the experimental results in 3.2, the amount of device memory (HBM/GDDR) resources does not significantly affect the performance of containers, so the experiments in this section mainly test the isolation effect of computing resources.

The experimental environment is shown in Table 6.

Table 6: Resource isolation experiment environment.

	Kubernetes Worker node information
Operating system	Ubuntu 18.04
Memory size	384GB
CPU model	Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz
GPU model	Nvidia Tesla T4,16GB
Nvidia driver version	515.43.04
CUDA version	11.7
Test container information	
CUDA version	10.2
Deep learning programming framework	PyTorch 1.9.0
Deep learning task	Training tasks such as ResNet18

The following experiment tests the effect of compute resource isolation between simultaneously deployed containers. At the beginning of the experiment, three containers containing different deep learning training tasks are deployed to the GPU at once, and the deep learning models, model domains, and GPU resource allocations used by the containers are shown in Table 7. The GPU utilization of all three containers exceeds 95% without any computational resource limitation. The real-time GPU utilization of each container is monitored using the GPU utilization monitoring interface provided by Nvidia.

Table 7: Experiment Containers and GPU Resource Allocations.

Container	**Model domain**	**Computing resources**	**Device memory (HBM/GDDR) resources**
ResNet18	Image classification	45%	3GB
Transformer	Natural language	30%	3GB
GRU	Natural language	15%	3GB

The GPU utilization of the three containers over a 60-second period is shown in Figure 10.

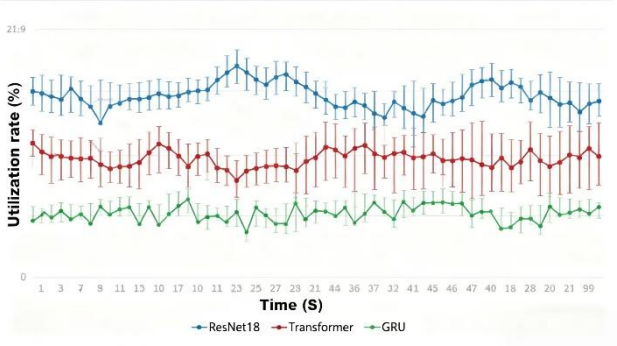

Figure 10: GPU utilization with container sharing.

It was found that the GPU utilization rates of the three test containers fluctuated to some extent, but all of them fluctuated above and below the amount of computing resources specified by the containers. The average GPU utilization rate of the ResNet18 task is 44.08%, the average GPU utilization rate of the Transformer task is 29.13%, and the average GPU utilization rate of the GRU task is 15.27%, which is very close to the specified amount of GPU computational resources of the three containers, which indicates that there is almost no interference among them.

The following experiment tests the isolation of computing resources between successively deployed containers. At the beginning of the experiment, a target container is deployed with the Transformer training task running, and then additional containers are deployed every 3 epochs according to the time interval to observe the performance fluctuation of the target container. In order to minimize the error, GPU resources need to be reserved for tools such as nvidia-smi, so the total GPU computing resources allocated to all containers is 90%, and the deep learning models, model domains, and GPU resource allocations of the target container and additional containers are shown in Table 8.

Table 8: Experiment Containers and GPU Resource Allocations.

Container	Model domain	Computing resources	Device memory (HBM/GDDR) resources
Transformer	Natural language	50%	3GB
ResNet18	Image recognition	10%	3GB
ShuffleNet V2	Image recognition	10%	3GB
MobileNet V3	Image recognition	10%	3GB
GRU	Image recognition	10%	3GB

The performance of the target container fluctuates as shown in Figure 11

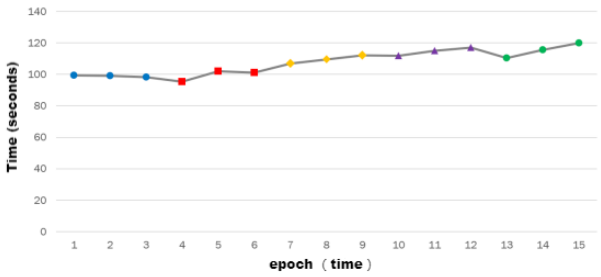

Figure 11: Performance impact of the target container when deploying additional containers.

The experiments found that the performance of the target container degraded as additional containers were deployed, and the more containers sharing GPUs, the greater the performance loss of the target container, as shown in Table 9. When the number of shared containers is greater than or equal to 3, the performance of the container receives a more obvious impact, but the performance loss is about 15%, which is acceptable.

Table 9: Performance loss of the target container when deploying additional containers.

The number of shared containers	Average epoch time	Performance loss
1	98.89s	0.00%
2	99.38s	0.50%
3	109.46s	10.69%
4	114.56s	15.85%
5	115.27s	16.56%

Based on the above experiments, it can be concluded that the resource isolation algorithm in this paper has a good isolation effect when allocating GPU computing and GPU memory resources for containers with less performance impact on other containers.

At the same time, the experimental results also illustrate some problems: when containers share GPUs and allocate the same share of GPU computing resources, the actual computing resources obtained by each container are lower than those when they share GPUs exclusively. And the more containers share GPUs, the lower the actual GPU computing resources the containers get. Therefore, when sharing GPUs, it is usually necessary to compensate for the resources.

Interference between containers causes a loss of performance, and there are two possible reasons for analyzing this. The first possible reason is the underlying sharing mechanism of the GPU. According to the introduction in the second part of this paper, due to the concurrent execution between GPU multi-tasks, which is actually still executing serial time slices under the microscopic point of view, its task scheduling, task context switching, and other GPU resource contention between tasks will lead to time overhead. The second possible reason is that there is still room for improvement in the specific implementation of the monitoring-based resource isolation and quota algorithm. Under multi-

tasking, the accuracy of the GPU utilization monitoring interface provided by Nvidia is not high enough, and the accuracy of resource isolation can be further improved by using a more accurate GPU utilization monitoring interface or algorithm.

5 Discussion

The experimental results demonstrate that the resource isolation and quota algorithm based on API interception and forwarding proposed in this paper achieves the expected outcomes in terms of virtualization overhead, resource allocation, and isolation. This section will provide a deeper analysis of these results and explore the broader implications and future directions of this research.

5.1 Method effectiveness and nonlinear pattern

Analysis The experimental results demonstrate that our system exhibits lower virtualization overhead (1.75% vs. 2.80% for KubeShare) while demonstrating superior inverse proportionality control in computing resource allocation (Figure 8). This advantage fundamentally stems from our method's adaptability to GPU workload nonlinear patterns.

As the problem analysis demonstrates, resource demands for tasks like AI/ML training are inherently dynamic and nonlinear. KubeShare's static time slice rotation algorithm, being an open-loop linear scheduling strategy, struggles to respond to instantaneous task fluctuations. Our monitoring-based adjustment algorithm (Algorithm 2), however, operates as a closed-loop feedback control system. By continuously monitoring GPU utilization (system output) and comparing it with predefined quotas (reference input), it dynamically adjusts kernel function activation rates (control input). This feedback mechanism enables automatic tracking and response to nonlinear disturbances introduced by tasks, ensuring precise and stable control under varying computational loads. Consequently, it achieves superior efficiency and fairness in long-term statistical performance.

5.2 Bottlenecks and theoretical implications of resource isolations

The resource isolation experiment (Section 3.3) demonstrates a critical phenomenon: while our algorithm effectively ensures each container receives its average resource allocation, the absolute performance of all containers declines as the number of shared containers increases (Table 7). This precisely exposes the limitations of the current approach as reactive control.

The root cause of performance degradation lies in two dimensions: the nonlinear overhead of underlying hardware. The GPU's core context switching and resource contention overheads exhibit nonlinear growth, escalating dramatically with concurrent tasks. These are inherent limitations that our algorithm cannot fully eliminate in user space.

The inherent delay of the control system: Our monitoring and adjustment algorithm relies on the measurement of utilization over a "past" period, which is a delayed and lagging control. It cannot predict the arrival of the next computation-intensive phase, thus failing to make optimal scheduling arrangements in advance.

This discovery strongly points to the next research direction: transitioning from reactive control to predictive control. The theories of optimal control and adaptive control provide us with theoretical tools. For instance, we can build nonlinear predictive models based on task execution patterns identified from API call sequences. A scheduler based on model predictive control (MPC) can forecast resource demands for each task over a future period and pre-solve the optimal resource allocation scheme, thereby proactively avoiding resource conflicts and fundamentally reducing performance losses caused by underlying overhead.

5.3 Path to universal intelligent scheduling

The success of this study demonstrates the feasibility of applying nonlinear recognition theory and feedback control mechanisms to GPU resource management. Our API interception layer, functioning as an efficient "sensor" network, enables dynamic online recognition of complex workloads. This lays the foundation for building a truly intelligent and adaptive cloud-native computing platform.

As outlined in Section 4, future work should focus on expanding and refining this framework. At the recognition level, it is essential to extend the system to heterogeneous GPU environments, identifying the distinct dynamic characteristics exhibited by different hardware architectures when handling the same task.

At the model level, we need to evolve from the current simple quantitative model to a nonlinear prediction model that can capture long-term dependencies.

At the control level, the system requires upgrading from conventional PID feedback control to optimal control based on model predictive control (MPC) or reinforcement learning (RL).

In conclusion, this paper not only presents an effective GPU virtualization system, but more importantly, it implements and validates a "identification-modeling-control" research paradigm. This paradigm transforms random and chaotic workload behaviors into identifiable, modelable, and ultimately optimizable objects, providing a clear and promising technical pathway to address increasingly complex computing environments and heterogeneous hardware in the future.

5.4 Control of nonlinear system

The paper gives six references about nonlinear system control, especially chaos system, adaptive control and intelligent control.

These studies collectively outline the evolutionary trajectory of nonlinear control theory in addressing challenges posed by "uncertainty" and "complexity." The core shift lies in transitioning from precise model-based control to robust control strategies that autonomously learn, estimate, and compensate for unknown system

components (e.g., unmodeled dynamics, external disturbances, and parameter uncertainties). Key technical approaches include adaptive control, backstepping, intelligent control (fuzzy logic, neural networks), and specialized designs for specific scenarios (e.g., synchronization, nonlinear inputs, and actuator physical constraints).

The core contribution of Literature 1 on adaptive fuzzy control for fixed-time synchronization in fractional-order chaotic systems lies in its integration of multiple cutting-edge concepts.

Fractional order system: It can describe some materials and processes with memory and heredity characteristics more accurately than integer order model.

Chaos synchronization is a key technology in the fields of secure communication and biological rhythm analysis.

The actual fixed time stability is more practical than the asymptotic stability and the finite time stability, because it requires that the convergence time of the system has an upper bound independent of the initial state, which is convenient for the performance evaluation of the system.

Adaptive fuzzy control: using fuzzy logic system to approximate the unknown nonlinear function in the system, and adjusting the parameters online by the adaptive law, without knowing the precise nonlinear model in advance.

The core contribution of this study is to address several practical constraints in control engineering.

Projection lag synchronization is a kind of generalized synchronization, which means there is a constant time delay between the state of the driving system and the response system. It can be used in communication for prediction or security coding.

Nonlinearity: The actual actuators (e.g. amplifier, valve) have nonlinearity characteristics such as saturation, dead zone, hysteresis, etc. Ignoring these characteristics will lead to performance degradation or even instability.

Output feedback: This is a critical point. It assumes that the entire state of the system is unobservable and can only be controlled through a finite set of output signals, which is more realistic than state feedback (assuming all states are known).

Reference 3. The core contribution of robust neural adaptive control for a class of uncertain nonlinear complex dynamic multivariable systems: This is a foundational and methodologically robust study.

Multivariable system: the main challenge of its control is the coupling effect of each channel in the system.

Robust neural adaptive control: Similar to fuzzy logic, neural networks (NNs) are another powerful universal approximator, used for online learning and compensation of uncertain nonlinear functions and coupling effects in systems. The term "robust" typically refers to techniques such as sliding mode control to address approximation errors and external disturbances.

Reference 4. The core contribution of adaptive backstepping control for a class of uncertain single-input single-output nonlinear systems: This represents a milestone method in nonlinear adaptive control.

The inverse step method is a recursive design tool for systems with strict feedback form or convertible to similar form. It derives the real control input and the adaptive law of parameters by designing the virtual control law and the stability function step by step.

SISO system: As the foundation of MIMO systems, the control design of SISO systems provides the basis for understanding more complex control methods.

The core contribution of nonlinear optimal control for induction motor driven gas compressor is a study for specific industrial application.

Induction motor is the most widely used motor in industry, but its model is multivariable, nonlinear and strongly coupled.

Gas compressor is a typical nonlinear load with complex dynamic characteristics.

Nonlinear optimal control: the goal is not only to stabilize the system, but also to achieve "optimal" under certain performance criteria (such as minimum energy consumption, maximum efficiency, and fastest response).

The core contribution of the adaptive backstepping control for a single-link manipulator driven by a DC motor is to solve the control problem of a typical distributed parameter system.

Flexible robot manipulator: The model is described by partial differential equation, because the linkage is flexible, its deformation is continuous. This is much more difficult than the control of rigid robot (described by ordinary differential equation).

Flexible vibration is the main challenge of the control, which requires the control of the motor rotation and the suppression of the elastic vibration of the connecting rod.

The six papers constitute a complete spectrum from basic theory to specialized problems and then to engineering applications: 1. Methodological foundation: Papers 4 (Adaptive Backstepping Method) and 3 (Neural Adaptive Control) provide the core tools for dealing with uncertain nonlinear systems.

2. Problem deepening: Using these tools, both literature 1 and literature 2 tackle more advanced and specific challenges. Specifically, literature 1 focuses on fractional-order systems and fixed-time convergence performance, while literature 2 simultaneously addresses practical engineering constraints (output feedback and nonlinear inputs) and special synchronization objectives (lag synchronization).

3. Engineering Applications: References 5 and 6 apply nonlinear control theory to two classical electromechanical systems. Reference 5 focuses on energy efficiency (optimal control), while Reference 6 addresses vibration control challenges caused by structural flexibility.

These studies collectively demonstrate the robust vitality of modern control theory: by integrating adaptive mechanisms, intelligent approximators (fuzzy/neural networks), and advanced nonlinear design methods (backstepping), it can provide effective, robust, and high-performance control solutions for a wide range of complex, uncertain, and nonlinear engineering systems.

5.5 Evaluating NVML-based metrics

To address the noise and fairness issues in current GPU utilization monitoring systems based on sampling, future research could optimize through two approaches: control theory and advanced monitoring metrics. The current algorithm (Algorithm 2) fundamentally operates as a proportional (P) controller. Implementing a proportional-integral-derivative (PID) controller represents a direct improvement. The integral component (I) effectively eliminates steady-state errors, ensuring containers maintain precise average utilization rates equal to their quotas over extended periods, thereby enhancing fairness. The derivative component (D) predicts utilization trends, enabling proactive regulation that reduces overshoot and fluctuations, resulting in smoother and more stable GPU utilization control.

Adjusting sampling and evaluation strategies: Appropriately extending the sampling interval ΔTΔT (Formula 5) for fairness assessment can effectively smooth short-term fluctuations and yield more stable average utilization estimates. However, this requires balancing control response speed and control accuracy. An adaptive ΔTΔT adjustment strategy may represent a viable compromise worth exploring.

2. Evaluate more granular NVML metrics. We also explored the possibility of adopting better monitoring metrics in the "Future Work" section.

Beyond optimizing control algorithms, it is equally crucial to explore underlying metrics that provide more precise insights than overall GPU utilization. NVIDIA NVML offers granular performance counters such as Stream Multiprocessor Occupancy (SM Occupancy), which measures the theoretical utilization of computing threads executing on SMs. Compared to overall GPU utilization, SM Occupancy more directly reflects the saturation level of computing units and is less affected by factors like memory access latency.

However, in practical applications, such metrics also face challenges: First, their acquisition and computation costs are typically higher; second, an effective mapping model between these new metrics and the actual performance of applications (such as task throughput) needs to be established. Therefore, systematically evaluating a series of NVML metrics including SM occupancy and selecting the optimal metric in terms of accuracy, cost, and practicality as feedback signals are key steps in building next-generation high-performance GPU shared systems.

5.6 API Interception risks and limitations

1. Forward compatibility and CUDA version dependency. The core of this solution involves intercepting and forwarding NVIDIA CUDA driver-level APIs. However, NVIDIA does not guarantee that the binary interface (ABI) of these underlying APIs remains stable across different CUDA major versions.

Risk: When CUDA drivers are upgraded on the node, the API function signatures, data structures, or behaviors may undergo unannounced changes. This could cause our interceptor library to become incompatible with the new drivers, resulting in interception failures, incorrect parameter forwarding, or even segment faults that crash container tasks.

Mitigation and Outlook: To ensure long-term system availability, it is essential to establish a continuous integration testing pipeline covering mainstream CUDA versions (e.g., 11.x,12.x). Any detected incompatibilities should trigger immediate updates to the interceptor code. From an architectural perspective, future enhancements could include implementing a lightweight ABI compatibility layer within the interceptor. This layer would adapt to different driver interfaces through runtime version detection, thereby enhancing system robustness.

2. Scalability in Heterogeneous GPU Clusters Our research currently focuses on single GPU blocks within a single node. However, production clusters typically consist of heterogeneous GPU models (e.g., V100, A100, H100).

3. Intercepting low-level driver APIs poses unique security and stability risks. These APIs operate in user mode but interact closely with kernel-mode drivers.

Stability risks: Our interceptor library operates within the same address space as the application. If the interceptor code itself contains defects (such as memory leaks or race conditions), it may not only affect the interceptor itself but also destabilize or even crash the entire container process. This creates a wider fault domain compared to virtualization solutions running in kernel mode or as standalone processes (e.g., SR-IOV-based vGPU).

Security Impact: Elevated Privileges: This interceptor library essentially grants the capability to inspect, modify, or discard all CUDA API calls. A maliciously altered interceptor may enable model theft, gradient poisoning, or computation result tampering, posing a severe supply chain security threat.

Expanding attack surface: Any vulnerability found in the CUDA driver API could be exploited or amplified through our interceptor.

Mitigation strategy: Strict engineering practices must be implemented to manage these risks, including but not limited to: digitally signing and verifying deployed interceptor binaries; pinning their versions during container image build processes and conducting vulnerability scans; and enforcing the least privilege principle to prevent containers from running with excessive permissions.

Summary: API interception is a highly flexible yet complex technical approach. While enabling granular resource control on generic hardware, it also requires maintaining compatibility, scalability, and security. This analysis does not invalidate the solution but aims to refine it for smoother deployment in production environments.

5.7 Experimental design limitations

Regarding modifications to address experimental design limitations, the reviewer's comments emphasized: "All experiments utilized a single ResNet18 model with fixed dataset sizes. To enhance generalizability, at least one experiment incorporating models of varying scales or

batch sizes should be included to evaluate sensitivity. The training batch sizes and input dimensions were not specified – which is essential for reproducibility. Container orchestration scenarios assume static quotas. Consider discussing dynamic scheduling based on observed workload characteristics." Specifically, we will address: the system's capability to monitor real-time resource utilization (GPU usage, memory consumption) and task progress (e.g., training loss reduction rate) for each container, and design a feedback controller. When detecting that a container's resource demand consistently falls below or exceeds its quota, the scheduler can dynamically adjust resource allocations or "lend" idle resources to tasks requiring more capacity, thereby achieving higher resource utilization and throughput at the cluster level.

5.8 Scalability considerations

We will prioritize enhancing system scalability as a core focus in our future development. By implementing dynamic grouping scheduling or workload-aware consolidation strategies, we aim to consolidate numerous fragmented computing requests into optimized batches. This approach reduces context switching frequency, thereby maintaining high overall performance while supporting a greater number of containers.

Quantified the scalability boundary: Through experiments, the maximum number of recommended containers supported by a single GPU was determined.

It reveals the performance bottleneck: the fundamental cause of nonlinear performance degradation is explained by the curve chart and theoretical analysis.

It points out the optimization path: it puts forward the specific future research direction to break through the current scalability limit.

It makes a clear understanding and prospect of its performance in a large-scale concurrent environment, and significantly improves the depth and practicability of the paper.

6 Conclusion and prospect

This paper primarily introduces a resource isolation and quota algorithm based on API interception and forwarding. First, we analyze the software and hardware architecture of NVIDIA GPUs, explaining the GPU virtualization approach adopted in this study, which relies on API interception and forwarding. We emphasize two key metrics—efficiency and fairness—guiding the subsequent algorithm implementation and experimental validation. Next, by examining the workflow of high-performance computing (HPC) tasks invoking CUDA APIs, we identify how these tasks utilize memory and computational resources, enabling us to model their GPU resource usage. Based on these models, we develop separate resource isolation and quota algorithms for memory and computational resources. For memory resources, we implement a quota-based allocation algorithm, while for computational resources, we employ a monitoring-based approach. Finally, we conduct experiments to evaluate the proposed solution in three aspects: virtualization overhead, resource quotas, and resource isolation. The results demonstrate that our algorithm performs well in terms of virtualization overhead and resource quota enforcement. However, due to the inherent overhead of GPU resource sharing at the hardware level and inaccuracies in computational resource monitoring interfaces, there remain certain limitations in resource isolation.

References

[1] Ji Z, Zhang X, Fu Z, et al. Performance-awareness based dynamic batch size SGD for distributed deep learning framework[J]. Journal of Computer Research and Development, 2019, 56(11): 2396-2409. (in Chinese) DOI: 10.7544/issn1000-1239.2019.20190610

[2] Peñaranda C, Reaño C, Silla F. A Parallel Compression Pipeline for Improving GPU Virtualization Data Transfers[J]. Sensors, 2024, 24(14): 4649-4649. DOI: 10.3390/s24144649

[3] Belkhiri A, Dagenais M. Analyzing GPU Performance in Virtualized Environments: A Case Study[J]. Future Internet, 2024, 16(3): 487-495. DOI: 10.3390/fi16030048

[4] Kumar S J N, Arun H C. A novel video compression model based on GPU virtualization with CUDA platform using bi-directional RNN[J]. International Journal of Information Technology, 2024, 16(1): 457-463. DOI: 10.1007/s41870-023-01266-1

[5] M. J C, Juan M, Baldomero I, et al. Using remote GPU virtualization techniques to enhance edge computing devices[J]. Future Generation Computer Systems, 2023, 14214-24. OI: 10.1016/j.future.2023.01.007

[6] Dhulipala L S, Casaprima N, Olivier A, et al. Harnessing distributed GPU computing for generalizable graph convolutional networks in power grid reliability assessments[J]. Energy and AI, 2025, 19(8): 471-471. DOI: 10.1016/j.egyai.2024.100371

[7] Adam K, Paweł C, Jerzy P. Dynamic GPU power capping with online performance tracing for energy efficient GPU computing using DEPO tool[J]. Future Generation Computer Systems, 2023, 145(5): 396-414. DOI: 10.1016/j.future.2023.05.026

[8] Hua Q, Qian S Y, Yang D Y, et al. Qore-DL: a QoSaware joint optimization framework for distributed deep learning training [J]. Journal of Systems Architecture, 2022, 130: 102640. DOI: 10.1016/j.sysarc.2022.102640

[9] Tsog N, Mubeen S, Sjödin M, et al. A Trade-Off between Computing Power and Energy Consumption of On-Board Data Processing in GPU Accelerated In-Orbit Space Systems: Full Article[J]. Transactions of the Japan Society for Aeronautical and Space Sciences, Aerospace Technology Japan,2021,19(5):700-708. DOI: 10.2322/tjsass.19.700

[10] John A, Kawash J, Alhajj R. Predictive container orchestration in the cloud using artificial intelligence

techniques[J]. Computing, 2025, 107(7): 150-150. DOI: 10.1007/s00607-024-01684-9

[11] Zhu Z Y, Tang X C, Zhao Q. A unified schedule policy of distributed machine learning framework for CPU-GPU cluster [J]. Journal of Northwestern Polytechnical University, 2021, 39(3): 529-538. (in Chinese) DOI: 10.1051/jnwpu/202139030529

[12] Li X Z, Wang R Y, Mo J R. Application of improved genetic algorithm in cloud resource scheduling [J]. Journal of Guilin University of Aerospace Technology, 2022, 27(1): 9-13. (in Chinese) DOI: 10.13624/j.cnki.issn2095-1248.2022.01.002

[13] Open source code reference website(https://www.rcuda.net/)