SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

December 2024, Vol. 67, Iss. 12, 222101:1–222101:17
https://doi.org/10.1007/s11432-024-4227-2

MuxFlow: efficient GPU sharing in production-level clusters with more than 10000 GPUs

Xuanzhe LIU[1,2*], Yihao ZHAO[1,2], Shufan LIU[3], Xiang LI[3], Yibo ZHU[4], Xin LIU[3*] & Xin JIN[1,2*]

[1]*School of Computer Science, Peking University, Beijing* 100871*, China;*
[2]*Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing* 100871*, China;*
[3]*ByteDance, Beijing* 100006*, China;*
[4]*StepFun, Shanghai* 200232*, China*

Received 10 May 2024/Revised 19 August 2024/Accepted 19 November 2024/Published online 13 December 2024

Abstract Large-scale GPU clusters are widely used to speed up both latency-critical (online) and best-effort (offline) deep learning (DL) workloads. However, similar to the common practice, the DL clusters at ByteDance dedicate each GPU to one workload or share workloads in time dimension, leading to very low GPU resource utilization. Existing techniques like NVIDIA MPS provide an opportunity to share multiple workloads in space on widely-deployed NVIDIA GPUs, but it cannot guarantee the performance of online workloads. We present MuxFlow, the first production system that can scale over massive GPUs to support highly efficient space-sharing for DL workloads. MuxFlow introduces a two-level protection mechanism for both memory and computation to guarantee the performance of online workloads. MuxFlow leverages dynamic streaming multiprocessor (SM) allocation to improve the efficiency of offline workloads. Based on our practical error analysis, we design a mixed error-handling mechanism to improve system reliability. MuxFlow has been deployed at ByteDance on more than 18000 GPUs. The deployment results indicate that MuxFlow substantially improves the GPU utilization from 26% to 76%, SM activity from 16% to 33%, and GPU memory usage from 42% to 48%.

Keywords GPU cluster, deep learning workload, cluster management, GPU sharing, deployed system

1 Introduction

At popular internet-scale service providers like ByteDance, deep learning (DL) has been widely integrated into its applications and services, such as intelligent recommendation [1,2], image classification [3,4], and language processing [5,6]. Some of them provide real-time inference and have critical latency requirements (called online workloads). Meanwhile, other workloads do not have hard latency requirements (called offline workloads). Similar to other large enterprises, ByteDance builds large-scale GPU clusters with tens of thousands of heterogeneous GPUs for DL workloads. To satisfy the latency requirements of online workloads, we reserve specific GPUs for online workloads.

Existing efforts such as scale-up and scale-out in datacenters have significantly improved the serving efficiency of online workloads management [7–9]. However, a major limitation is that most solutions dedicate the entire GPU to a single workload. At ByteDance, we observe that an online workload usually cannot fully utilize the expensive GPU resources (Figure 1), which is also reported in previous studies [10, 11]. The reason is two-fold. First, the frequency of online requests fluctuates from time to time. When the request frequency is low, more GPU computation units are idle, leading to a great waste of resources. Second, even if the request frequency is high, the batch size of online workloads is usually limited to a small value to satisfy latency requirements. Thus, GPUs are still underutilized, which wastes expensive hardware, limits the capability of large-scale GPU clusters, and increases cluster cost.

A common idea is to share GPUs among multiple workloads with different latency requirements [11–13], i.e., sharing GPUs between online and offline workloads. Time-sharing and space-sharing are two

* Corresponding author (email: liuxuanzhe@pku.edu.cn, liuxin.ai@bytedance.com, xinjinpku@pku.edu.cn)

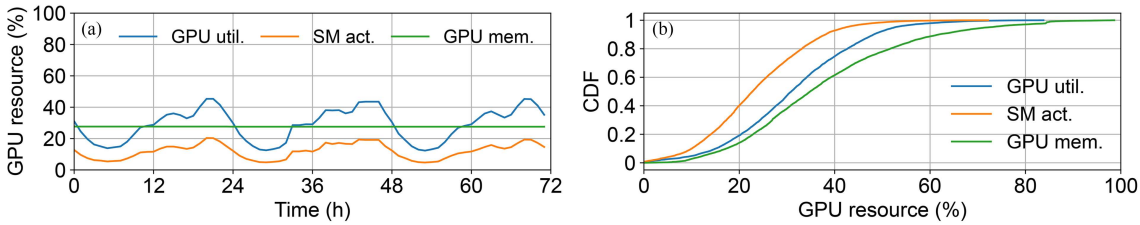

Figure 1 (Color online) (a) Resource usage of one typical online workload. GPU util., SM act., and GPU mem. are short for GPU utilization, SM activity, and GPU memory usage, respectively. (b) GPU resource statistics in a production cluster for online workloads.

paradigms for GPU sharing. Time-sharing [13] assigns time slices to different workloads, but it may degrade the performance of online workloads and cannot improve the GPU resource utilization in space. Space-sharing [11] is a better option to improve the GPU resource utilization. However, existing approaches for GPU sharing are not adequate for production clusters, because they violate three basic requirements: (1) Performance of online workloads. The primary goal for production clusters is to guarantee the performance of online workloads, such as real-time recommendation and machine translation. These workloads have hard latency requirements because longer latency deteriorates user experience. (2) Reliability. It is critical to ensure the reliability of production clusters, and thus GPU sharing should not lead to more failures of workload execution, especially for online workloads. (3) Practicability. Deploying GPU sharing faces multiple practical issues, e.g., usability, compatibility with existing infrastructure, and the ability to deal with specific features of commodity GPUs.

For widely-deployed NVIDIA GPUs, the multi-process service (MPS) is feasible for space-sharing due to its flexibility and compatibility. However, the native MPS cannot guarantee the aforementioned performance of online workloads and reliability. First, MPS shares GPU computation units without guaranteeing the performance of the shared workloads. Second, it is reported that MPS can incur serious error propagation problems [14]; i.e., when one workload encounters an error, the shared workload may also be influenced.

This paper presents MuxFlow, to the best of our knowledge, the first system that can scale over massive GPU clusters to support efficient and reliable GPU space-sharing for DL workloads in a real production environment. MuxFlow exploits a two-level protection mechanism to guarantee the performance of online workloads from both the workload level and GPU level. At the workload level, we propose ByteCUDA to constrain the GPU memory and computation power used by offline workloads. ByteCUDA monitors the GPU memory allocation to limit the memory usage of offline workloads, and controls kernel launches to limit the computing power used by offline workloads. ByteCUDA provides adjustable parameters to control how much the online workloads are influenced. At the GPU level, we use SysProbe to monitor the GPU status. The SysProbe maintains a state machine according to multi-dimensional GPU metrics, and evicts offline workloads when sharing can compromise the performance of online workloads. The workload-level ByteCUDA provides active and fine-grained control of offline workloads, while the GPU-level SysProbe provides an overall guarantee.

To guarantee the reliability of clusters, we investigate all propagated errors in our production clusters and find that the errors are extremely skewed. We find that 99% propagated errors are caused by SIGINT and SIGTERM signals, which are usually used to stop containers in Kubernetes. However, designing a single mechanism to cover all errors is impossible, especially in a complex production environment. Thus, we propose a mixed error-handling mechanism. For 99% propagated errors, MuxFlow employs a graceful exit mechanism that intercepts related signals and releases CUDA contexts actively. For other corner cases, MuxFlow automatically detects the errors according to our summarized error patterns and resets execution contexts.

Furthermore, MuxFlow improves the efficiency of offline workloads. MuxFlow dynamically allocates the computation units of NVIDIA GPUs, i.e., streaming multiprocessor (SM), to offline workloads. Our key intuition is that we can set the SM percentage of offline workloads complementary to the SM percentage used by online workloads, with an acceptable slowdown of online workloads.

The main contribution of this paper is to introduce the design of our practical cluster system for efficient and reliable GPU space-sharing. MuxFlow has been deployed at ByteDance on more than 18000 heterogeneous GPUs since 2021. MuxFlow serves tens of thousands of daily workloads. Deployment results show that MuxFlow improves the GPU utilization from 26% to 76%, SM activity from 16% to

33%, and GPU memory usage from 42% to 48%. We also share experiences from our deployment and unveil a set of unique challenges to inspire future research. First, we share our efforts on system reliability in the internal clusters and the challenges of generalizing it to cloud settings. Second, we share the design tradeoffs when we apply MuxFlow in production, including the slowdown of online workloads, the strategy of ByteCUDA, parameter settings, CPU and memory sharing, and the number of shared workloads.

2 Motivation

In this section, we begin with introducing DL workloads and important terminologies. Then we describe the observations from the clusters for online workloads at ByteDance to motivate the design of MuxFlow. We end by discussing opportunities and challenges to share GPUs.

2.1 DL workloads

DL workloads use deep neural networks (DNNs) to perform inference or training. DL workloads are usually classified into two categories, i.e., online workloads and offline workloads, according to the latency requirements.

Online workloads refer to latency-critical inference workloads, such as real-time recommendation, language processing, image classification, and speech recognition at ByteDance. Online workloads have strict latency requirements because longer end-to-end latency hurts the user experience. Online workloads usually have millions of requests for each workload every day. According to algorithm engineers at ByteDance, a 20% slowdown of latency can still meet the latency requirements for most online workloads.

We take one typical online workload in the production cluster of ByteDance as an example and show its GPU computation utilization and memory usage in Figure 1(a). Both the GPU utilization and SM activity fluctuate greatly in one day because the number of online requests varies from time to time. For example, more users use entertainment applications in the evening and send more online requests to related services. The GPU memory usage is stable because the DL framework, e.g., PyTorch [15], caches the intermediate state for efficiency. Besides, we observe that the curves of the GPU usage metrics are smooth in minutes and periodical in days. Thus, we can predict the GPU usage metrics based on history.

Offline workloads do not have strict latency requirements, such as DL training, batch inference over a large dataset, and scientific computing at ByteDance. The offline workloads do not have hard time requirements and can usually highly utilize the computation units of GPUs [16], making them suitable to fill the idle GPU resources.

2.2 Production cluster for online workloads

Production clusters use GPUs to accelerate DL workloads [7, 8, 11]. GPUs are usually assigned to online workloads exclusively to satisfy the latency requirements. We study GPU resource utilization in production clusters from two aspects: computation utilization and memory usage.

Low GPU resource utilization. We collect one week's statistics of GPU computation utilization and memory usage in the inference cluster of ByteDance, as shown in Figure 1(b). As for the GPU computation utilization, we use two metrics: GPU utilization and SM activity. GPU utilization and SM activity represent how busy a GPU is in time and space, respectively. GPU memory usage is the ratio of used memory to memory capacity. Figure 1(b) illustrates that both GPU utilization and SM activity are lower than 60% for more than 99% GPUs. In addition, GPU memory usage is less than 60% for about 90% GPUs. These numbers show that GPUs are underutilized in both computation and memory, indicating a large waste of valuable GPU resources.

2.3 Opportunities in GPU sharing

Recent studies [11, 13] have exploited GPU sharing approaches to improve GPU resource utilization and save resources. There are two aspects of GPU sharing, i.e., time-sharing and space-sharing. We compare GPU sharing approaches for widely-deployed NVIDIA GPUs with an example as shown in Figure 2.

Time-sharing is not efficient in improving GPU resource utilization. In time-sharing, shared workloads use different time slices. To protect the performance of online workloads, priority-based time-sharing [13] (Figure 2(a)) assigns more time slices to high-priority workloads. However, a single online

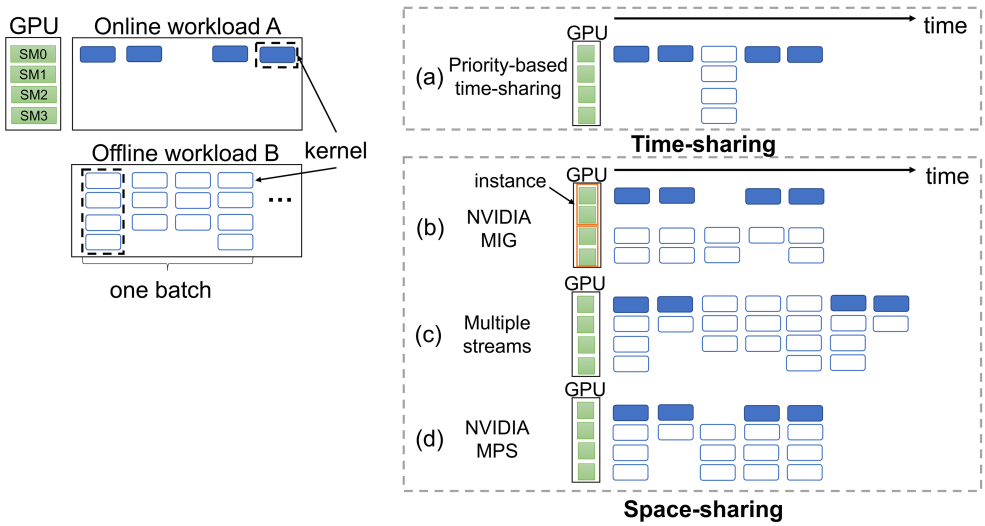

Figure 2 (Color online) Example of different GPU sharing approaches for NVIDIA GPUs containing one online workload A and one offline workload B. The GPU has four SMs. A's kernels are represented by solid squares and B's kernels are represented by hollow squares. We show the duration of one inference of A and repeat B batch by batch. (a) Priority-based time-sharing; (b) NVIDIA MIG; (c) multiple streams; (d) NVIDIA MPS.

workload usually cannot fill all SMs of one GPU completely [10, 11], leading to a waste of GPU computation units.

Opportunity: space-sharing of GPUs. When a workload cannot fully utilize the GPU computation units, i.e., SMs, it can share the idle SMs with other workloads. The SMs of one GPU can be divided into multiple parts, and used by different workloads simultaneously, i.e., space-sharing. We summarize three space-sharing approaches to share widely-deployed NVIDIA GPUs in Figure 2. NVIDIA provides multi-instance GPU (MIG) which can partition one GPU into multiple instances, as shown in Figure 2(b). However, the partition cannot be dynamically adjusted during workload execution, and thus, we have to allocate maximum resources for online workloads which leads to a waste of GPU resources. Additionally, MIG only works for specific new-generation GPU types, e.g., A100 and H100, while many production clusters have deployed GPUs that do not support MIG.

CUDA provides multiple streams (Figure 2(c)) to execute kernels from multiple workloads, but the concurrent workloads can significantly degrade the performance of online workloads. Besides, this approach needs to modify users' code and merge multiple workloads into one process, which is hard to manage in production clusters.

We find that NVIDIA MPS (Figure 2(d)) provides the best trade-off between GPU resource utilization and the performance of online workloads. MPS is supported by Kepler and newer NVIDIA GPUs which are the majority of the GPUs used in our production clusters. MPS enables NVIDIA GPUs to execute multiple workloads at the same time by assigning different sets of SMs to the shared workloads. Besides, MPS provides environment variables to roughly control the SM percentage used by each workload, which enables performance protection of online workloads.

Figure 3 shows the effect of MPS. We choose the inference process of DenseNet201 [17] as the online workload and the training process of VGG16 [3] as the offline workloads. The normalized performance is the average iteration duration when sharing divided by that when running alone. We constrain the SM percentage for the offline workload to change the performance of both workloads. Figure 3 demonstrates that one GPU can provide up to 53% more computation power while slowing online workloads less than 20%, indicating the potential of sharing GPUs in space with MPS. However, the direct application of MPS still faces challenges listed in Subsection 2.4.

2.4 Challenges for space-sharing

We summarize the challenges of applying GPU space-sharing in large-scale DL clusters below.

Performance requirements. The primary goal of production clusters is to guarantee the performance of online workloads. MPS enables us to roughly change the SM percentage used by each workload. However, it cannot guarantee the performance of online workloads. First, MPS can only constrain the

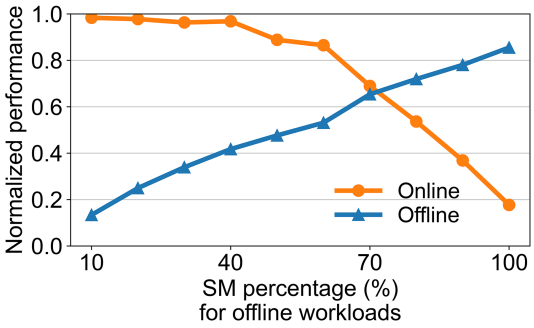

Figure 3 (Color online) Impact of the SM percentage for offline workloads (DenseNet201 as the online workload and VGG16 as the offline workload).

Table 1 Machine configurations of the heterogeneous clusters managed by MuxFlow. Misc GPUs include P4, V100, and A30

GPU type	#CPUs	Mem (GiB)	#GPUs	#Nodes
T4	96/128	251/376/503	4/8	2356
A10	128	2015	4	220
Misc	64/96/128	251/376/1007	4/8	22

allocated SM percentage which does not guarantee the performance. Second, MPS cannot timely respond to workload variations. For example, online requests may suddenly burst due to a special activity, but the SM percentage for offline workloads cannot be reduced timely, which leads to the degradation of online performance.

Reliability. For production clusters, the shared workloads, especially the online workloads, should not be influenced by other workloads. However, MPS is notorious for its serious error propagation problems. That is, when one workload encounters an error, the shared workload will also be influenced and even stopped. This problem occurs in almost every GPU in clusters directly using MPS, hindering the opportunities of applying MPS in practice. Such a problem requires carefully-designed strategies for system reliability.

Practicability. Various practical problems exacerbate the deployment of GPU sharing. First, the devices do not always perform ideally and abnormal device status may greatly hinder the workloads. For example, when the device is overheating or some resources are overcommitted, there will be a serious drop in execution speed. Such cases require extra monitoring and handling mechanisms. Second, some specific features of GPUs should also be considered. For example, the GPU clock frequency will be degraded when the GPU utilization is high, which is commonly observed in practice and can greatly impact the performance of workloads. Third, the sharing techniques should be compatible with existing infrastructure and introduce little burden to users.

Efficiency of offline workloads. Different SM percentages assigned to offline workloads can greatly influence the performance of both shared workloads. We change the SM percentage assigned to offline workloads from 10% to 100% as shown in Figure 3. The normalized performance of both online workload and offline workload varies by more than 5×. In consequence, choosing a proper SM percentage is important for efficiency.

3 MuxFlow overview

MuxFlow is a DL system that enables efficient space-sharing. MuxFlow is deployed in four clusters with more than 18000 heterogeneous GPUs at ByteDance. The machine configurations are shown in Table 1. The architecture of MuxFlow is shown in Figure 4. MuxFlow consists of a service manager for online workloads, a global manager for offline workloads, and a set of local executors on each node.

Service manager. In this paper, we only describe the functionality of the service manager used at ByteDance as the details of the service manager are beyond the scope of this paper. The service manager is responsible for online workload deployment, online request discovery, and horizontal pod autoscaling. To speed up the online serving process, it applies several widely-used techniques, e.g., batching and pipelining the preprocessing and inference process.

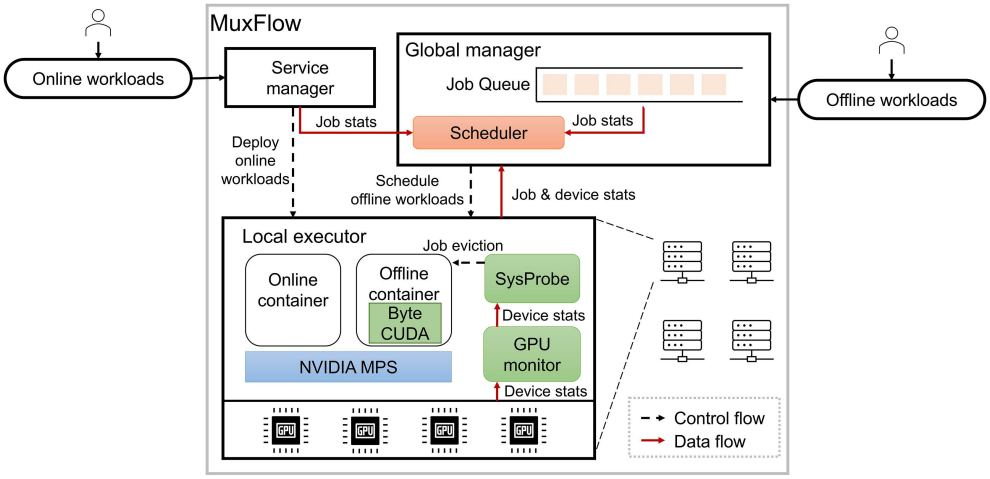

Figure 4 (Color online) MuxFlow architecture.

Global manager. When an offline workload is submitted to MuxFlow, the global manager buffers the offline workload in a pending queue and makes scheduling decisions.

Local executor. Each local executor manages the workloads on one node. The local executor executes workloads according to the scheduling decision of the scheduler and monitors the running workloads. Besides, it evicts the offline workload if the online workload is under threat. The workloads share the same GPU in space with MPS. There are four components in the local executor: online container, offline container with ByteCUDA, GPU monitor, and SysProbe.

The online container runs the online workload and serves online requests from upstream callers. The offline container runs the offline workload. With ByteCUDA built in the offline container, the execution of the offline workload is controlled to guarantee the performance of the online workload. The GPU monitor continuously collects GPU metrics, such as GPU utilization, memory usage, and SM clock. These data reflect the workload pressure of each GPU and they are leveraged by the SysProbe and ByteCUDA to manage offline workloads. The SysProbe maintains a state machine reflecting the device status and ensures that the device is not in an unhealthy status. The state machine transits according to the GPU metrics collected by the GPU monitor. When the state machine indicates one online workload is highly influenced, the SysProbe will evict the shared offline workload.

4 Efficient space-sharing

The goal of MuxFlow is to guarantee the performance of online workloads, improve the efficiency of offline workloads, and provide reliable space-sharing. In this section, we introduce how to provide efficient space-sharing in local executors. We first describe how we protect the performance of online workloads with a two-level protection mechanism. Then we introduce the dynamic SM allocation mechanism to improve the efficiency of offline workloads.

4.1 Two-level performance protection

MPS provides an environment variable to control the SM percentage used by a workload[1)]. We can roughly limit the offline workload with this environment variable and reduce the slowdown of online workloads indirectly. However, in production clusters, we need rigorous protection mechanisms for online workloads. MuxFlow employs a two-level protection mechanism as shown in Figure 5. Specifically, MuxFlow controls offline workloads to protect online workloads at the workload level by ByteCUDA and the GPU level by SysProbe.

First of all, we need GPU metrics to make decisions on how to control the offline workloads. In the local executor, the GPU monitor collects real-time GPU metrics periodically. The collection interval is in the millisecond level for timely control. The metrics include GPU resource utilization (e.g., GPU

1) CUDA_MPS_ACTIVE_THREAD_PERCENTAGE configures the active thread percentage at the client process level and limits the SM percentage used by the client process.

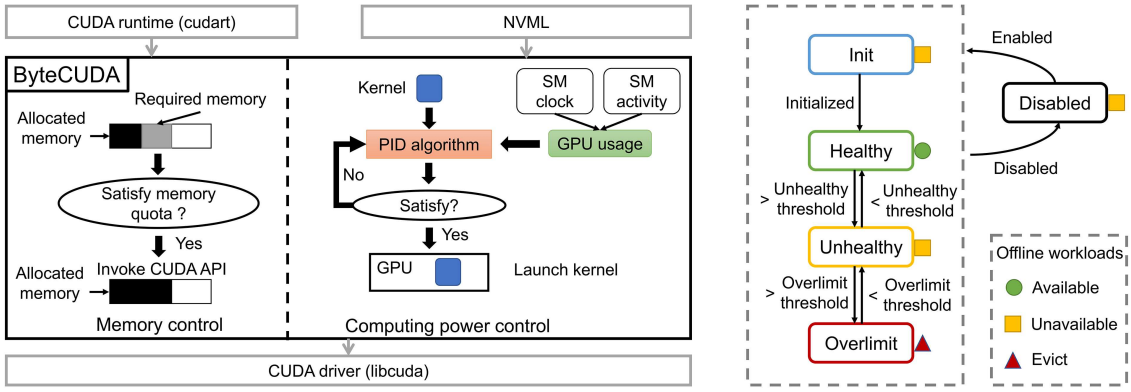

Figure 5 (Color online) Two-level performance protection for online workloads. ByteCUDA and SysProbe protect the performance of online workloads from the workload and GPU levels, respectively. (a) ByteCUDA; (b) the state machine of SysProbe.

utilization, SM activity, and GPU memory usage), and GPU device status (e.g., SM clocks, power, and temperature).

Workload level. ByteCUDA is built in the offline container to control the GPU memory and computation units used by the offline workload, as shown in Figure 5(a). For memory, ByteCUDA keeps track of the GPU memory usage and makes sure that the memory used by the offline workload does not exceed the GPU memory quota. Specifically, whenever the DL framework, e.g., TensorFlow [18] and PyTorch [15], applies for GPU memory by calling the related CUDA API, the call will be checked by ByteCUDA first.

For computation units, we aim to guarantee the performance of online workloads and improve GPU computation utilization. For NVIDIA GPUs, the SM clock represents how fast the SMs execute instructions. The performance of online workloads is greatly affected by the SM clock, and the SM clock will decrease when the GPU load is high. The decrease in the SM clock is especially noteworthy for NVIDIA GPUs for inference, e.g., T4. Thus, our goal is equivalent to attaining both a high SM clock and high GPU utilization. Formally, we define U_{SM} as the SM activity and a_{C} as a clock factor that is negatively correlated with the SM clock. We use the GPU usage U_{GPU} to quantify our goal,

$$U_{\mathrm{GPU}} = U_{\mathrm{SM}} \times a_{\mathrm{C}}. \tag{1}$$

However, the SM clock and GPU utilization are conflicting in practice because the SM clock is negatively correlated to the GPU load, while the GPU utilization is positively correlated to the GPU load. Note that when sharing online and offline workloads, it is efficient enough to get an SM clock that is close to the SM clock when the online workload runs separately. Consequently, ByteCUDA sets an SM clock threshold for these two goals. When the SM clock is below the threshold, ByteCUDA tends to improve the SM clock. When the SM clock is over the threshold, ByteCUDA tends to improve the GPU utilization. The factor a_{C} can be calculated by

$$a_{\mathrm{C}} = \begin{cases} 1 + a_{\mathrm{L}} \times (T_{\mathrm{SM}} - C_{\mathrm{SM}})/T_{\mathrm{SM}}, & C_{\mathrm{SM}} < T_{\mathrm{SM}}, \\ 1 - a_{\mathrm{H}} \times (C_{\mathrm{SM}} - T_{\mathrm{SM}})/(C_{\mathrm{H}} - T_{\mathrm{SM}}), & C_{\mathrm{SM}} \geqslant T_{\mathrm{SM}}, \end{cases} \tag{2}$$

where a_{L} is a parameter for low SM clock, a_{H} is a parameter for high SM clock, C_{SM} is the SM clock, T_{SM} is a SM clock threshold, and C_{H} is the highest SM clock. a_{L} is much larger than a_{H} to show the preference of increasing the SM clock when it is below the threshold. With the GPU usage U_{GPU}, ByteCUDA will delay the kernel when U_{GPU} is high and launch the kernel when U_{GPU} is low. Additionally, as the GPU usage U_{GPU} may change rapidly, ByteCUDA leverages the PID algorithm [19] to provide stable and robust control.

GPU level. ByteCUDA constrains offline workloads in fine grain and provides indirect performance protection for online workloads. However, it cannot react to changes caused by online workloads in time. For example, when the GPU memory usage of the online workload bursts, ByteCUDA cannot dynamically adjust the GPU memory quota for offline workloads. Additionally, ByteCUDA cannot respond to device exceptions or potential influence brought by high resource utilization. Thus, at the GPU level, MuxFlow

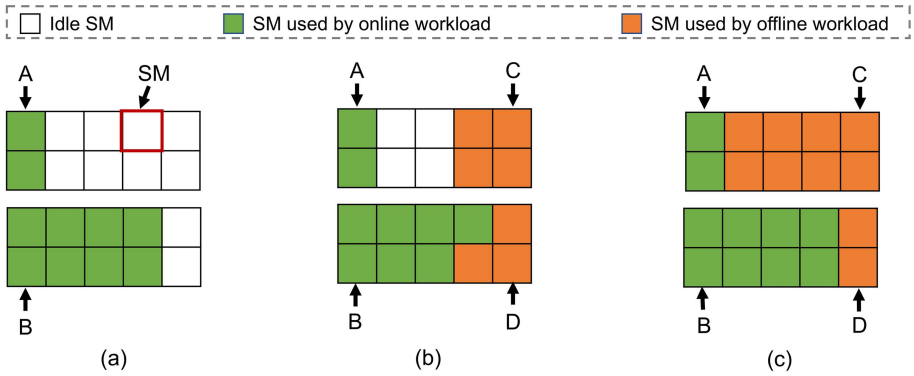

Figure 6 (Color online) A and B are two online workloads. C and D are two offline workloads. (a) Online-only has unused SMs. (b) Fixed allocation (which allocates 40% SMs to offline workloads) has unused SMs when sharing with A, and creates contentions when sharing with B. (c) Dynamic allocation fully utilizes SMs and does not create contentions.

uses SysProbe to monitor the GPU device status with a state machine. Figure 5(b) shows the state machine of SysProbe. The state machine has five states and each state has a set of metric thresholds for GPU utilization, SM activity, SM clock, GPU memory usage, device temperature, and power. The threshold values are empirically selected and tuned during deployment.

The five states of SysProbe are as follows. (1) Init state represents that the GPU is being initialized. When the initialization finishes, the Init state will transit to the Healthy state. (2) Healthy state represents that the GPU is healthy and the offline workloads can be scheduled to or executed on GPUs in this state. The metric thresholds of this state guarantee that the online workload is not influenced by the offline workloads. Once one metric reaches the Unhealthy threshold, the state will transit to the Unhealthy state. Furthermore, once one metric exceeds the Overlimit threshold, the state will directly transit to the Overlimit state. (3) Unhealthy state represents that the online workloads may be influenced and the offline workloads are forbidden to be scheduled to GPUs in this state. Once one metric exceeds the Overlimit threshold, the state will transit to the Overlimit state. Oppositely, if all metrics are below the Healthy threshold, the state will transit to the Healthy state. (4) Overlimit state represents that the GPU device is overloaded, and the offline workloads are evicted. When all metrics are below the Overlimit threshold after a period, the state will transit to the Unhealthy state. To avoid frequent eviction, the period is set to be the exponent of the times going into the Overlimit state during the last two hours. (5) Disabled state represents that the GPU device is unavailable and no workload can run on it.

4.2 Dynamic SM allocation

In Figure 3, we illustrate that the SM percentage assigned to offline workloads can influence the speed of shared workloads dramatically. In other words, we can balance the speed of online workloads and that of offline workloads by changing the SM allocation. Our goal is to maximize the efficiency of offline workloads with an acceptable slowdown of the online workloads. As SMs are the computation units of GPUs, maximizing the efficiency of offline workloads is approximately equal to maximizing the percentage of SMs assigned to offline workloads. Apparently, fixed SM allocation is not efficient for all sharing cases. We use the example in Figure 6 to illustrate the drawbacks of fixed SM allocation. A and B are online workloads, and C and D are offline workloads. Assume that the SM percentage is set to 40% for offline workloads. The online workload A only uses 20% SMs and leaves more than 40% SMs. If we fix the SM percentage for offline workloads to 40%, there will be 40% idle SMs. The online workload B uses 80% SMs when running alone and the left SMs are less than 40%. With the fixed SM allocation, the offline workload D will occupy B's SMs and slow the online workload B down.

To provide efficient space-sharing, we propose a dynamic SM allocation mechanism by selecting the proper SM percentage for offline workloads. It assigns the SM percentage according to the SM activity of online workloads, as shown in Figure 6(c). In practice, we continuously monitor the SM activity of online workloads. Assume the SM activity of one online workload in the last execution interval is x%, then the SM percentage is set to $(100 - x)$% for the shared offline workload in the next execution interval. The execution interval is several minutes considering the restart overhead and the timeliness of adjustment. For example, we can set the SM percentage for the offline workload C to 80% and D to 20%. In this way, the computation units (SMs) are fully used and the shared workloads do not contend for SMs.

Table 2 Propagated errors of MPS in a production cluster

Error type	Description	Proportion (%)	Countermeasure
MPS hangs caused by SIGINT/SIGTERM	Once some offline workloads are terminated by SIGINT/SIGTERM, there is a high probability to cause MPS hangs and the shared online workloads cannot continue to serve requests.	99	Graceful exit mechanism
MPS server crashes	The MPS server may crash due to program bugs.	0.28	Error pattern analysis and restart
XID31	XID31 event represents GPU memory page fault, which is reported from the NVIDIA driver.	0.15	
MPS hangs caused by other reasons	MPS may hang due to other reasons, like CUDA sticky errors and SIGKILL.	0.08	
Other errors	Other errors include XID event, ECC error, and high utilization fault.	0.49	

5 System reliability

Sharing GPU in large-scale DL clusters usually encounters reliability problems. First, shared workloads may interfere with each other even with the isolation ability of containers. Specifically, MPS has a serious error propagation problem; i.e., the error of one workload may impact other workloads that share the same GPU. The error propagation problem is dangerous in large-scale clusters, especially for online workloads. Second, the number of workloads (or the number of containers) is doubled, which indicates a higher failure rate for the clusters. We propose a mixed error-handling mechanism to handle the reliability problem.

The error propagation problem of MPS can be triggered by various reasons. For example, if one offline workload is terminated by the SIGINT signal, the MPS context may hang and the shared online workload cannot serve requests. We summarize propagated errors in our production cluster with MPS enabled as shown in Table 2. We find that 99% of such propagated errors are caused by SIGINT/SIGTERM. Some offline workloads, such as PySpark jobs, are terminated by SIGINT/SIGTERM. If some kernel is executed while receiving the signal, the MPS will hang due to hardware-level fatal exceptions and the workload cannot continue to serve requests. To handle this dominant error type, one possible approach is to modify the application-level mechanism of signal processing. For example, the PySpark job can let workers synchronize when receiving the signal to avoid fatal exceptions. However, the application-level mechanism is not general for various job types. At ByteDance, we use ByteCUDA to intercept SIGINT and SIGTERM signals and perform graceful exits. Specifically, we initialize ByteCUDA for each GPU process/thread. When ByteCUDA receives SIGINT/SIGTERM signals, it will freeze all kernel launches and release the CUDA context actively. All GPU processes/threads share the same pipe file to monitor the freeze signal from each other.

Other errors only account for 1%, e.g., MPS server crash (caused by program bugs), XID31 event (GPU memory page fault), and MPS hangs caused by other issues (e.g., CUDA sticky errors and SIGKILL). For these errors, we identify them according to the error message or our summarized error patterns. Some errors, like XID31 and MPS server exceptions, can be identified from the error message in the log file. Other errors require monitoring device metrics. For example, when the GPU utilization is always 100% but the workload does not proceed, MuxFlow will report a high utilization fault. An automated detector monitors GPUs and alerts when the error patterns are satisfied. Once ByteCUDA gets the alert, it will solve these errors according to predefined rules. These rules can be classified into four categories, maintaining devices (e.g., reboot the device, reinstall the system, drain the device, or change system configurations), restarting MPS or workload, constantly monitoring, and reporting to human experts.

6 Implementation

MuxFlow is deployed at the production clusters of ByteDance to serve daily DL workloads. MuxFlow is integrated with Kubernetes to manage thousands of GPUs in each cluster and more than 18000 GPUs in total. MuxFlow is implemented with about 48000 lines of code in C++, Python, and Go.

Service manager. For online workloads, we use the existing service manager at ByteDance which deploys containers, serves requests, and autoscales pods. The entire pipeline of serving online requests

includes receiving requests, preprocessing, model inference, and sending results.

Global manager. The number of possible sharing plans is factorial to the number of workloads, which is huge for a production cluster. Therefore, the efficiency of the scheduling strategy is more important than the optimality. We choose first-come-first-serve (FCFS) as the scheduling strategy for efficiency. We modify the Kubernetes scheduler to schedule offline workloads. Note that the scheduling strategy is orthogonal to our design and can be further explored. However, as the characteristics of online workloads may change over time, we need to periodically execute the dynamic SM allocation mechanism. The execution interval is set to 15 min to minimize the overhead of terminating, resetting, and resuming workloads.

Local executor. Each local executor executes online workloads according to the service manager and offline workloads according to the global manager. DL workloads are executed in Docker containers with our custom components. We add Best-Effort GPU DevicePlugin in Kubernetes and relevant control paths with Kubelet and SysProbe for offline workloads. To control the SM percentage of offline workloads, we change the environment variable CUDA_MPS_ACTIVE_THREAD_PERCENTAGE provided by MPS. When changing the SM percentage for offline workloads, we need to restart related offline workloads to reallocate the SM percentage, but perform no action for online workloads. The GPU monitor collects resource metrics through DCGM and NVML for NVIDIA GPUs. The SysProbe will ask the NodeManager in Kubernetes to evict offline workloads when the state is Unhealthy. ByteCUDA intercepts nearly 800 CUDA driver APIs for GPU memory allocation and kernel launch. The GPU memory quota of offline workloads is fixed to 40% as Figure 1 reports that most online workloads use less than 60% GPU memory. The parameters to calculate GPU usage in (1) and (2) are empirically selected through trial-and-error. The principle of how to select these parameters will be discussed in Subsection 8.

7 Evaluation

Our evaluation consists of testbed experiments, trace-driven simulations, and results from the production deployment. We mainly focus on the efficiency of MuxFlow. For reliability, the error rate of MuxFlow is similar to the dedicated inference clusters in production deployment at ByteDance.

7.1 Experimental setup

Testbed. We conduct the testbed experiments with 125 machines and 1000 GPUs. Each machine is equipped with 8 NVIDIA Tesla T4 GPUs, 2 Intel Xeon Platinum CPUs, 251 G memory, and 100 + 25 G NIC. We use PyTorch v1.8.0 with CUDA 11.1 for offline workloads. We use real online workloads in our production clusters and these workloads use diverse DL frameworks and CUDA drivers.

Simulator. Similar to recent studies [20, 21], we build a simulator to evaluate a broader set of configurations, traces, and baselines. We profile the iteration duration, GPU resource utilization, and device metrics of the selected DL workloads when they are executed separately and shared with others. The profiling results include more than 200 executions. The difference between the simulation and testbed experiment is under 5%, showing the high fidelity of the simulator.

Workloads. We use the actual online workloads and real-time requests at ByteDance for the testbed experiment. The online workloads use a wide spectrum of DL models, including CNNs, GNNs, LLMs, and recommendation models. For trace-driven simulations, we select three online workloads deployed over hundreds of GPUs at ByteDance, and generate requests according to their actual query per second (QPS) varying from 20 to 190. For offline workloads, we leverage the public trace from Microsoft [22] and split the trace according to the virtual cluster ID. We choose DL models from four popular DL models including ResNet50 [4], VGG16 [3], DensNet201 [17], and Inception-V3 [23], in accordance with the common practice [11, 20, 21]. The number of offline workloads in these traces varies from 1410 to 7287.

Baselines. We compare MuxFlow with four related systems: (1) Online-only executes only the online workloads and shows the optimal latency for online workloads. (2) Time-sharing shares workloads in time and assigns the time slices of GPUs to the shared workloads by the GPU driver strategy, which is adopted by Gandiva [24]. (3) Priority-based time-sharing (PB-time-sharing) sets online workloads with high priority and assigns more time slices of GPUs to high-priority workloads to protect their performance, which is adopted by AntMan [13] and PAI [25]. (4) Priority-based multi-stream (PB-multi-

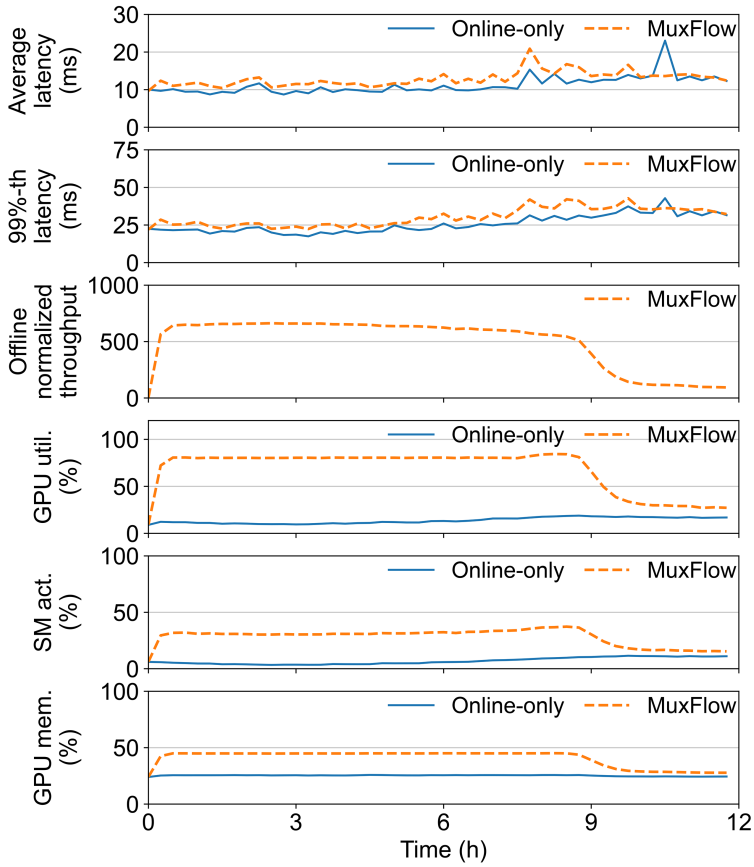

Figure 7 (Color online) Detailed metrics in the testbed experiment.

stream) utilizes multi-stream with priority to share workloads in space and guarantee the performance of online workloads, which is adopted by DeepPool [26] and Reef [11].

Metrics. Average latency and 99%-th latency are two common metrics to evaluate the performance of online workloads [8,11]. Average job completion time (JCT) is used to reflect the workload efficiency [13, 21]. Offline normalized throughput shows the sharing efficiency. We define how much GPU the offline workloads get in the aspect of computation speed as the oversold GPU. This metric is in the range of $[0, 1]$, where 0 represents that the offline workloads get no GPU computation resource, and 1 represents that the offline workloads get equivalent GPU computation resources as they are executed exclusively. This metric can be calculated by

$$\text{Oversold GPU} = \frac{\sum_{w \in W_{\text{off}}} T_w^{\text{sep}}}{\sum_{w \in W_{\text{off}}} T_w^{\text{real}}}, \tag{3}$$

where W_{off} represents offline workloads, T_w^{real} represents the real execution time of w, and T_w^{sep} represents the execution time of w when running exclusively. We report GPU resource utilization with three metrics: GPU utilization, SM activity, and GPU memory usage.

7.2 Testbed experiments

We first evaluate MuxFlow on a testbed with 1000 GPUs. Figure 7 shows the detailed metrics for online workloads, offline workloads, and GPU resource utilization. To get the metrics of Online-only, we stop the offline workloads for one minute in every execution interval. The execution interval is set to 15 min. Most offline workloads finish before 8 h. Thus, there is an obvious shift in the offline normalized throughput and GPU resource utilizations between 7 to 8 h.

Performance of online workloads. Figure 7 shows that MuxFlow increases the average latency of online workloads by 16.4% and the 99%-th latency by 18.4%. These results indicate that MuxFlow slows down online workloads by less than 20%, i.e., 10 ms. It is worth mentioning that such a slowdown is

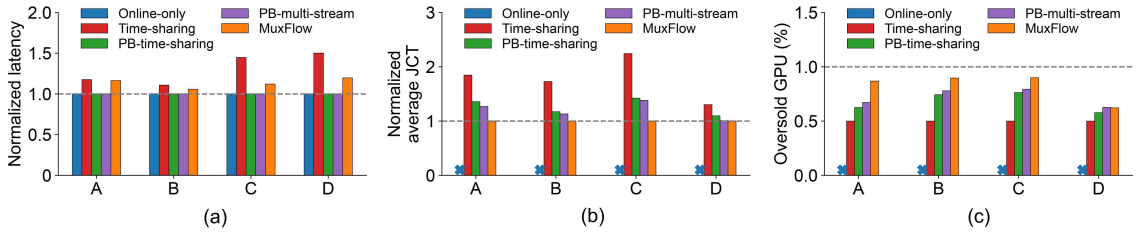

Figure 8 (Color online) Compare MuxFlow with related work. A, B, C, and D are different traces. (a) Latency of online workloads; (b) JCT of offline workloads; (c) oversold GPU.

almost imperceptible for most online workloads, e.g., recommendation services and machine translation. In the wide spectrum of production online workloads deployed in ByteDance, the latency requirements of most online workloads are more than 100 ms, hence the 10 ms slowdown of online workloads is acceptable in practice. Note that the slowdown is further diluted when considering the entire pipeline of serving online workloads. We observe that 1.5% executions of offline workloads are evicted, demonstrating the functionality of performance protection mechanisms.

Efficiency of offline workloads. We find that MuxFlow provides up to 71.2% GPU resources to offline workloads, which is a substantial saving considering the large quantities of GPUs in large-scale production clusters.

GPU resource utilization. Figure 7 compares the GPU computation utilization and memory usage between Online-only and MuxFlow. The utilization numbers are the average of all GPUs. MuxFlow improves the GPU utilization by 4.8×, SM activity by 4.0×, and GPU memory usage by 1.6×.

Reliability. We observe that during the 12-hour testbed experiments, no error propagation problem happens. That is, no online workload is influenced by offline workload errors, demonstrating the reliability of MuxFlow.

7.3 Comparison with related work

We compare MuxFlow with four related solutions, Online-only, time-sharing (used by Gandiva [24]), PB-time-sharing (used by AntMan [13] and PAI [25]), and PB-multi-stream (used by DeepPool [26] and Reef [11]). The related solutions are implemented in our simulator, and evaluated with production online workloads and representative offline workloads. Figure 8 reports the average latency of online workloads, average JCT of offline workloads, and oversold GPU to offline workloads. We normalize the latency by that of Online-only and other metrics by those of MuxFlow. MuxFlow improves the average JCT by up to 2.2×, and the oversold GPU by up to 2.0×, while slowing down the online workloads by less than 20%. Time-sharing slows down online workloads by up to 50%, indicating a great impact on online workloads. PB-time-sharing utilizes priority to protect the performance of online workloads, but its metrics for offline workloads are worse than MuxFlow because MuxFlow can utilize the wasted GPU resources in space. PB-multi-stream has worse offline metrics than MuxFlow, showing the efficacy of MuxFlow. Additionally, PB-multi-stream needs extra efforts of users to modify their code (e.g., about 10000 s lines JSON and CUDA code for ResNet in REEF [11]) or system managers to manually merge multiple workloads, which is impractical for large-scale clusters.

7.4 Analysis of MuxFlow

Impact of the two-level performance protection. To understand the impact of the two-level performance protection mechanism, we evaluate the original MPS with our proposed mechanism as shown in Figure 9. Here we focus on ByteCUDA because SysProbe mainly takes effect in large-scale clusters and extreme cases. We choose DenseNet201 [17] and Inception V3 [23] as online workloads, and ResNet50 [4] and VGG16 [3] as offline workloads to illustrate the benefits of our design. We normalize the latency when one workload is shared with another by the latency when the workload is executed separately. When we only use MPS, the performance of online workloads is degraded by more than 20% in 3/4 cases, and the degradation can be up to 3×. The slowdown of online workloads, whose priority is higher than offline workloads, is a disaster in production deployment. ByteCUDA can guarantee a less-than-20% degradation of online performance by sacrificing the performance of low-priority offline workloads, illustrating the benefits of MuxFlow.

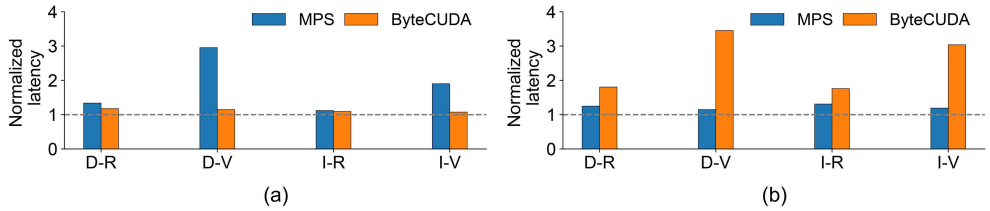

Figure 9 (Color online) Impact of the two-level performance protection mechanism. The latency is normalized by the latency when the workload is executed separately. A-B represents sharing one online workload A with one offline workload B. D, I, V, and R are short for DenseNet201, Inception V3, VGG16, and ResNet50, respectively. (a) Online workloads; (b) offline workloads.

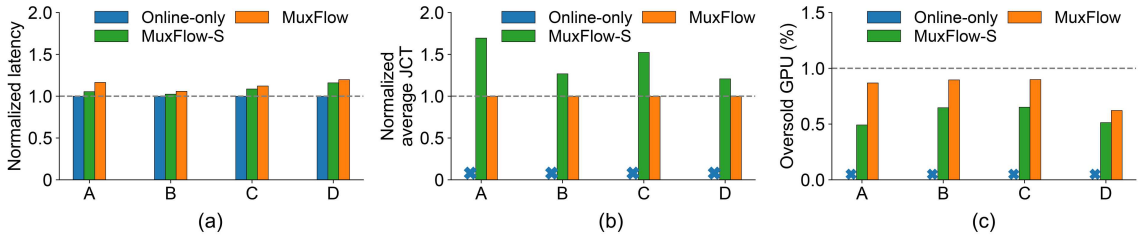

Figure 10 (Color online) Impact of the dynamic SM allocation mechanism. (a) Latency of online workloads; (b) JCT of offline workloads; (c) oversold GPU.

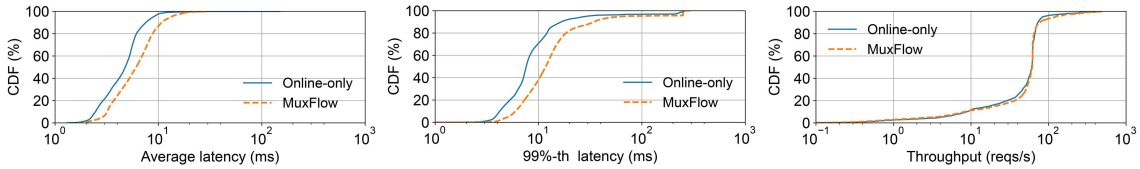

Figure 11 (Color online) Performance of online workloads in deployment.

Impact of the dynamic SM allocation. We study the impact of the dynamic SM allocation mechanism, as shown in Figure 10. We leverage the simulator to compare MuxFlow with MuxFlow without the dynamic SM allocation mechanism (MuxFlow-S). Compared with MuxFlow-S, MuxFlow improves the average JCT by up to 1.7× and oversold GPU by up to 1.8×, while increasing online latency by less than 11%. These improvements confirm that only the online protection mechanisms are not efficient, and the dynamic SM allocation mechanism is important for offline efficiency.

7.5 Production deployment

We have deployed MuxFlow on the production clusters with more than 18000 GPUs at ByteDance. To verify the performance protection for online workloads provided by MuxFlow, we collect the latency and throughput of online workloads that are deployed with both MuxFlow and dedicated inference clusters (Online-only), as shown in Figure 11. The average latency increases by less than 10 ms and 99%-th latency increases by less than 30 ms. The slowdown of online workloads is acceptable compared with the latency requirements. We also show the number of GPUs used by offline workloads in Figure 12, indicating that we can save 10000 s GPUs every day. Besides, we collect the average resource utilization of MuxFlow and Online-only for four weeks. We observe that MuxFlow improves the GPU utilization from 26% to 76%, SM activity from 16% to 33%, and GPU memory usage from 42% to 48%, indicating the efficiency of MuxFlow. The GPU memory usage increases less than other utilizations because of our conservative memory limitation for offline workloads.

The percentage of daily error devices of MuxFlow is 0.9%, which is close to the error rate of Online-only, 0.7%. Note that compared to Online-only, MuxFlow runs two containers on each GPU device—one container for an online workload and the other for an offline workload. Given that MuxFlow runs 2× as many containers as Online-only, the increase in the error rate (0.2%) is slight. The extra device errors of MuxFlow come from MPS server crashes and MPS hangs.

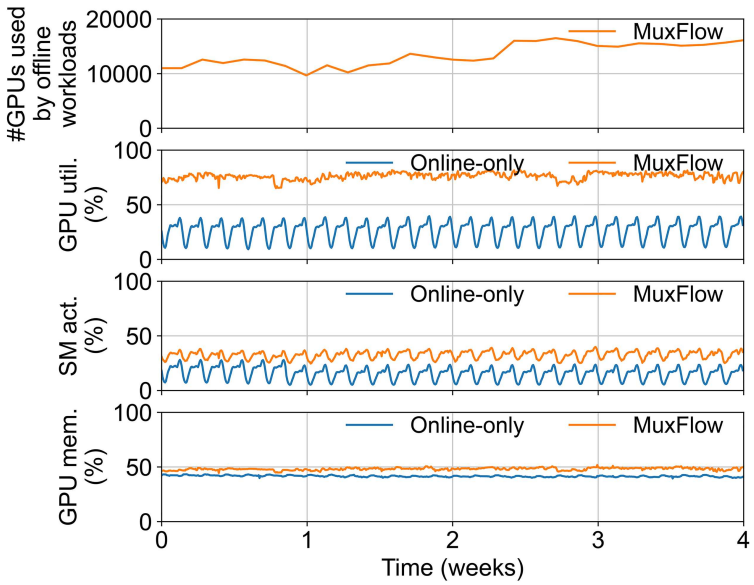

Figure 12 (Color online) GPU utilization in deployment.

8 Lessons and future directions

Reliable space-sharing. How to provide reliable space-sharing is one of the most important problems of deploying GPU sharing. To our best knowledge, we are the first to thoroughly analyze MPS-related unreliable cases in production clusters and propose corresponding solutions. Besides, we have worked with NVIDIA to improve MPS. For example, we observed that sharing workloads compiled by different GCC versions can cause MPS server hangs. This problem has been verified and fixed in NVIDIA GPU Driver 470.

However, things become complicated when considering malicious behaviors, e.g., intriguing sticky CUDA error by dividing zero on purpose. To avoid malicious behaviors, MuxFlow only accepts trustworthy offline workloads from internal users, and we employ a fault detector with manually designed rules to monitor and alert abnormal situations. Yet if considering external users in a cloud setting, we need more general approaches for system reliability. One opportunity is to leverage DL to discover malicious behaviors automatically [27]. Besides, enabling the scheduler to identify fail-prone workloads and avoid sharing them is another possible approach.

Slowdown of online workloads. In this paper, we limit the slowdown of online workloads to less than 20%. The slowdown is affordable and acceptable in our deployment scenario, because most latency requirements are over 20% longer than the latency of online workloads. For online workloads with more strict latency requirements, we reserve exclusive GPUs for them. Besides, we focus on the percentage of slowdown because we find that the percentage is more important than the absolute time in large-scale clusters with diverse latency requirements. Note that the slowdown threshold is a tradeoff between the online service quality and resource utilization, and it can be changed in MuxFlow by two mechanisms. First, the parameters of GPU usage (Eqs. (1) and (2)) in ByteCUDA affect the execution of offline and online workloads. Second, we can adjust the SM percentage assigned to offline workloads to change the slowdown of online workloads. During our deployment, we find that the latency is time-varying based on query frequency, system upgrade, and service evolution. Thus, we plan to change the slowdown threshold and latency requirements automatically.

Overhead of the offline workloads. ByteCUDA introduces some conditional statements in the calls of CUDA APIs for the offline workloads and the slowdown of offline workloads is less than 5% as we measured. ByteCUDA introduces no overhead for online workloads. Besides, we execute the dynamic SM allocation mechanism periodically which may terminate offline workloads. We select the proper execution interval for the dynamic SM allocation mechanism, i.e., 15 min, to balance between the adjustment efficiency and the preemption overhead. On the other hand, the oversold GPU shown in Figures 8 and 10 measures the real achieved throughput. The oversold GPU is more than 70%, demonstrating that the benefit of sharing outweighs the overhead.

Strategy of ByteCUDA. ByteCUDA is responsible for controlling the GPU memory and computing power of offline workloads. The GPU memory usage of offline workloads is limited to 40% according to the statistic in Figure 1. For computing power, its influence is reflected by the latency of online workloads, which is not as real-time as other metrics like GPU power, SM clock, and GPU utilization. Therefore, we need to adjust the computing power according to the aforementioned real-time metrics. Considering the conflict between the GPU utilization and the SM clock, we cannot use only one index. During the deployment, we explored multiple adjusting strategies. These strategies mainly fall into two categories, i.e., utilizing GPU power and utilizing SM clock. We observe that the strategies using GPU power are good at protecting online performance. However, these strategies constrain offline workloads even when the computing units are underutilized, wasting computing units. Thus, ByteCUDA adopts the strategy using SM clock and SM activity (Eq. (1)).

Parameter settings. MuxFlow includes a few parameters. In (2), a_{L} should be larger than a_{H} to show the preference of increasing the SM clock for faster execution. T_{SM} is set according to the SM clock of GPUs that are dedicated to running online workloads. The thresholds in SysProbe are chosen based on the historical failure data in our clusters. The unhealthy thresholds should cover more failures than the overlimit thresholds. We adjust these thresholds periodically.

CPU and memory sharing. Except for GPUs, CPUs and memory are also shared between online and offline workloads. MuxFlow uses SysProbe to obtain resource utilization of online workloads, calculate remaining resources for offline workloads, and create virtual resources for offline workloads. Similar to GPUs, once the utilization of CPUs and memory is higher than the thresholds, the offline workloads will be evicted to avoid CPU conflict and reduce disk I/O. Note that we also monitor other related indices that may degrade online performance, e.g., the count of pages reclaimed by the kernel swap daemon. Different from GPUs, CPUs and memory are shared among workloads even without offline workloads. We bind the CPU cores to online and offline containers separately according to their CPU load. During our deployment, we fixed a Linux kernel bug that stops the CPU occupation of processes prematurely and leads to higher latency.

The number of shared workloads. In MuxFlow, we share at most one offline workload with an online workload on a GPU because one offline workload is usually enough to fill up the SMs of a GPU. Sharing multiple offline workloads with multiple online workloads may bring more benefits, especially for lightweight offline workloads. However, there are three challenges in sharing multiple workloads. First, we need to guarantee the performance of all online workloads. Second, we need to limit the total SM percentage used by multiple offline workloads which cannot be simply limited by MPS parameters. Third, how to schedule and group workloads becomes an NP-hard problem [21].

Large language model (LLM). LLM is the trend of DL workloads. In our experiments, we include some widely-deployed language models for online workloads, like GPT and Bert inference. Our system and mechanisms are general and can be applied to LLM workloads. However, to achieve efficient GPU-sharing for LLMs, specific optimizations need to be explored considering the characteristics of LLMs, e.g., the large requirements of memory for key-value cache and the long inference process.

9 Related work

DL workload management. Existing DL management systems are designed for online or offline workloads, but not both. The primary goals of online workload management [7–9, 28] are meeting the latency requirements and improving the overall throughput. However, these systems let one workload occupy GPUs and thus, cannot fully use GPUs. Most prior management systems for offline workloads [20, 29–33] also allocate GPUs exclusively. Existing systems for offline workloads cannot be directly applied to GPU-sharing clusters because they cannot ensure the performance of online workloads. Differently, MuxFlow leverages space-sharing to improve GPU resource utilization and applies a two-level protection mechanism to guarantee the performance of online workloads.

Resource sharing for big data workloads. Prior studies [34–39] have studied resource sharing for big data workloads and CPU clusters. Many large enterprises also deploy resource-sharing clusters [40–42]. In comparison, DL workloads use GPUs to speed up DL workloads, and GPUs lack efficient and reliable sharing mechanisms. Thus, it is more challenging to deploy GPU sharing in production clusters.

GPU sharing for DL workloads. Recently, GPU sharing has been studied for DL workloads. The techniques to share GPUs mainly fall into two categories. Time-sharing approaches [21, 24, 43, 44] may

impact the performance of online workloads. Some approaches [13, 25, 45] assign time slices according to priorities to guarantee the performance of online workloads. However, these approaches still cannot improve resource utilization for each time slice.

Space-sharing can improve GPU utilization in space. MPS is a general method to multiplex jobs on NVIDIA GPUs [46–48]. However, these approaches may impact the performance of online workloads or system reliability. Laius [49] is a GPU-sharing runtime for only one GPU and does not consider system reliability, while MuxFlow is a cluster-wide GPU-sharing system deployed on more than 18000 GPUs and we analyze and handle the error propagation problem of MPS for system reliability. Some work leverages model similarity [50] or priority-based multi-stream [11, 26]. However, these approaches require manually merging multiple workloads into one process, modifying users' code, or changing CUDA and DL frameworks, which is not friendly to existing infrastructure and users. In contrast, MuxFlow is a practical space-sharing system that can scale over 18000 GPUs and has been deployed in production environment at ByteDance.

10 Conclusion

In this paper, we presented MuxFlow, the first DL system that can scale over massive GPUs to achieve efficient space sharing in production DL clusters. MuxFlow uses a two-level performance protection mechanism to ensure the performance of online workloads from both the workload level and GPU level. MuxFlow exploits dynamical SM allocation to improve the efficiency of offline workloads. To improve system reliability, MuxFlow uses a mixed error-handling mechanism based on the thorough analysis of propagated errors in production clusters. The evaluation results demonstrate the efficiency of MuxFlow. MuxFlow has been deployed in DL clusters at ByteDance with more than 18000 heterogeneous GPUs.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 62325201, 62172008), National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), and the PKU-ByteDance Joint-Lab Program.

References

1 Covington P, Jay A, Emre S. Deep neural networks for YouTube recommendations. In: Proceedings of the 10th ACM Conference on Recommender Systems, 2016. 191–198
2 Gao W H, Fan X J, Wang C, et al. Learning an end-to-end structure for retrieval in large-scale recommendations. In: Proceedings of the 30th ACM International Conference on Information and Knowledge Management, 2021. 524–533
3 Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. In: Proceedings of the 3rd International Conference on Learning Representations, 2015
4 He K M, Zhang X Y, Ren S Q, et al. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016. 770–778
5 Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017. 6000–6010
6 Gehring J, Auli M, Grangier D, et al. Convolutional sequence to sequence learning. In: Proceedings of the International Conference on Machine Learning, 2017. 1243–1252
7 Crankshaw D, Wang X, Zhou G, et al. Clipper: a low-latency online prediction serving system. In: Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation, 2017. 613–627
8 Gujarati A, Karimi R, Alzayat S, et al. Serving DNNs like clockwork: performance predictability from the bottom up. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 2020. 443–462
9 Shen H C, Chen L Q, Jin Y C, et al. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019. 322–337
10 Ma L X, Xie Z Q, Yang Z, et al. Rammer: enabling holistic deep learning compiler optimizations with rtasks. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 2020. 881–897
11 Han M C, Zhang H Z, Chen R, et al. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In: Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation, 2022. 539–558
12 Xiang Y C, Kim H. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In: Proceedings of the IEEE Real-Time Systems Symposium, 2019. 392–405
13 Xiao W C, Ren S R, Li Y, et al. AntMan: dynamic scaling on GPU clusters for deep learning. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 2020. 533–548
14 Wu B Y, Zhang Z L, Bai Z H, et al. Transparent GPU sharing in container clouds for deep learning workloads. In: Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation, 2023. 69–75
15 Paszke A, Gross S, Massa F, et al. PyTorch: an imperative style, high-performance deep learning library. In: Proceedings of the Advances in Neural Information Processing Systems, 2019
16 Dai H L, Peng X, Shi X H, et al. Reveal training performance mystery between TensorFlow and PyTorch in the single GPU environment. Sci China Inf Sci, 2022, 65: 112103
17 Huang G, Liu Z, van Der Maaten L, et al. Densely connected convolutional networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017. 4700–4708
18 Abadi M, Barham P, Chen J M, et al. TensorFlow: a system for large-scale machine learning. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, 2016. 265–283
19 Johnson M A, Moradi M H. PID Control. London: Springer-Verlag London Ltd., 2005

20 Gu J C, Chowdhury M, Shin K G, et al. Tiresias: a GPU cluster manager for distributed deep learning. In: Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, 2019. 485–500
21 Zhao Y H, Liu Y Q, Peng Y H, et al. Multi-resource interleaving for deep learning training. In: Proceedings of the ACM SIGCOMM 2022 Conference, 2022. 428–440
22 Jeon M, Venkataraman S, Phanishayee A, et al. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In: Proceedings of the USENIX Annual Technical Conference, 2019. 947–960
23 Szegedy C, Vanhoucke V, Ioffe S, et al. Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016. 2818–2826
24 Xiao W C, Bhardwaj R, Ramjee R, et al. Gandiva: introspective cluster scheduling for deep learning. In: Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, 2018. 595–610
25 Weng Q Z, Xiao W C, Yu Y H, et al. MLaaS in the wild: workload analysis and scheduling in large-scale heterogeneous GPU clusters. In: Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation, 2022. 945–960
26 Park S J, Fried J, Kim S, et al. Efficient strong scaling through burst parallel training. In: Proceedings of the 5th Conference on Machine Learning and Systems, 2022. 748–761
27 Saufi S R, Ahmad Z A B, Leong M S, et al. Challenges and opportunities of deep learning models for machinery fault detection and diagnosis: a review. IEEE Access, 2019, 7: 122644–122662
28 Romero F, Li Q, Yadwadkar N J, et al. INFaaS: automated model-less inference serving. In: Proceedings of the USENIX Annual Technical Conference, 2021. 397–411
29 Hwang C, Kim T, Kim S, et al. Elastic resource sharing for distributed deep learning. In: Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation, 2021. 721–739
30 Qiao A, Choe S K, Subramanya S J, et al. Pollux: co-adaptive cluster scheduling for goodput-optimized deep learning. In: Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation, 2021. 1–18
31 Miao X P, Nie X N, Zhang H L, et al. Hetu: a highly efficient automatic parallel distributed deep learning system. Sci China Inf Sci, 2023, 66: 117101
32 Li W X, Lin N, Zhang M Z, et al. VNet: a versatile network to train real-time semantic segmentation models on a single GPU. Sci China Inf Sci, 2022, 65: 139105
33 Hu S-M, Liang D, Yang G-Y, et al. Jittor: a novel deep learning framework with meta-operators and unified graph execution. Sci China Inf Sci, 2020, 63: 222103
34 Ghodsi A, Zaharia M, Hindman B, et al. Dominant resource fairness: fair allocation of multiple resource types. In: Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, 2011. 323–336
35 Grandl R, Ananthanarayanan G, Kandula S, et al. Multi-resource packing for cluster schedulers. SIGCOMM Comput Commun Rev, 2014, 44: 455–466
36 Grandl R, Kandula S, Rao S, et al. Graphene: packing and dependency-aware scheduling for data-parallel clusters. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, 2016. 81–87
37 Grandl R, Chowdhury M, Akella A, et al. Altruistic scheduling in multi-resource clusters. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, 2016. 65–70
38 Ousterhout K, Canel C, Ratnasamy S, et al. Monotasks: architecting for performance clarity in data analytics frameworks. In: Proceedings of the 26th Symposium on Operating Systems Principles, 2017. 184–100
39 Li X P, Pan D Y, Wang Y D, et al. Scheduling multi-tenant cloud workflow tasks with resource reliability. Sci China Inf Sci, 2022, 65: 192106
40 Boutin E, Ekanayake J, Lin W, et al. Apollo: scalable and coordinated scheduling for cloud-scale computing. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, 2014. 285–300
41 Verma A, Pedrosa L, Korupolu M, et al. Large-scale cluster management at Google with Borg. In: Proceedings of the 10th European Conference on Computer Systems, 2015. 1–7
42 Tirmazi M, Barker A, Deng N, et al. Borg: the next generation. In: Proceedings of the 15th European Conference on Computer Systems, 2020. 1–4
43 Yu P F, Chowdhury M. Salus: fine-grained GPU sharing primitives for deep learning applications. In: Proceedings of Machine Learning and Systems, 2020. 98–111
44 Bai Z H, Zhang Z, Zhu Y B, et al. PipeSwitch: fast pipelined context switching for deep learning applications. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 2020. 499–514
45 Gu J, Song S B, Li Y, et al. GaiaGPU: sharing GPUs in container clouds. In: Proceedings of the IEEE International Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications, 2018. 469–476
46 Narayanan D, Santhanam K, Kazhamiaka F, et al. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 2020. 481–498
47 Dhakal A, Kulkarni S G, Ramakrishnan K K. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In: Proceedings of the 11th ACM Symposium on Cloud Computing, 2020. 492–506
48 Choi S, Lee S, Kim Y, et al. Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing. In: Proceedings of the USENIX Annual Technical Conference, 2022. 199–216
49 Zhang W, Cui W H, Fu K H, et al. Laius: towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In: Proceedings of the ACM International Conference on Supercomputing, 2019. 58–68
50 Zhang Q L, Han Z H, Yang F, et al. Retiarii: a deep learning exploratory-training framework. In: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 2020. 919–936