Multiplexing Dynamic Deep Learning Workloads with SLO-awareness in GPU Clusters

Wenyan Chen
University of Macau
Shenzhen Institute of Advanced Technology, CAS
Macau SAR, China
yc17498@um.edu.mo

Chengzhi Lu
Shenzhen Institute of Advanced Technology, CAS
Univ. of CAS, Univ. of Macau
Macau SAR, China
cz.lu@siat.ac.cn

Huanle Xu*
University of Macau
Macau SAR, China
huanlexu@um.edu.mo

Kejiang Ye
Shenzhen Institute of Advanced Technology, CAS
Shenzhen, China
kj.ye@siat.ac.cn

Chengzhong Xu
University of Macau
Macau SAR, China
czxu@um.edu.mo

Abstract

Deep learning (DL) inference services are widely recognized as crucial workloads in large-scale cloud clusters. However, due to the stringent latency requirements, cloud providers often over-provision GPU resources, resulting in underutilization of the available GPU potential. Although co-locating tasks on the same device can enhance utilization, ensuring Service Level Objectives (SLOs) guarantees for multiplexing highly dynamic inference services becomes extremely challenging due to significant resource interference.

In this paper, we introduce Mudi, a new SLO-aware system designed to optimize the utilization of GPU resources within large-scale clusters. Mudi achieves this by efficiently multiplexing DL inference services with training tasks through spatial sharing. The fundamental concept behind Mudi involves profiling the latency of inference services using a piece-wise linear function that accurately captures resource interference. Leveraging this quantification of interference, Mudi designs a scalable cluster-wide co-location policy, determining the optimal multiplexing of training tasks and inference services to maximize resource efficiency. Furthermore, Mudi incorporates adaptive batching and resource scaling mechanisms to rapidly adapt to the dynamic workloads. Experimental results demonstrate that Mudi improves 42% of GPU resource utilization and achieves up to 2.27× higher training efficiency while satisfying inference SLOs, as compared to state-of-the-art multiplexing methods.

CCS Concepts: • **Computer systems organization** → **Cloud computing**.

Keywords: SLO-awareness, Dynamic workloads, Deep learning

ACM Reference Format:
Wenyan Chen, Chengzhi Lu, Huanle Xu, Kejiang Ye, and Chengzhong Xu. 2025. Multiplexing Dynamic Deep Learning Workloads with SLO-awareness in GPU Clusters. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3696074

*Corresponding author.

1 Introduction

The adoption of DL applications has proven highly effective in various domains, such as face recognition [24, 84], AI painting [31, 69], and intelligent assistants [23, 49]. These applications heavily rely on large DL training (DLT) models, which are subsequently used for online inference services. The computation for DLT models has been significantly enhanced by powerful GPUs, and to handle the increasing demand for evolving DL models, inference computation has also been migrated to GPU servers [61, 79].

Inference services are commonly performed in real-time to meet online requests with strict latency requirements, typically ranging from tens to hundreds of milliseconds per request. These requirements are often defined as Service Level Objectives (SLOs) [8, 14, 58]. However, to meet these SLOs, cluster managers tend to allocate entire GPUs to inference services, resulting in under-utilization of resources during periods of low request arrival rates. According to our trace analysis from Alibaba Cloud, the average resource utilization of inference services is as low as 37%. Similarly, training tasks, which are typically more resource-intensive and time-consuming, also exhibit low resource utilization rates. In fact, in today's production clusters, about 30% of the GPU device duration dedicated to training tasks have a GPU utilization[1] of less than 10% [28, 33]. Despite this low utilization, DL training tasks in production often experience long waiting times before they can be executed [27, 28, 45, 60].

An extensively adopted approach to enhance resource utilization in production clusters is multiplexing [40, 80]. This involves co-locating multiple workloads on the same GPU device using either temporal sharing [19, 20, 73], or spatial sharing techniques via NVIDIA MPS [7, 14, 46, 82], or MIG [39, 48]. Even though multiplexing has proven effective, it can potentially lead to significant interference caused by various resource contention [73, 75], which can ultimately degrade task performance.

Existing approaches for mitigating multiplexing interference in GPU clusters mainly focus on two levels: cluster-wide [73, 82] and per-device [14, 39]. At the cluster-wide level, optimization entails selecting appropriate workloads for co-location across the entire cluster to minimize overall resource contention [29, 75, 77], leveraging techniques such as workload characterization. On the other hand, per-device control tends to be more fine-grained, focusing on precise control of kernel launching [64, 72, 73, 80] or configuring the appropriate resource partition size to separate the execution of co-located workloads within a device [7, 39]. However, a fundamental limitation of the state-of-the-art solutions is that these two-level optimizations are often conducted separately [7, 74, 81], leading to suboptimal results. Typically, device-level control needs to modify the application configuration or alter the underlying hardware resource allocation, which significantly affects task execution behavior in terms of resource usage, and in turn, has a great impact on the efficiency of cluster-wide optimization. Therefore, it is essential to design these two-level optimizations in tandem, although this can be a challenging task. Another significant limitation is that existing solutions struggle to promptly adapt to highly fluctuating online inference services or unobserved training task arrivals [68, 82], which can potentially result in SLO violations or reduced training throughput.

To address these challenges, this paper introduces Mudi, a new Multiplexing system for highly dynamic DL workloads that prioritizes SLO-awareness for inference services in GPU clusters. By multiplexing inference services with training tasks on the same device using spatial multiplexing, there is an excellent opportunity for Mudi to enhance utilization. The key observation behind Mudi is that the latency curve of each inference service can be expressed as a piece-wise linear function in relation to resource partition. Furthermore, function slopes can effectively capture resource interference caused by the co-located workload. Based on this insight, Mudi enables joint optimization of cluster-wide workload co-location with device-level control.

[1]GPU utilization represents the percentage of time during which one or more kernels were executing on the GPU.

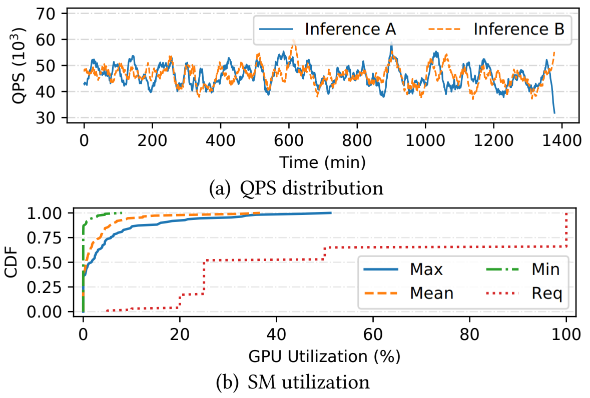
Mudi is built upon a global optimization framework designed to maximize training performance while ensuring compliance with inference SLOs. To tackle this optimization challenge, Mudi employs a predictive approach to accurately estimate the slope of the latency curve for each inference service based on the network architecture of the training task. With this prediction, Mudi intelligently assigns incoming training tasks to the most suitable GPU device in the entire cluster, taking into account various resource configurations and inference service batching sizes to minimize interference. At the device level, Mudi incorporates adaptive batching and dynamic resource scaling to find the best resource partition size and batching size for different rates of request arrival. Mudi designs efficient optimization methods to tackle large search space and simultaneously achieve rapid convergence. Additionally, Mudi incorporates a GPU memory management mechanism to ensure uninterrupted operation of the inference service, effectively addressing potential memory limitations for multiplexing.

As a multiplexing system, Mudi is designed to be highly scalable and flexible. It can seamlessly integrate with various scheduling policies, such as shortest job first, fair sharing, and priority-based scheduling, without requiring any modifications to its core multiplexing algorithms. Additionally, Mudi is fully compatible with MIG, treating each MIG instance as a distinct, smaller GPU. This compatibility enables the concurrent multiplexing of inference services and training tasks on a single GPU. Mudi can handle a wide variety of DL workloads, except in scenarios where the cumulative memory usage of model weights and intermediate results exceeds the GPU memory capacity, such as LLM inference with extensive key-value caches consumed by long contexts.

We have implemented a prototype of Mudi on the Kubernetes platform [1] and conducted evaluations using real-world DL workloads on a private 12-A100 GPU cluster and a simulated 1000-GPU cluster. The experimental results validate the effectiveness of Mudi, showcasing a significant improvement of 42% in GPU resource utilization and achieving 2.27× higher training efficiency while maintaining compliance with inference SLOs, outperforming state-of-the-art multiplexing methods. Furthermore, Mudi attains a high accuracy of 85% for latency prediction, further showcasing its robustness. In summary, our contributions are as follows:

▷ **Quantification of interference.** We present a novel quantification method that effectively captures resource interference by analyzing the slopes of piece-wise linear functions. This quantification is achieved through the utilization of an efficient profiling method with low estimation error.

▷ **Joint optimization within a large search space.** We design a highly scalable optimization method that allows for the co-optimization of two-level interference mitigation within a large search space. This approach enables Mudi to



(a) QPS distribution

(b) SM utilization

Figure 1. Distribution of request arrival rates and GPU resource utilization of online inference services in Alibaba.

rapidly adapt to highly dynamic DL workloads, encompassing both inference services and training tasks.

▹ **Handling unobserved training tasks.** We have developed a methodology that utilizes network architecture characteristics to effectively multiplex previously unobserved online arrivals of training tasks.

2 Background and Motivation

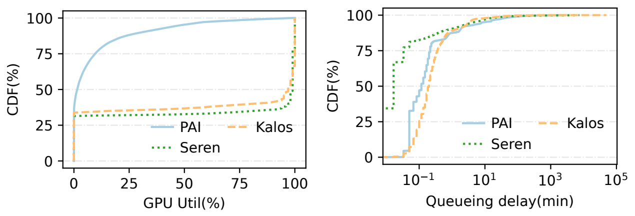
In this section, we present critical DL workload characteristics from the trace analysis on large-scale GPU clusters. These findings drive the development of more efficient spatial multiplexing solutions among DL workloads.

2.1 Characteristics of DL Workloads

2.1.1 Inference services. In production clusters, inference services are commonly regarded as high-priority workloads due to their requirement to meet strict SLOs.

Highly fluctuating inference workloads. To illustrate the dynamic and random characteristics of online inference requests, we conducted an analysis on the distribution of QPS (queries per second) for two typical face recognition services deployed in Alibaba clusters. As shown in Fig. 1(a), the QPS exhibits random fluctuations ranging from 30,000 to 60,000, with no discernible periodic patterns but occasional inflection points over time. This fluctuation pattern holds true across other inference services offered by Alibaba. The highly fluctuating nature of request arrivals poses a challenge when accurately predicting resource requirements.

Underutilization of GPU resources for inference services. To meet their inference SLOs and maintain high throughput, service providers often over-provision GPU resources for inference services in anticipation of potential bursty workloads. However, this practice can result in low utilization even when techniques such as batching are applied. To explore this issue, we conducted an additional analysis on the GPU utilization of all online inference services from Alibaba Cloud, as illustrated in Fig. 1(b). This figure presents the maximum, mean, and minimum GPU utilization, as well as the requested GPU resources, for all services over a week. The results indicate a tendency among inference services



(a) GPU utilization

(b) Queueing delay

Figure 2. GPU utilization and queueing delay of DL training tasks in large-scale GPU clusters.

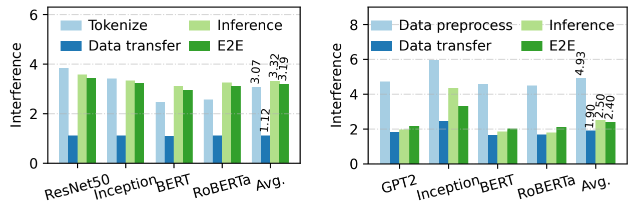
to request more GPU resources than they actually utilize. Specifically, the GPU utilization for all services remained below 52%, with an average SM utilization of less than 37%. These findings strongly suggest that a significant number of allocated GPUs are *underutilized*, even when co-located with other workloads on the same GPU.

2.1.2 DL training tasks. DL training is unlike online inference in that it is computationally intensive and time-consuming. Additionally, there are usually no strict SLOs associated with DL training, which means that there is room for temporary delays or deprioritization of training tasks until GPU resources become available.

Underutilization and long queueing delay of DL training tasks. We analyze traces of training tasks obtained from large-scale GPU clusters at Alibaba (`PAI`) [4, 70, 71], and two LLM clusters (`Seren`, `Kalos`) in Shanghai AI Lab [28, 38]. The results, illustrated in Fig. 2, showcase the CDF of GPU utilization and queueing delay for training tasks across these clusters. Our findings reveal that in `PAI`, `Seren` and `Kalos`, GPU utilization remains nearly zero for approximately 30% of the time and falls below 50% for an impressive 85% of the time in `PAI`. Such wastage of GPU resources is significant. Additionally, queueing delays for DL training tasks are frequently lengthy, with the most extended delay exceeding 1,000 minutes. The main reasons for low utilization include GPU resource over-provisioning and resource fragmentation [71] on each device, as well as communication bottlenecks among multiple training tasks. Additionally, the extended queuing delay is a consequence of a high volume of task requests, with users frequently requesting more GPUs than actually needed. Based on these observations, it is evident that there is a strong desire to multiplex DL tasks in order to enhance GPU resource utilization and minimize queueing delay, ultimately leading to faster training.

2.2 DL Multiplexing: Opportunities and Challenges

2.2.1 Opportunities. While spatial multiplexing has been proven to be effective for improving GPU utilization, it can potentially create substantial interference due to resource contention among co-located workloads such as CPU resource, PCI-e bandwidth, memory bandwidth, and GPU L2 cache [74], which can result in SLO violations for inference services. Therefore, it is crucial to carefully choose compatible workloads for multiplexing to minimize interference.



(a) GPT2 (b) ResNet50

Figure 3. Breakdown of average interference for GPT2 and ResNet50 services multiplexed with other inference tasks.

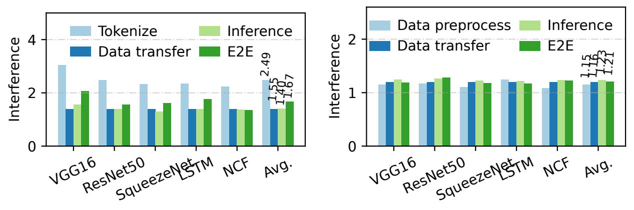
Table 1. Inference services with SLOs from various domains

Field	Model	Dataset	Param (M)	SLO (ms)
♦	ResNet50 [24]	ImageNet [12]	25.6	150
♦	Inception [67]	ImageNet	23.8	120
★	GPT2 [54]	SQuAD [55]	335	100
♡	BERT [13]	SQuAD	110	330
♣	RoBERTa [42]	SQuAD	125	110
♠	YOLOS [15]	COCO [41]	30.7	2200

♦ Image Classification ★ Text Generation ♡ Language Modeling ♣ Question Answering ♠ Object Detection.

Interference between inference services is high. Existing systems, such as gpulet [7] and Astraea [81] primarily concentrate on the multiplexing of various inference services on a single GPU to enhance throughput. However, we show that co-locating inference services together will result in significant higher request latency. To demonstrate this, we conduct experiments to individually multiplex two common inference services - GPT2 for text processing and ResNet50 for image processing - with other services detailed in Tab. 1 using MPS on an A100 GPU. In order to control interference, we adjust the batching size from $\{16, 32, 64, 128, 256\}$ of GPT2 (or ResNet50) and the percentage of GPU resources (GPU%) allocated to each inference service from 10% to 90%. With batching, multiple inference requests are bundled together and served concurrently by the DL backend. In these experiments, we define *interference* as the average value of T_{colo}/T_{solo} across all requests under each configuration, where T_{colo} and T_{solo} represent the latency of co-located and standalone workloads, respectively.

As shown in Fig. 3, the end-to-end (E2E) interference experienced by GPT2/ResNet50 can be significant, depending on the co-located task, with average values of 3.19× and 2.40×, respectively. To examine the underlying causes, we analyze the interference introduced across all three phases: data preprocessing/tokenization, data transmission between the host and GPU memory via the PCI-e channel, and inference execution. We discovered that in standalone mode, each phase of GPT2/ResNet50 account for 4%/7%, 10%/71%, and 86%/22% of the total time, respectively, across different batching sizes and GPU% configurations. However, when GPT2 (or ResNet50) was multiplexed with other inference services, its tokenization (or preprocessing) phase experienced 3.07× (or 4.93×) interference. This can be attributed to the tokenization and image preprocessing phases being multi-threaded and parallel, requiring substantial CPU resources for execution, leading to CPU contention. Moreover, the inference phase of GPT2 is also heavily impacted by the co-located service due to the contention on CPU process. This is primarily caused by the control flow involved during inference, which can often account for up to 72% of the total execution time in the inference stage [78]. As depicted in Fig. 3(b), the inference phase of ResNet50 is also adversely affected, experiencing a 2.5× increase in interference under multiplexing. While the control flow in ResNet50 does not constitute as large a proportion as it does in sequentially generated models, it still suffers from CPU contention.



(a) GPT2 (b) ResNet50

Figure 4. Breakdown of average interference for GPT2 and ResNet50 multiplexed with various training tasks.

Interference between inference and training tasks is moderate. We conduct another similar experiment in which GPT2/ResNet50 was multiplexed with various training tasks, as detailed in Tab. 3. The results, depicted in Fig. 4, reveal that the E2E interference is significantly reduced when the co-located workloads are training, with average values of 1.67×/1.21×. The key reason behind is that the training tasks require longer time to train using a single thread on CPU [83], which diminishes CPU contention. It shows lower CPU and memory utilization when co-locating inference with training, at 21.26% and 11.07% respectively, compared to co-locating inference services together, which exhibited 44.58% CPU and 15.70% memory utilization. Consequently, the tokenization phase for GPT2 (data preprocessing phase for ResNet50) is reduced to 2.49× (1.15×). Similarly, the interference during the inference stage is also significantly reduced, going from 3.92× to 1.4× for GPT2, and from 2.5× to 1.23× for ResNet50. Due to the less contention on the CPU side, the GPU resource utilization is improved when co-locating inference with training. Specifically, the SM utilization rate was higher at 88.87% when co-locating inference with training, versus 65.93% in the co-located inference services scenario. Furthermore, the frequency of data transfers required by training is less than that of inference. This results in reduced data transfer interference when co-locating inference of image tasks with training, with a factor of 1.16×. In contrast, the data transfer interference is higher at 1.9× when co-locating inference of image tasks with other inference services.

Takeaway. The interference between inference and training is relatively mild due to lower contention on CPU resources and PCI-e bandwidth, offering a beneficial opportunity to enhance GPU utilization while meeting the SLOs of inference and improving training throughput.

2.2.2 Challenges. Multiplexing inference services with training tasks can provide benefits, however, there are several challenges involved in achieving global optimization for the entire cluster in practical scenarios.

C_1**: Dynamic workload arrivals.** The first challenge is due to the highly dynamic nature in both inference and training tasks. As illustrated in Fig. 1(a), the arrival rates of requests for inference services exhibit significant fluctuations. Therefore, to effectively mitigate interference, a real-time per-device control mechanism is essential to adapt to these varying arrivals. However, this poses a significant challenge as the mechanism must be able to anticipate outcomes in advance to ensure compliance with SLOs. Moreover, unlike long-running inference tasks that can be thoroughly profiled in advance, training tasks fall into the category of batch processing jobs, which encompass various types due to the inclusion of fine-tuning workloads [16, 26, 53]. Consequently, the per-device control mechanism needs to frequently handle previously unseen training jobs.

C_2**: Intricate coupling between cluster-wide task co-location and device-level configurations.** The second challenge arises from the complex interdependence between task co-location, batching configuration [7, 8], and resource scaling. As shown in Fig. 4, the co-location pattern has a heavy impact on the interference introduced to inference services, making it crucial to fine-tune device-level configurations such as resource partitioning and batching sizes to maintain SLOs. However, changes to task configurations can also impact task execution behaviors, which can have a significant impact on the efficiency of optimizing cluster-wide co-location. The situation becomes even more complicated when optimizing training throughput simultaneously.

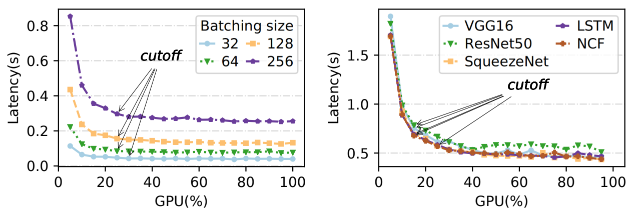
C_3**: Large optimization space.** The third challenge stems from the vast number of potential combinations of batching sizes and resource partition sizes. The batching size for each inference can range from 2 to one thousand depending on the GPU memory limit. Additionally, the partition size can vary from 1% to 100%. Consequently, the search space for finding optimal multiplexing solution exceeds $1000^N \times 100^N$, where N is the number of inference services.

3 Mudi Overview

In order to tackle the challenges outlined in § 2.2.2, we propose Mudi, a new and efficient system for multiplexing inference services and training tasks using MPS within a cluster. Mudi aims to achieve the following primary goals: 1) Ensure the SLOs for inference services are met under dynamic workloads. 2) Maximize the throughput for DL training tasks.

3.1 Key Ideas

I_1**: Explicit modeling of inference latency.** Mudi relies on explicit quantification of latency for inference services concerning resource partitioning, batching size, and resource interference. This quantification facilitates the implementation of real-time dynamic batching and resource scaling, as it



(a) GPT2 with solo-run (b) GPT2 with colo-run

Figure 5. Latency of GPT2 with various batching sizes under solo-run and colocation with training tasks using batching size=256 on an A100 GPU.

allows for convenient justification of whether the SLOs of online services are being maintained, thus addressing challenge C_1. Simultaneously, it enables the seamless coordination of workload co-location at a cluster-wide level and interference control at the device level, all within a single optimization framework, effectively addressing challenge C_2.

The key observation supporting this quantification is that the inference latency follows a piece-wise linear function in relation to resource partitioning, as illustrated in Fig. 5(a). There exists a cutoff point where latency only marginally decreases when resource allocation exceeds this threshold and the exact cutoff points vary depending on the batching sizes utilized. Furthermore, this piece-wise linear relationship persists even when inference is multiplexed with other training tasks, as shown in Fig. 5(b).

I_2**: Predicting interference using underlying network architectures.** The slope of the piece-wise linear function of the interference latency effectively captures the interference introduced by the co-located training task. Mudi leverages the network architecture of the DL training task to estimate this slope in advance based on offline profiles, allowing it to easily adapt to previously unobserved workloads and effectively address challenge C_1. Moreover, this prediction empowers Mudi to easily identify the smallest resource allocation that ensures SLOs for inference services under each selection of batching size. Consequently, it reduces the search space effectively, addressing challenge C_3.

3.2 System Architecture

We have developed three key components in Mudi, as shown in Fig. 6. The **Offline Profiler** is responsible for profiling the latency and interference of inference services in multiplexing mode. The **Online Multiplexer** receives DL training tasks from end-users and finds an optimal device to assign the training task that introduces minimal interference to the inference service. Meanwhile, the **Local Coordinator** exposes the device stats and controls the configurations.

The **Offline Profiler** operates offline to profile samples for estimating interference caused by different workload co-locations. It comprises two modules. First, the *Latency Profiler*❶ profiles the latency of each inference service when multiplexed with training tasks under various batching sizes and GPU%. Second, the *Interference Modeler*❷ models the

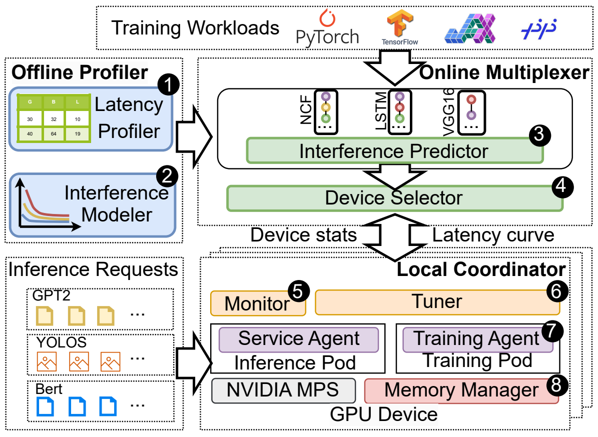


Figure 6. The system architecture of Mudi.

degradation in service latency caused by interference based on network architectures from training tasks.

The **Online Multiplexer**, functioning as a centralized component across the entire cluster, comprises two modules: the *Interference Predictor*❸ and the *Device Selector*❹. The *Interference Predictor* utilizes latency curves obtained from the **Offline Profiler**, along with extracted network architectures, to forecast the interference caused by incoming training tasks. The *Device Selector* is responsible for assigning training tasks to GPUs, selecting the device that introduces minimal interference for all potential co-location patterns.

The **Local Coordinator** is deployed on each GPU device and consists of four primary modules. The *Monitor*❺ continuously monitors the QPS of each inference service. When the change in QPS exceeds a certain threshold, it triggers the tuning process to adjust batching size and GPU% to ensure the SLOs. The *Tuner*❻ adopts efficient optimization methods to determine the optimal batching and GPU% for each co-located workload based on performance profiles obtained from the **Online Multiplexer**. The *Agent*❼ is integrated into both the inference (*Service Agent*) and training (*Training Agent*) processes. The *Service Agent* controls the batching size and GPU% for the corresponding inference service, and *Training Agent* controls the GPU% that is allocated to its training task. These processes are automated and allow the system to adapt to the dynamic QPS of inference. The *Memory Manager*❽ can swap the memory of training tasks between device and host to prevent out-of-memory errors of co-located inference service.

4 Inference Latency Quantification

In this section, we present the design details of Mudi's quantification approach for inference latency.

4.1 Offline Profiling

4.1.1 Inference latency profiling. As highlighted in § 3.1, the inference latency can be characterized as a piece-wise linear function in relation to the resource partition size (denoted as GPU%). And the co-located training task has a

Table 2. Fitting error of three representative models

Model \ Samples	5	6	7	8	9
Polynomial fitting	9.81	8.31	7.25	6.71	5.53
MLP fitting	**7.32**	7.36	7.24	7.15	6.99
Piece-wise linear	10.03	**6.41**	**4.27**	**3.91**	**3.78**

heavy impact on the slope of the latency curve across different batching sizes. Hence, the *Latency Profiler* opts to profile the relationship between the latency of each inference service and GPU% value under a fixed batching size and co-located training task.

Specifically, the *Latency Profiler* gathers the P99 tail latency data for each inference service i across a range of GPU% values from 10% to 90%, with increments of 10%. The batching size is selected from $\{16, 32, 64, 128, 256, 512\}$, and the co-located training task is chosen from the first five types listed in Tab. 3. Let $L^i_{b,\Psi}$ denote the latency under a specific GPU% value, denoted by $\triangle_i$, given batching size b and co-location workload Ψ. The *Latency Profiler* fits all collected samples $\{(\triangle_i, L^i_{b,\Psi})\}_i$ into a piece-wise function:

$$L^i_{b,\Psi} = \begin{cases} k^i_{\Psi,1} \cdot (\triangle_i - \triangle_0) + l_0, & \triangle_i \le \triangle_0, \\ k^i_{\Psi,2} \cdot (\triangle_i - \triangle_0) + l_0, & otherwise. \end{cases} \quad (1)$$

Here, $k_{\Psi,1}, k_{\Psi,2}$ symbolize the slopes of the piece-wise linear curve, while $(\triangle_0, l_0)$ represents the cutoff point. To determine the cutoff point, the *Latency Profiler* calculates the curvature of each set of three consecutive points and identifies the middle point in the set that yields the lowest curvature [59]. Using this point as a basis, Mudi employs the small-least-squares method to fit the slopes $k_{\Psi,1}, k_{\Psi,2}$.

While more complex models like polynomial fitting or neural network models have the potential to enhance profiling accuracy, they require a substantially larger sample size to achieve accuracy comparable to that of piece-wise linear fitting. As shown in Tab. 2, the piece-wise linear model consistently outperforms the other models when using fewer than 10 training samples. Although enlarging the training set can improve fitting accuracy, it also significantly increases the profiling overhead in the interference modeling stage (§ 4.1.2), which requires estimating the parameters of fitting models across various batching sizes and co-located training tasks. Additionally, we observed a significant decrease in testing error for the piece-wise model when the number of training samples increases from 5 to 6. Therefore, to balance profiling overhead and fitting accuracy, we opt for the piece-wise model with 6 training samples. Moreover, we also evaluated the E2E performance of the multiplexing approach based on polynomial fitting. The results indicate that the resulting average inference latency/training time is 6.3%/15.8% higher than those using piece-wise models.

4.1.2 Interference modeling. With the fitted functions from the *Latency Profiler*, the *Interference Modeler* constructs a learning model to measure interference.

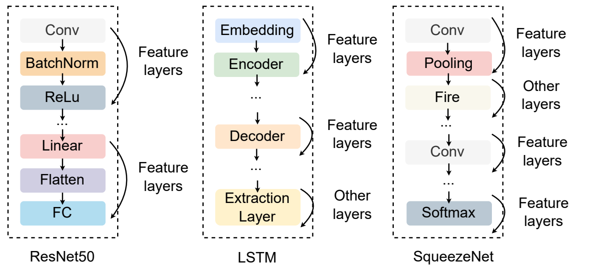


Figure 7. The network layers of representative training tasks that Mudi identifies, where Extraction Layer, Fire, and others, are classified into other_layers.

To train the learner, the *Interference Modeler* utilizes the network architecture of the co-located training task and the batching size b of inference i as input features, denoted as $\mathcal{X} = [\Psi, b]$. Regarding the network architecture Ψ, the *Interference Modeler* primarily focuses on the number of specific layers such as [conv, linear, activations, embeddings, encoder, decoder, flatten, batch_normalization, fc, pooling, other_layers], as depicted in Fig. 7. These layers are commonly found in various DL models and significantly impact the GPU cycles and memory resources during parameter updates in training, thereby affecting inference latency. Hence, the *Interference Modeler* extracts these network layers for each training task involved in the offline profiling process. Mudi selectively excludes unpopular layers to reduce the need for a large number of samples for accurate predictions and to prevent overfitting to unobserved training tasks.

The output from the learner includes the slopes $k^i_{\Psi,1}, k^i_{\Psi,2}$, as well as the cutoff point $(\triangle_0, l_0)$, which can be denoted as $\mathcal{Y} = [k^i_{\Psi,1}, k^i_{\Psi,2}, \triangle_0, l_0]$. To establish the relationship between $\mathcal{X}$ and $\mathcal{Y}$, the *Interference Modeler* collects all $\mathcal{Y}$ values from the *Latency Profiler* and utilizes lightweight models such as random forest (RF), support vector regression (SVR), etc., for each inference service i. Additionally, the *Interference Modeler* determines the optimal model as the learner for each metric in $\mathcal{Y}$ individually. Furthermore, the prediction model is adaptable and can be incrementally updated to accommodate new workloads that arrive at the cluster.

4.2 Online Prediction

When a new training task j arrives, the *Training Agent* of Mudi starts by extracting its network architecture to evaluate the interference that may be introduced to potential co-located inference services. For models with static computation graphs, such as ONNX/TensorFlow, *Training Agent* directly extracts their network layers from the model files. For dynamic computation graph model like Pytorch, Mudi selects a GPU device with the lowest GPU utilization throughout the cluster and runs the training task on it for a mini-batch to trace the invoked modules. This allows the *Training Agent* to collect the quantities of each identified network layer.

With the input features $\mathcal{X}$, the *Interference Predictor* employs the corresponding learning model to predict the slopes and cutoff point for the associated piece-wise linear function. As a result, the **Online Multiplexer** is capable of forecasting the latency of an inference service i under each GPU% value $\triangle_i$, with a batching size b_i, while being co-located with a training task j comprising network architectures Ψ_j. In simpler terms, the **Online Multiplexer** obtains $L^i_{b,\Psi} = \mathsf{P}_i(b_i, \triangle_i, \Psi_j)$. These latency predictions are then utilized by the *Device Selector* to make multiplexing decisions.

5 Online Multiplexing Approach

In this section, we present Mudi's multiplexing optimizations. To provide a comprehensive understanding of the key designs, we begin by introducing the global optimization framework that underlies Mudi.

5.1 Optimization Model

Considering there are N inferences services across the entire cluster, it is feasible for an inference service to be co-located with multiple training tasks on a single GPU device that has limited GPU memory capacity. Once a new training task j arrives at the cluster, Mudi assigns it to a specific device and configures an appropriate batching size and resource partition for the co-located inference service i based on the inference QPS W_i. The primary objective is to guarantee the SLO for inference while simultaneously maximizing the training throughput, effectively minimizing the training time. This leads to the following optimization problem:

$$\begin{aligned} \min_{\left\{x^i_j,\, b_i,\, \triangle_i\right\}} \quad & \sum_{j \in \mathsf{A}(t)} \sum_{i=1}^{\mathsf{N}} x^i_j \cdot \mathsf{Iteration}_j(b_i, \triangle_i) \\ \text{s.t.,} \quad & \frac{\mathsf{W}_i}{b_i} \cdot \mathsf{P}_i(b_i, \triangle_i, \Psi_j) \le \mathsf{SLO}_i,\ \forall 1 \le i \le \mathsf{N}, \\ & \triangle_i \le 1,\ \forall 1 \le i \le \mathsf{N}, \\ & \sum_{i=1}^{\mathsf{N}} x^i_j = 1, \text{and } x^i_j \in \left\{0, 1\right\}. \end{aligned} \tag{2}$$

In the formulation, $\mathsf{A}(t)$ denotes the set of training tasks to be executed, and x^i_j is a binary optimization variable that indicates whether training task j is multiplexed with inference service i. $\mathsf{Iteration}_j(b_i, \triangle_i)$ captures the mini-batch training time for task j, which is influenced by the batching size and resource partition of the inference service due to resource interference. The first constraint ensures that the latency of all requests from each service adheres to the SLO. The second constraint specifies that an inference service is limited to utilizing a maximum of 100% of the GPU resources.

In general, finding the optimal solution to this problem is intractable. The difficulty arises from two major aspects. First, accurately quantifying the training throughput in the presence of resource interference is a complex task, rendering existing optimization solvers unsuitable for the problem. Moreover, even if offline profiling is employed for estimating the throughput in advance, it fails to accommodate unobserved workloads. Second, the expansive optimization spaces render it impractical to exhaustively enumerate all potential co-locations and per-device configurations in order to determine the optimal solution. This lack of scalability arises from the substantial overhead required to collect mini-batch training time when considering every possible combination. To address these issues, Mudi employs an efficient approximation solution that initially involves identifying the ideal inference service to be multiplexed with an incoming training task based solely on interference estimation. Subsequently, Mudi performs local optimization on each device.

5.2 Cluster-wide Workload Co-location

Upon receiving the latency curve $L^i_{b,\Psi} = \mathsf{P}_i(b_i, \triangle_i, \Psi_j)$ from the *Online Predictor*, the *Device Selector* proceeds to quantify the average value of the slopes derived from the curve. This metric not only measures the extent of performance interference that the training task may impose on the inference i, but also provides an evaluation of the potential advantages of multiplexing the new training task with i. On one hand, a smaller slope indicates a reduced level of interference, thereby enhancing the ability to meet the SLO for the inference service i. On the other hand, a smaller slope implies that the inference service is less sensitive to variations in resource partition size. This characteristic enables a higher allocation of GPU resources to the training task, which is advantageous for optimizing the objective in (2).

Subsequently, the *Device Selector* assigns an incoming training task to the device that yields the smallest average slope across all batching sizes within the set $\{16, 32, 64, 128, 256, 512\}$. This assignment strategy effectively resolves the optimization variable x^i_j in (2), allowing for a quick response to previously unseen workloads. Note that for each inference service, changes in co-location occur infrequently, as a new co-location decision is made for pending training tasks only after an existing training task has been completed.

5.3 Device-level Multiplexing

When a training task is assigned to the selected GPU device or the change in the QPS of an inference service i exceeds the threshold, the *Tuner* takes on the responsibility of finding the optimal configuration $(b_i, \triangle_i)$ for co-located workloads. The goal is to minimize $\mathsf{Iteration}_j(b_i, \triangle_i)$ of the training task while adhering to the SLO constraint for inference i.

Tuning both the batching size and GPU% simultaneously may not be necessary and can be costly for several reasons. First, based on our observations of DL workload characteristics during GPU% and batching size tuning, we found that batching size has a stronger impact on running efficiency. Adjusting the batching size alone following the explicit quantification of inference latency may be sufficient to meet SLOs most of the time. Second, due to the limitations of MPS, one needs to terminate and restart the process to update the corresponding GPU% environment variable, which adds complexity and overhead. Therefore, the *Tuner* employs a two-phase approach for decoupling: adaptive batching and dynamic resource scaling. This approach enables Mudi to effectively reduce the optimization space and accommodate the uncertainties inherent in a cluster environment.

5.3.1 Adaptive batching. The *Tuner* leverages Bayesian Optimization (BO) [17] to find the optimal batching size configuration for each inference service. BO is a blackbox optimization method that does not require the exact expression of the objective function $\mathsf{Iteration}_j(b_i, \triangle_i)$. Instead, it uses an iterative approach to guide its exploration process.

It is important to note that the relationship between batching size of inference service and training throughput is not strictly monotonic. This is due to the varying ratio between data transfer time and computation time during inference, which is highly dependent on the characteristics of the inference service and the specific batch being processed. In other words, a larger batching size does not necessarily result in higher interference to a training task. This uncertainty can be effectively addressed through the utilization of BO. Another significant advantage of BO is its ability to capture interactions and relationships between configurations and objective functions based solely on real-time feedback.

To make BO more efficient, The *Tuner* selects Gaussian Process (GP) [56] as the surrogate model for approximating the objective function min $\mathsf{Iteration}_j(b_i, \triangle_i)$, in order to reduce the search cost. In each round of BO, the surrogate model is updated using online sampled mini-batch iteration times obtained from previously adopted configurations $(b_i, \triangle_i)$, provided by the *Training Agent*. Furthermore, the *Tuner* devises an efficient acquisition function based on the lower confidence bound (LCB [36, 63]) to guide the exploration process effectively:

$$\min_{b_i \in \mathcal{R}} \mathcal{A}(b_i) = \mu(b_i, \triangle_i) - \beta_n^{1/2} \sqrt{\sigma(b_i, \triangle_i)}. \tag{3}$$

Here, $\mathcal{R}$ denotes the search space that contains all candidate batching sizes, $\mu(\cdot)$ and $\sigma(\cdot)$ represent the estimated mean and variance by the GP, respectively. β_n is incorporated to balance the exploration and exploitation of BO, and n is the iteration time. In the *Tuner* module, we set $\beta_n = 2\log(|\mathcal{R}|/n^2)$, which facilitates faster convergence.

During the exploration process, it is essential for the *Tuner* to ensure that the candidate configuration does not violate the SLO, as specified by the first constraint in (2). To address this, the *Tuner* incorporates the constraint into the GP framework, continuously updating the surrogate model until convergence is achieved for $\mathcal{A}(b_i)$. The evaluation conducted in § 7.5 demonstrates that GP-LCB effectively identifies the optimal configuration within 25 iterations. Notably, updating the batching size merely necessitates passing the new b_i as a parameter to the inference service. This allows for on-the-fly updates without restarting the service.

5.3.2 Dynamic resource scaling.

The *Tuner* module dynamically adjusts the GPU resource partitions of the inference service in two cases: 1) when a new training task is multiplexed with an inference service, and 2) when the *Monitor* detects that a change in the QPS of the inference service.

In case a new training task j is assigned to co-locate with inference service i, the *Tuner* initializes a GPU% value for i to be the maximum value among all cutoff points under different batching sizes. Subsequently, adaptive batching is performed to determine the optimal batching size for inference based on this initial resource partition. However, since the initial GPU% value may be excessive for service i, the *Tuner* adjusts the resource allocation to improve training efficiency. Specifically, the *Tuner* determines the minimum GPU% that maintains the SLO of service i:

$$\triangle_i = \arg\min \triangle, \text{ s.t., } \mathsf{W}_i/b_i \cdot \mathsf{P}_i(b_i, \triangle, \Psi_j) \le \mathsf{SLO}_i. \quad (4)$$

To address the optimization problem (4), the *Tuner* leverages the CVXPY library and utilizes the ECOS solver. To accommodate prediction errors, the *Tuner* sets the actual GPU% value to be 10% larger than the solution obtained from the solver.

In the event that the *Monitor* detects a QPS change rate surpassing 50%, the *Tuner* will modify the GPU resource partition size to ensure compliance with the SLO. To accomplish this, the *Tuner* solves (4) to determine the optimal GPU% configurations for the co-located workloads. Following that, the *Tuner* initiates adaptive batching to find the optimal batching size that maximizes the training efficiency. However, updating the GPU% value requires terminating the previous inference service and restarting a new one with the updated GPU% configuration, which results in a tens-of-second time overhead. To hide the reconfiguration overhead, the *Tuner* prepares a shadow instance with the new configuration, running as a separate process tailored to the current QPS. Once this shadow instance is ready to handle requests, it takes over as the active inference service, and the old service is subsequently terminated. In rare circumstances, even with adaptive tuning and resource scaling, it is not feasible to meet the SLO given the current bursty QPS. In such cases, Mudi avoids using multiplexing and instead preemptively pauses the training task until suitable resources become available.

5.4 Analysis of Mudi's Optimality

Mudi's methodology hinges on assessing the average performance across various batching sizes using the piece-wise linear function to enable cluster-wide co-location. As a result, a deviation emerges between Mudi's approach and the optimal solution concerning co-location decisions. The optimal solution is characterized as the one that minimizes the iteration time of the training task while meeting the SLO of the multiplexed inference service. The effectiveness rate of Mudi's co-location policy is represented as $\mathcal{P}$, denoting the probability that Mudi successively identifies the optimal co-location. Hence, for all training tasks M, the anticipated performance of Mudi's policy is constrained by the worst-case scenario, formulated as follows

$$\mathcal{E} \le \frac{1}{M} \sum_{j=1}^{M} \left(\mathcal{P} \cdot \mathsf{Iteration}_j^* + (1-\mathcal{P}) \cdot \mathsf{Iteration}_j^\dagger \right), \quad (5)$$

where $\mathsf{Iteration}_j^*$ and $\mathsf{Iteration}_j^\dagger$ denote the iteration times training task j under the optimal and worst cases, respectively. We evaluated the effectiveness of Mudi's co-location policy within a physical cluster, as elaborated in § 7.1. The findings reveal that Mudi attains an effective rate of 92.67% in identifying the optimal co-location. Subsequently, through an exhaustive approach, we determined the iteration times and SLO violations for the best and worst configurations (co-location, batch size, and GPU%). The outcomes show that $\mathcal{E}$ stands at 1.10 for iteration time and 1.08 for SLO violation, with the optimal values as the reference point. This implies that the difference between Mudi's policy and the optimal solution is limited to a maximum of 10%.

5.5 Extension to Multiplexing More Tasks

Mudi can be seamlessly extended to support more training tasks on a single GPU. However, as analyzed in [6], the marginal benefit of multiplexing training tasks typically begins to diminish beyond three tasks. Moreover, it is crucial to ensure SLO compliance for inference, which is considered a high-priority workload. Therefore, to maintain inference performance and maximize training efficiency, Mudi enables the multiplexing of one inference service and no more than three training tasks on the same GPU. To facilitate this, the *Latency Profiler* enhances its sampling for each inference service by co-locating two or three training tasks. Mudi subsequently employs these expanded samples to establish piecewise-linear fittings. During online prediction, the *Interference Modeler* designates the cumulative feature layers as Ψ for all training tasks co-located with inference i. During the resource scaling phase, the *Tuner* first determines the optimal resource partition size for inference i, and then evenly distributes the unassigned portion of GPU resources among all associated co-located training tasks.

5.6 GPU Memory Management

In order to prevent out-of-memory errors caused by training large models or a sudden surge in load for inference services on devices with limited GPU memory [35, 73], Mudi integrates a memory management mechanism. This mechanism dynamically swaps the memory of training tasks between the device and host, ensuring that the SLO requirements for inference are consistently met.

The management mechanism maintains a unified memory pool that is shared between the host and device, with a single pointer used to manage memory swapping by CUDA driver API. Mudi prioritizes inference memory pointer address on the device and that of the training tasks on the host. When there is insufficient memory, Mudi swaps the memory of training from the device to the host. To accomplish this, Mudi utilizes a middleware between DL workloads and CUDA dynamic-link libraries. Notably, this approach differs from current solutions that require modifications of DL frameworks [73], making it more flexible to use.

6 Implementation Details

We have implemented Mudi on Kubernetes v1.23.2 [1], using over 4,000 lines of Go code and 1,400 lines of Python code. All DL workloads are executed within Docker containers.

Online Multiplexer. By leveraging the Kubernetes scheduling framework [2], we implement two policies, namely the *Interference Predictor* and the *Device Selector*, as score plugins. The scheduler listens for task submission events from the Kubernetes `API server` and maintains a queue to cache submitted workloads that will be scheduled on a FCFS (first-come-first-served) basis. We also develop a `GPUShare-Device-Plugin` to expose the device stats including the GPU resource utilization to the *Device Selector*.

Local Coordinator. We utilize `DaemonSet` to implement the *Local Coordinator*, which ensures that all eligible nodes run a copy of a `Pod`. For a worker with multiple devices, our *Local Coordinator* creates a Go coroutine for each device and triggers the tuning of GPU%/batching processes for co-located workloads. The *Tuner* is implemented as a `DaemonSet` service using Python. Additionally, the updated configurations and other intermediate results are stored using `ETCD`, a highly available distributed database. When a configuration key/value pair is updated, the controller process in the *Agent* belonging to each inference service or training task perceives the new configuration and updates accordingly in a timely manner. The *Training Agent* also records the mini-batch training time for the associated task.

Monitor. We deploy the *Monitor* as a long-running process using a Go coroutine on each device. The *Monitor* periodically collects the QPS of each online inference service, and for each batch of requests, it tracks the corresponding latency and stores this information in `ETCD`. In cases where the *Monitor* detects that the SLO is at risk of being violated, it triggers adaptive batching or resource scaling accordingly.

Memory Manager. To implement the memory swapping scheme, we utilize the `cudaMallocManaged` API from CUDA Unified Memory [47]. This middleware intercepts CUDA memory APIs like `cuMemAlloc` and substitutes them with unified memory allocation calls,`cuMemAllocManaged`. We deploy this middleware on each server. Crucially, this memory restructuring is entirely transparent to users, eliminating any need for them to modify their application code.

7 System Evaluation

In this section, we present a comprehensive evaluation of Mudi on both a physical cluster and a simulated cluster using representative DL workloads, as shown in Tab. 1 and Tab. 3.

Table 3. DL training tasks from various domains

Field	Task Name	Dataset	Optimizer	batchsize	Size	Frac.
♦	VGG16 [62]	CIFAR10 [37]	Adam	512	S	14%
♦	SqueezeNet [32]	CIFAR10	Adam	512	S	14%
♦	ResNet50 [24]	CIFAR100 [37]	Adam	1024	S	14%
▷	NCF [25]	MovieLens [22]	SGD	1024	M	12%
♣	LSTM [51]	Wikitext-2 [44]	Adadelta	256	M	12%
□	AD-GCL [66]	Reddit[3]	Adam	64	M	12%
♣	Bert [13]	SQuAD [55]	AdamW	32	L	12%
♠	YOLOv5 [34]	COCO [41]	SGD	64	L	10%
♦	ResNet18 [24]	ImageNet [12]	SGD	128	XL	2%

♦ Image Classification ▷ Recommendation System □ Social Network ♡ Language Modeling ♠ Object Detection ♣ Question Answering.

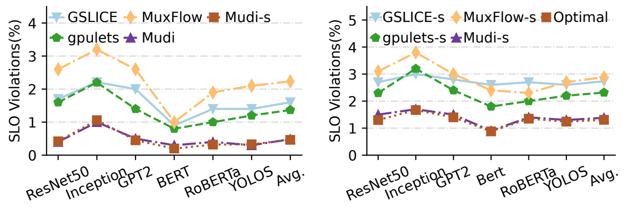
7.1 Experimental Setup

Physical cluster. We conducted experiments in a private 3-node cluster, each node is equipped with 4 NVIDIA A100 GPUs featuring 40GB memory, and 2 Intel Xeon Gold 6230R CPUs. The CUDA version is 12.0 and NVIDIA driver version is 525.15.06. All DL workloads are developed using Pytorch 1.9.0 and Tensorflow 2.5.0.

Simulated cluster. We simulated a large-scale cluster consisting of 1000 GPUs to assess the scalability of Mudi. To mimic multiplexing behavior, we fitted functions that delineate the interplay between latency/minibatch time with various *batching sizes* and `GPU%` for each colocation of inference services and training tasks, utilizing profiles obtained from the physical cluster. Specifically, we used 270 samples to build the fitting functions for each inference service and training task. These functions are then used to generate performance feedback at runtime in a simulated environment.

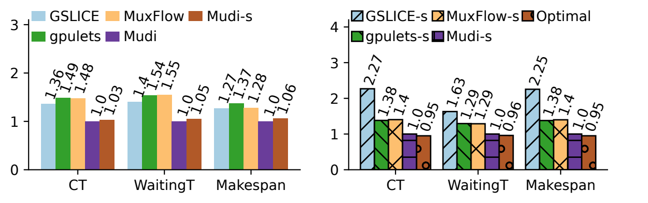
DL workloads. We chose six inference services outlined in Tab. 1 and nine training tasks listed in Tab. 3 within various domains. For each inference service, we simulated the arrival of requests using a Poisson random distribution with an average inter-arrival time of 5ms. The training task arrival process follows Microsoft production traces [45] in the physical cluster, while it is scaled by a factor of 80 in the simulated cluster. We categorized each training task into different scales according to their running time: Small (<1 GPU-hour), Medium (1∼10 GPU-hours), Large (10∼100 GPU-hours) and XLarge (>100 GPU-hours).

Baselines. We compared Mudi with three baselines: *GSLICE [14]*, the core concept of GSLICE involves allocating GPU partitions based on feedback regarding inference latency and throughput; *gpulets [7]*, it proposes a novel GPU virtualization unit that enables dynamic resource allocation for inference services; *MuxFlow [82]*, it implements dynamic SM allocations and matching-based scheduling techniques to enhance the efficiency of offline training tasks. Since GSLICE and gpulets only address inference services, we have incorporated a tuning mechanism for training in these baselines to ensure a fair comparison. This tuning mechanism is similar to the one used for inference in GSLICE and gpulets. In Mudi and MuxFlow, the profiling is constrained to include only the first five types of training tasks.



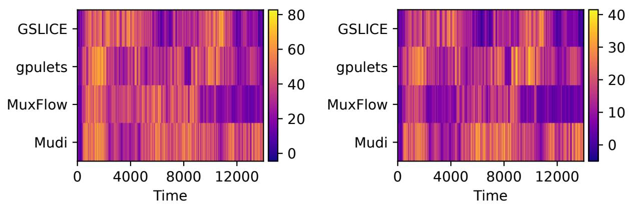
(a) Small-scale (b) Large-scale

Figure 8. SLO violation rates of all inference services, x-s refers to a system in a simulated cluster.



(a) Small-scale (b) Large-scale

Figure 9. Normalized CT, waiting time, and makespan comparison for training tasks.



(a) SM utilization (b) Memory utilization

Figure 10. Average resource utilization in physical cluster.

7.2 End-to-End Performance

We demonstrate Mudi's effectiveness through executing 300 training tasks (**small-scale**) in a physical cluster and 5,000 training tasks (**large-scale**) in a simulated cluster.

Inference latency. Our analysis (as illustrated in Fig. 8) reveals that Mudi yields the lowest SLO violation rates in terms of P99 tail latency for each inference service among these four systems in both physical and simulated clusters. Here, the SLO violation rate is defined as the percentage of times the tail latency exceeds the SLO. Mudi achieves SLO violation rates as low as 0.5% (1.2%) on average in physical (simulated) cluster. On the whole, Mudi achieves a reduction of up to 5.5×, 2.2×, 4.2×, 2.3×, 3.8×, and 6× for ResNet50, Inception, GPT2, BERT, RoBERTa, and YOLOS, respectively. On the other hand, MuxFlow experiences the highest SLO violations due to its reliance on pre-profiled training tasks, rendering it less adaptable to unseen workloads. Consequently, this leads to high interference on each inference service, resulting in a noticeable increase in SLO violation rate. GSLICE and gpulets demonstrate higher SLO violations in inference than Mudi, mainly because they do not consider cluster-wide co-location interference. Conversely, Mudi utilizes the network architectures of training tasks to estimate interference in advance. These design mechanisms enable Mudi to effectively mitigate cluster-wide interference for inference services with unobserved training tasks. Moreover, to maintain this low SLO violation rate, Mudi efficiently handles memory by swapping memory between the device and host for training tasks. This practice ensures that the SLO requirements for inference are consistently met, especially during unexpected spikes in request loads. We observed that memory swapping occurred, on average, once every 23.08 minutes for VGG16, 37.5 minutes for SqueezeNet, 19.20 minutes for ResNet50, 33.33 minutes for NCF, 27.30 minutes for LSTM, 28.57 minutes for AD-GCL, 23.08 minutes for BERT, 19.35 minutes for YOLOv5, and 17.14 minutes for ResNet18.

Training efficiency. To assess the effectiveness of Mudi in improving training efficiency, we utilize three metrics: ***CT***, ***WaitingT***, and ***makespan***. CT represents the task completion time. WaitingT quantifies the duration that a task has spent waiting before being executed. And makespan refers to the total time taken to complete all training tasks.

Fig. 9 illustrates that Mudi reduces the overall CT by up to 2.27×, 1.49×, and 1.48×, respectively, in comparison to GSLICE, gpulets, and MuxFlow in large-scale clusters. Furthermore, Mudi demonstrates the ability to reduce waiting time and makespan by up to 1.63× and 2.25×, respectively, compared to the baselines. This is attributed to the fact that GSLICE and gpulets primarily focus on inference services without considering optimizing training efficiency through spatial multiplexing, resulting in lower training throughput. Mudi holds an advantage over MuxFlow due to its adaptive per-device control approaches. These strategies effectively enhance training throughput, resulting in faster completion and reduced waiting times. Additionally, the results depicted in Fig. 8(a) and Fig. 9(a) indicate that the simulator can effectively reproduce behavior that closely resembles that of the physical cluster, with minor discrepancies of less than 4.7% in both SLO violations and CT.

GPU utilization. As depicted in Fig. 10, Mudi exhibits significantly higher utilization of SM and memory compared to the baselines, particularly during the latter half of the timeframe due to the increased prediction accuracy. Over the long run, Mudi achieves up to 60% utilization of SM and 35% utilization of memory across all GPU devices, which is 42% higher and 19% higher than baselines, respectively. This enhancement can be attributed to the effective co-design of cluster-wide and device-level multiplexing. Specifically, our observations reveal that cluster-wide optimization contributes to an improvement of up to 37% in SM utilization and 16% in memory utilization, while device-level control introduces an additional improvement of 39% in SM utilization and 17% in memory utilization. These findings strongly emphasize the effectiveness of our multiplexing designs.

Optimality analysis. To validate the optimality of Mudi, we profile additional samples and devise an `Optimal` baseline by employing an exhaustive search method to select the best colocation and configurations for both inference and training

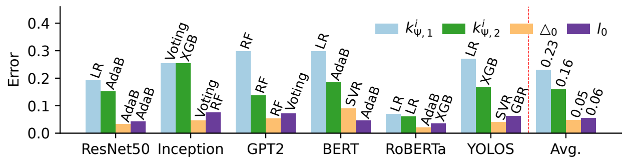


Figure 11. The accuracy of interference modeling for all inference services. The note located to the left of the red line at the top of each bar indicates the best prediction models.

tasks. A comparative analysis in the simulated cluster demonstrates that Mudi's inference SLO violation rate is nearly equivalent to the Optimal baseline, as depicted in Fig. 8(b), with an average discrepancy of only 5.86%. Meanwhile, in Fig. 9(b), the training tasks in Mudi exhibit exceptional alignment with the optimal baseline in terms of CT, waitingT, and makespan. These metrics deviate from the optimal values by no more than 5%, underscoring that Mudi's multiplexing mechanism closely aligns with the optimal solution.

7.3 Microscopic Analysis

Accuracy of interference modeling. The accurate modeling of interference is crucial for Mudi to make optimal multiplexing decisions. To illustrate this, we utilized 70 offline collected latency samples to train a dedicated model for each parameter within the piece-wise linear function associated with each inference service. Additionally, we used 20 latency samples collected from the co-location with the last four unobserved training tasks in Tab. 3 to fit the piece-wise linear functions, with these parameters serving as the test set. The prediction error is defined as Error $= |y_{pred} - y_{true}|/y_{true}$. As shown in Fig. 11, all prediction errors fall below 0.3, with the average prediction errors for $k^i_{\Psi,1}$, $k^i_{\Psi,2}$, $\triangle_0$, and l_0 being 0.23, 0.16, 0.05, and 0.06, respectively. These results highlight that Mudi can adapt well to unobserved training tasks by leveraging the network architectures as prediction features.

We also assess the prediction accuracy of E2E latency as the number of training samples increases. When an inference service is co-located with a new training task, Mudi samples the inference latency under this new co-location to fit the corresponding piece-wise function. The parameters of this function are then leveraged to perform incremental training for updating the *Interference Predictor*. Fig. 12 presents the results for training sample sizes ranging from 30 to 90, showing that the prediction error decreases from up to 0.6 to below 0.16 for all inference services. These results highlight that Mudi can become significantly more efficient by incorporating new samples as new training tasks are introduced to the cluster.

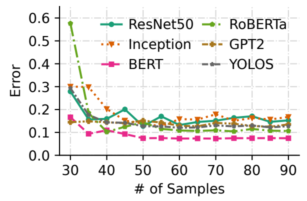
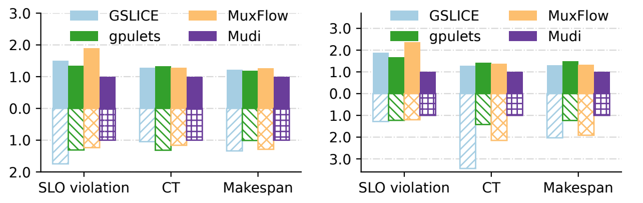


Figure 12. The prediction accuracy of inference latency increases as more samples are included during learning.



(a) Cluster-wide co-location (b) Batching/GPU% tuning

Figure 13. The benefits of individual optimization (solid /hollow bar represents the physical/simulated cluster).

Effectiveness of cluster-level co-location. To evaluate the advantage of cluster-wide multiplexing, we disabled the *Tuner* service under Mudi. In Fig. 13(a), we normalized the SLO violation rate, CT, and makespan to Mudi's values for comparison in the two clusters. The results demonstrate that Mudi can reduce the CT by up to 1.33× and the makespan by up to 1.26×, while achieving a lower SLO violation rate of 1.3%/2.8% in physical/simulated cluster. Although this cluster-wide optimization alone yields a higher SLO violation rate compared to the original two-level co-optimization (by 1.65×/2.43× in physical/simulated cluster), it still significantly outperforms baselines, with SLO violation rates reduced by up to 1.74× in large-scale simulated cluster. These findings highlight the effectiveness of Mudi's cluster-level co-location policy in selecting an optimal device for multiplexing, utilizing interference modeling for each inference.

Effectiveness of per device control. To assess the effectiveness of Mudi's batching/GPU% tuning methods, we replaced Mudi's cluster-wide multiplexing method with random task co-location. As shown in Fig. 13(b), Mudi achieves the lowest SLO violation rate at 1.03%, which is 1.1× higher than the original Mudi. This outstanding performance is attributed to Mudi's explicit quantification of inference latency. Furthermore, Mudi demonstrates superior performance in terms of CT and makespan for training tasks. Specifically, the CT and makespan are up to 3.44×, 2.03× lower than the baseline methods. These results indicate that Mudi's tuning mechanisms exhibit remarkable adaptability to dynamic requests and significantly enhance training efficiency.

7.4 System Robustness

System throughput. To measure the maximum throughput while maintaining SLOs for inference services across different systems, we conducted experiments where we gradually increased the QPS rate until the SLOs were no longer met. Simultaneously, training tasks were multiplexed and Mudi allocates a partition of at least 10% of the GPU to facilitate the seamless execution of these training tasks.

The maximum throughput that each system could support for each inference service is presented in Fig. 14. We observed that Mudi can make the most use of GPU resources to achieve the highest throughput of all inference services. Specifically, Mudi achieves an increase in maximum throughput by up to

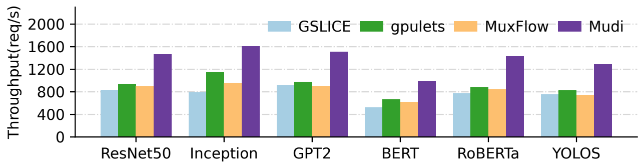
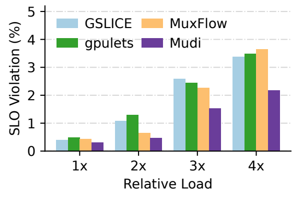
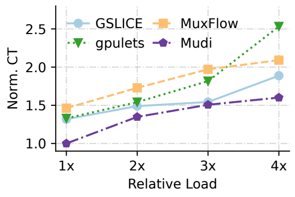


Figure 14. The maximum achievable throughput for each inference service while guaranteeing SLOs.



(a) SLO violation



(b) Norm. CT

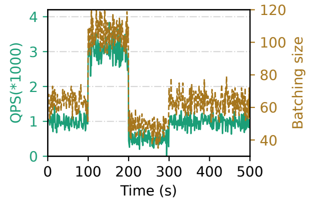
Figure 15. Sensitivity to various inference loads.

78%, 103%, 67%, 89%, 85%, and 73% for ResNet50, Inception, GPT2, BERT, RoBERTa, and YOLOS, respectively.

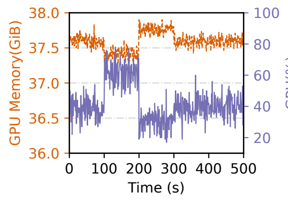
Sensitivity to heavy loads. To examine the response of Mudi under heavy loads, we increased the request arrival rates of all inference services by 2×, 3×, 4×. As shown in Fig. 15, all systems experience higher SLO violations and longer CTs for training tasks as QPS increases. However, Mudi consistently surpasses other baselines in terms of maintaining a lower SLO violation rate. Furthermore, as the QPS increases, Mudi demonstrates a slower escalation in its SLO violation rate compared to the baselines. These findings strongly support Mudi's superior ability to efficiently manage heavy workloads for inference services. Additionally, Fig. 15(b) reveals that Mudi exhibits a nonlinear increase in CT as the load intensifies, while gpulets and GSLCIE showcase a linear increase due to the lack of cluster-wide optimization. Consequently, co-located workloads experience higher resource interference with these baselines.

Ability to handle bursty QPS. Our observations demonstrate that Mudi is capable of handling bursty QPS rates while ensuring SLOs. The ability of Mudi to rapidly respond to bursty loads is exemplified through a case study using the inference service ResNet50 and the training task YOLOv5, as depicted in Fig. 16. At 100s, the QPS of ResNet50 momentarily bursts to 3×. In response, the *Tuner* adjusts the batching size (as shown in Fig. 16(a)) and GPU% to accommodate this change, resulting in a very low SLO violation rate of 0.71%. Additionally, due to GPU memory limitation, some memory of YOLOv5 is swapped from device to host, as depicted in Fig. 16(b). When the QPS decreases at 200s, the remaining GPU memory and SMs are reclaimed for training. This data swapping process incurs a minor transmission overhead between the host and the device, with the average transfer time for the training task YOLOv5 observed to be 23.31ms.

Furthermore, the *Memory Manager* effectively addresses the risk of GPU memory OOM errors as the batching size increases. This memory-swapping capability is demonstrated



(a) Batching size changes with QPS fluctuating of inference



(b) Memory swapping of training and GPU% tuning of inference

Figure 16. Mudi's behavior under bursty QPS.

Table 4. The fraction of time when memory swapping occurs

ResNet50	Inception	GPT2	BERT	RoBERTa	YOLOS
16.08%	19.82%	28.40%	15.53%	27.30%	33.43%

in Tab. 4 during bursty QPS scenarios. Notably, for ResNet50, the performance of the workload remains stable even when GPU memory usage exceeds capacity for approximately 16% of the time. The frequency of situations where capacity is exceeded varies across different inference services, depending on factors such as input and model sizes. For instance, YOLOS operates in an overcapacity state for over one-third of the time; however, it successfully handles inference requests without encountering any OOM errors.

Capability to handle more training tasks. We fitted additional 90 piecewise linear functions for co-location prediction and configuration tuning under the scenario of multiplexing more training tasks within a GPU. This enhanced implementation, referred to as *Mudi-more*, is compared with other two systems: 1) Random: which uses random placement strategy and evenly distributes GPU resources among all workloads, and 2) Mudi: which only allows multiplexing one inference and one training.

The results depicted in Fig. 17 demonstrate that *Mudi-more* outperforms the Random strategy across all evaluated metrics. However, *Mudi-more* records a mean SLO violation of 0.52%, which is 1.03× that of Mudi. Additionally, *Mudi-more* shows a modest increase in CT (1.07×) and makespan (1.09×) compared to Mudi. This is due to the increased number of training tasks sharing the same GPU, requiring Mudi to swap more GPU memory from the GPU to the host (37.78%, which is 1.61× that of just one training task) to maintain SLO compliance for inference requests. This activity delays the completion of training. Furthermore, the interference between co-located workloads is more pronounced with multiple training tasks, leading to extended CT and makespan. The average tuning/multiplexing overhead with *Mudi-more* (18 iterations/16ms) is marginally

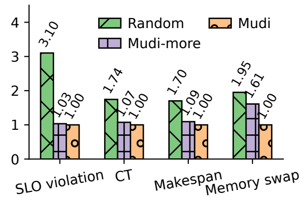
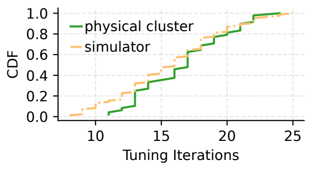
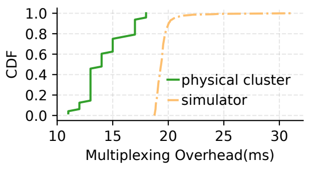


Figure 17. Mudi's behavior on multiplexing more training tasks in physical cluster.



(a) Tuning iterations



(b) Multiplexing overhead

Figure 18. The distribution of computational overhead.

greater (12.4%/14.3%) than Mudi in the physical cluster. These overheads become significantly more noticeable (18.8%/26.3%) in the large-scale simulated cluster, due to a more dynamic system environment and a larger search space of co-located patterns. Consequently, considering both overhead and performance, it is advisable to use Mudi for multiplexing one inference and one training task to achieve optimal performance.

7.5 System Overhead

Tuning overhead. We have measured the search overhead of GP-LCB. As illustrated in Fig. 18(a), GP-LCB converges within 17 iterations in over half the cases. The search process can be completed within a maximum of 24 iterations in a physical cluster. In the simulated cluster, all tuning processes require fewer than 25 iterations, averaging 16. These results suggest that Mudi can quickly determine an optimal batching size under dynamic QPS in less than 1.92 seconds, enhancing training throughput while meeting SLO constraints. Moreover, the simulated-based results also demonstrate that Mudi can scale well in large-scale clusters.

Multiplexing overhead. The multiplexing overhead in Mudi primarily comes from the *Online Prediction* and *Device Selector*. The results in Fig. 18(b) indicate that the overall time when making cluster-wide multiplexing decisions is below 18ms, with an average of 14ms in physical cluster. And the simulated-based results show this overhead is below 31ms with an average 19ms. These findings indicate that Mudi is capable of making real-time task assignments.

8 Related Work

Underlying multiplexing optimizations. To improve the performance of multiplexing at a lower level, several studies have investigated kernel-level control strategies [9, 65, 73, 80]. For example, AntMan [73] adjusts the timing of kernel launches to maintain performance for high-priority workloads. Pilotfish [80] and other research efforts [9, 65] focus on regulating the rate of kernel requests from low-priority workloads. However, these approaches do not address interference mitigation at a cluster-wide level.

Interference-aware co-location. Several studies have investigated various techniques, such as collaborative filtering [10, 11, 57], and BO [5, 50], to optimize multiplexing for traditional cloud applications. Other works delve into machine learning-based approaches to model co-location interference [7, 9, 43, 74]. Nevertheless, these works do not consider both dynamic batching and resource scaling, resulting in low resource efficiency. Additionally, some researchers propose kernel padding scheduling [18, 21, 80] to mitigate interference among all co-located DL applications. Nevertheless, implementing these techniques requires bespoke modifications to DL frameworks such as TensorFlow.

Adaptive batching and dynamic resource partitioning. Tuning the batching size or resource partition sizes adaptively to improve the performance of DL workloads has been extensively studied in previous works [7, 8, 14, 30, 52, 61, 68, 74, 76]. For instance, Pollux [52] optimizes the batchsize and number of GPU workers for training tasks to maximize *goodput*, without considering multiplexing. INFaaS [58] and INFless [76] can automatically select batchsizes and hardware settings based on users' SLO requirements in serverless computing. Morphling [68] employs meta-learning techniques to efficiently configure batchsizes, GPU timeshare, GPU memory, and GPU types for each inference service. iGniter [74] enhances GPU resource allocation by considering factors such as GPU L2 cache profiles and GPU power consumption among co-located applications. However, these systems do not optimize task co-locations across the entire cluster, which can easily result in sub-optimal performance.

9 Conclusion and Remarks

This paper presents Mudi, a novel multiplexing system that optimizes both cluster-wide co-location and device-level interference control. By efficiently multiplexing online inference services with DL training tasks across all GPU devices within a cluster, Mudi enhances resource utilization while guaranteeing inference SLOs and maximizing training task throughput. Mudi leverages the quantification of inference latency through a piece-wise linear function and a predictive approach to estimate interference based on deep network architectures. This allows Mudi to achieve global optimization with rapid response to highly dynamic workloads.

Acknowledgments

We thank the anonymous reviewers, and our shepherd, Dr. Deepak Narayanan, for their valuable feedback and suggestions. This work is supported in part by the National Key R&D Program of China (No. 2021YFB3300200), the Science and Technology Development Fund of Macau (0024/2022/A1, 0071/2023/ITP2, 0081/2022/A2, and 0123/2022/AFJ), the Multi-Year Research Grant of University of Macau (MYRG2022-00119-FST, MYRG-GRG2023-00019-FST-UMDF), National Natural Science Foundation of China (No. 62072451, 92267105), Guangdong Basic and Applied Basic Research Foundation (No. 2023B1515130002), Guangdong Special Support Plan (No. 2021TQ06X990), Shenzhen Basic Research Program (No. JCYJ20220818101610023, KJZD20230923113800001).

References

[1] 2023. Kubernetes: Production-grade container orchestration. https://kubernetes.io

[2] 2023. Kubernetes scheduling framework. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework

[3] 2023. REDDIT. https://chrsmrrs.github.io/datasets/docs/datasets

[4] Alibaba. 2023. GPU Traces. https://github.com/alibaba/clusterdata

[5] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of NSDI*.

[6] Wenyan Chen, Zizhao Mo, Huanle Xu, Kejiang Ye, and Chengzhong Xu. 2023. Interference-aware Multiplexing for Deep Learning in GPU Clusters: A Middleware Approach. In *Proceedings of SC*.

[7] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *Proceedings of ATC*.

[8] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System.. In *Proceedings of NSDI*.

[9] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2021. Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction. In *Proceedings of SC*.

[10] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. (2013).

[11] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. (2014).

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of CVPR*.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[14] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of SoCC*.

[15] Yuxin Fang, Bencheng Liao, Xinggang Wang, Jiemin Fang, Jiyang Qi, Rui Wu, Jianwei Niu, and Wenyu Liu. 2021. You only look at one sequence: Rethinking transformer in vision through object detection. In *Proceedings of NeurIPS*.

[16] Yangyang Feng, Minhui Xie, Zijie Tian, Shuo Wang, Youyou Lu, and Jiwu Shu. 2023. Mobius: Fine tuning large-scale models on commodity gpu servers. In *Proceedings of ASPLOS*.

[17] Peter I Frazier. 2018. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811* (2018).

[18] Guin Gilman, Samuel S Ogden, Tian Guo, and Robert J Walls. 2021. Demystifying the placement policies of the NVIDIA GPU thread block scheduler for concurrent kernels. In *Proceedings of SIGMETRICS*.

[19] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like clockwork: Performance predictability from the bottom up. In *Proceedings of OSDI*.

[20] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. 2022. Cocktail: A multidimensional optimization for model serving in cloud. In *Proceedings of NSDI*.

[21] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *Proceedings of OSDI*.

[22] F Maxwell Harper and Joseph A Konstan. 2015. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (TIIS)* 5, 4 (2015).

[23] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. 2015. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of ASPLOS*.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of CVPR*.

[25] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of WWW*.

[26] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[27] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of SC*.

[28] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. 2024. Characterization of large language model development in the datacenter. In *Proceedings of NSDI*.

[29] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2023. Lucid: A Non-intrusive, Scalable and Interpretable Scheduler for Deep Learning Training Jobs. In *Proceedings of ASPLOS*.

[30] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. 2021. Scrooge: A cost-effective deep learning inference system. In *Proceedings of SoCC*.

[31] Zhewei Huang, Wen Heng, and Shuchang Zhou. 2019. Learning to paint with model-based deep reinforcement learning. In *Proceedings of ICCV*.

[32] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360*.

[33] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of ATC*.

[34] G Jocher, A Stoken, J Borovec, A Chaurasia, L Changyu, V Abhiram, et al. 2021. Ultralytics/Yolov5: V5. 0—YOLOv5-P6 1280 Models, AWS, Supervise. *Ly and YouTube Integrations* 10 (2021).

[35] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. 2023. DeepUM: Tensor Migration and Prefetching in Unified Memory. In *Proceedings of ASPLOS*.

[36] Andreas Krause and Cheng Ong. 2011. Contextual gaussian process bandit optimization. In *Proceedings of NeurIPS*.

[37] Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. (2009).

[38] Shanghai AI Lab. 2024. GPU Traces. https://github.com/InternLM/AcmeTrace

[39] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. MISO: exploiting multi-instance GPU capability on multi-tenant GPU clusters. In *Proceedings of SoCC*.

[40] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic Scheduling for Deep Learning Clusters. In *Proceedings of EuroSys*.

[41] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *Proceedings of ICCV*.

[42] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[43] Daniel Mendoza, Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. Interference-aware scheduling for inference serving. In *Proceedings of EuroMLSys*.

[44] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).
[45] Microsoft. 2019. Philly Traces. https://github.com/msr-fiddle/philly-traces
[46] NVIDIA. 2022. Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html
[47] NVIDIA. 2023. CUDA Unified Memory. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd
[48] NVIDIA. 2023. Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu
[49] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[50] Tirthak Patel and Devesh Tiwari. 2020. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *Proceedings of HPCA*.
[51] Pytorch. 2023. Word-level language modeling rnn. https://github.com/pytorch/examples/tree/master/word_language_model
[52] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *Proceedings of OSDI*.
[53] QinghaoHu, ZhishengYe, MengZhang, QiaolingChen, PengSun, YonggangWen, and TianweiZhang. 2023. Hydro: Surrogate-Based Hyperparameter Tuning Service in Datacenters. In *Proceedings of OSDI*.
[54] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
[55] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).
[56] Carl Edward Rasmussen. 2003. Gaussian processes in machine learning. In *Summer school on machine learning*. Springer, 63–71.
[57] Francisco Romero and Christina Delimitrou. 2018. Mage: Online and interference-aware scheduling for multi-scale heterogeneous systems. In *Proceedings of PACT*.
[58] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving.. In *Proceedings of ATC*.
[59] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. 2011. Finding a "kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st international conference on distributed computing systems workshops*. IEEE, 166–171.
[60] SenseTime. 2021. Helios Traces. https://github.com/S-Lab-System-Group/HeliosData
[61] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of SOSP*.
[62] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[63] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. 2009. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995* (2009).
[64] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of EuroSys*.
[65] Qingxiao Sun, Yi Liu, Hailong Yang, Ruizhe Zhang, Ming Dun, Mingzhen Li, Xiaoyan Liu, Wencong Xiao, Yong Li, Zhongzhi Luan, et al. 2022. CoGNN: efficient scheduling for concurrent GNN training on GPUs. In *Proceedings of SC*.
[66] Susheel Suresh, Pan Li, Cong Hao, and Jennifer Neville. 2021. Adversarial Graph Augmentation to Improve Graph Contrastive Learning. In *Proceedings of NeurIPS*.
[67] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of AAAI*.
[68] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. 2021. Morphling: fast, near-optimal auto-configuration for cloud-native model serving. In *Proceedings of SoCC*.
[69] Sheng-Yu Wang, David Bau, and Jun-Yan Zhu. 2021. Sketch your own gan. In *Proceedings of ICCV*.
[70] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *Proceedings of NSDI*.
[71] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. 2023. Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent. In *Proceedings of ATC*.
[72] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *Proceedings of NSDI*.
[73] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic scaling on GPU clusters for deep learning. In *Proceedings of OSDI*.
[74] Fei Xu, Jianian Xu, Jiabin Chen, Li Chen, Ruitao Shang, Zhi Zhou, and Fangming Liu. 2022. igniter: Interference-aware gpu resource provisioning for predictable dnn inference in the cloud. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2022), 812–827.
[75] Xin Xu, Na Zhang, Michael Cui, Jiayuan He, and Ridhi Surana. 2019. Characterization and Prediction of Performance Interference on Mediated Passthrough GPUs for Interference-aware Scheduler. In *Proceedings of HotCloud*.
[76] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of ASPLOS*.
[77] Gingfung Yeung, Damian Borowiec, Adrian Friday, Richard Harper, and Peter Garraghan. 2020. Towards GPU Utilization Prediction for Cloud Deep Learning. In *Proceedings of HotCloud*.
[78] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. 2023. Cocktailer: Analyzing and Optimizing Dynamic Control Flow in Deep Learning. In *Proceedings of OSDI*.
[79] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving.. In *Proceedings of ATC*.
[80] Wei Zhang, Binghao Chen, Zhenhua Han, Quan Chen, Peng Cheng, Fan Yang, Ran Shu, Yuqing Yang, and Minyi Guo. 2022. PilotFish: Harvesting Free Cycles of Cloud Gaming with Deep Learning Training. In *Proceedings of ATC*.
[81] Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, and Minyi Guo. 2022. Astraea: towards QoS-aware and resource-efficient multi-stage GPU services. In *Proceedings of ASPLOS*.
[82] Yihao Zhao, Xin Liu, Shufan Liu, Xiang Li, Yibo Zhu, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. MuxFlow: Efficient and Safe GPU Sharing in Large-Scale Production Deep Learning Clusters. arXiv:2303.13803 [cs.DC]
[83] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately estimating the efficacy of optimizations for DNN training. In *Proceedings of ATC*.
[84] Zheng Zhu, Guan Huang, Jiankang Deng, Yun Ye, Junjie Huang, Xinze Chen, Jiagang Zhu, Tian Yang, Jiwen Lu, Dalong Du, et al. 2021. Webface260m: A benchmark unveiling the power of million-scale deep face recognition. In *Proceedings of CVPR*.