Future Generation Computer Systems 181 (2026) 108413

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Prediction-based GPU sharing for distributed training

Changyong Shin [a], Younghun Go [a], Yeonho Yoo [b], Jinwoo Jeong [a], Jaehyun Hwang [c,*], Gyeongsik Yang [a,*], Chuck Yoo [a,*]

[a] Department of Computer Science and Engineering, Korea University, 145 Anam-ro, Seoul, 02841, Republic of Korea
[b] Department of Computer Science and Artificial Intelligence, Dongguk University, 30, Pildong-ro 1-gil, Jung-gu, Seoul, 04620, Republic of Korea
[c] Department of Electrical and Computer Engineering, Sungkyunkwan University, 2066 Seobu-ro, Suwon, 16419, Gyeonggi-do, Republic of Korea

ARTICLE INFO

Keywords:
Cloud computing
GPU Sharing
Service level agreement
Performance prediction
GPU Scheduling

ABSTRACT

GPU sharing aims to enhance the efficiency of GPU utilization by running distributed deep learning training jobs concurrently. However, GPU sharing poses a significant challenge: the increase in job completion time (JCT) caused by interference between jobs is inconsistent, complicating job scheduling. Our experiments reveal that the degree of JCT increase varies by as much as ~3.7×. While previous studies have analyzed this JCT inconsistency problem, none of them have been able to minimize the inconsistency. We propose TensorShare, a proactive GPU sharing technique that leverages a deep learning model to predict the extent of JCT increase. This study defines a new metric, called GPU SLA, which represents the upper threshold of JCT increase. TensorShare then introduces a novel scheduler that proactively identifies which jobs meet GPU SLA while minimizing the JCT increase. Our evaluation shows that TensorShare improves GPU SLA satisfaction rates by 26.1×–47.3× and reduces the JCT increase by 37%–60%. Furthermore, we evaluate TensorShare with large language models that are not included in training TensorShare's prediction model, achieving ~7× and ~10.3× improvements in GPU SLA satisfaction and JCT inconsistency, respectively.

1. Introduction

Distributed training (DT) of deep learning models has become the dominant workload for GPU clouds across various domains, including large language models (LLMs) such as ChatGPT, autonomous driving, and others [1–4]. Hoping to consume less GPU and achieve less training time, "GPU sharing" has been proposed to multiplex GPUs to execute multiple DT jobs [5–9]. However, it is widely known that GPU sharing increases job completion time (JCT) compared to "dedicated use," where GPU is exclusively allocated to execute each DT job. This is due to interference between jobs when the GPU is shared.

Moreover, when DT jobs share GPU, their JCT values become highly inconsistent. Our motivating experiments show that the JCT of a DT job can vary by ~3.7× depending on how the GPUs are shared. This leaves end-users puzzled about why their jobs behave differently. Stable and predictable performance is an essential property of cloud services [10–12]. Consequently, JCT inconsistency is recognized as a significant drawback of GPU sharing, hindering its practical use in real-world GPU clouds [7,9]. As a result, many existing GPU clouds operate with dedicated use, leading to poor GPU utilization, often falling below 50% [8,13].

The issue of JCT inconsistency has only recently gained attention in the literature. Recent studies, such as [5,7,9], have reported on the degree of interference between jobs on GPUs. They measure GPU utilization and GPU memory utilization of DT jobs. They found that when GPU utilization is high, DT jobs experience significant interference, resulting in inconsistent JCT. However, they did not address reducing interference or improving JCT inconsistency. Other studies [14–16] monitor the training speed or GPU throughput of DT jobs, pausing jobs via preemption when throughput decreases. Yet, the JCT of the jobs remains quite inconsistent in these studies.

To tackle the JCT inconsistency problem, this study introduces the concept of "GPU service level agreement" (gSLA) for GPU sharing. We define the gSLA as the "upper threshold" of the JCT ratio between GPU sharing and dedicated use.[1] As an illustration, if the gSLA is set to 2, it indicates that end-users can tolerate up to 2× increase in JCT compared to dedicated use. We assume that the gSLA is determined by end-users,

[1] DT jobs operate under various completion conditions: 1) a fixed number of iterations, 2) unlimited iterations until the model converges, or 3) customized conditions like early stopping. By defining the gSLA as a ratio of JCT, we believe our gSLA definition can accommodate these diverse completion conditions.

* Corresponding authors.
E-mail addresses: cyshin@os.korea.ac.kr (C. Shin), yhgo@os.korea.ac.kr (Y. Go), yhyoo@dgu.ac.kr (Y. Yoo), jwjeong@os.korea.ac.kr (J. Jeong), jh.hwang@skku.edu (J. Hwang), g_yang@korea.ac.kr (G. Yang), chuckyoo@os.korea.ac.kr (C. Yoo).

https://doi.org/10.1016/j.future.2026.108413
Received 6 March 2025; Received in revised form 21 January 2026; Accepted 31 January 2026
Available online 3 February 2026

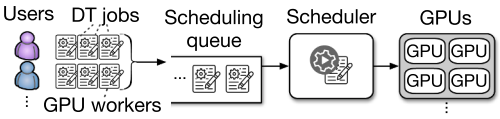


Fig. 1. GPU cloud and DT jobs.

which is a reasonable assumption since they can decide how long they are willing to wait. Using the gSLA, we transform the problem of improving inconsistent JCT into the problem of meeting the gSLA-specified requirements. To the best of our knowledge, this is a first study to address JCT inconsistency in the context of GPU sharing.

With the gSLA, we present a new GPU sharing technique, called TensorShare, which incorporates two innovative techniques. First, TensorShare predicts the JCT increase ratio of DT jobs, denoted as δ, using a new machine learning model called the δ predictor. The δ predictor is designed through: 1) profiling fine-grained GPU metrics of DT jobs (dataset), 2) in-depth feature engineering, and 3) neural architecture search (§3.2). This design is inspired by recent studies that predict resource consumption for deep learning jobs [17–19]. Note that the predictive capabilities of TensorShare make it proactive, as it operates before executing the jobs. Second, we propose a novel scheduler to select an appropriate pair (combination) of DT jobs for GPU sharing while satisfying the gSLA. Using the δ predictor, the TensorShare scheduler checks the jobs in the scheduling queue and filters out those whose δ values exceed their gSLAs. It then selects the optimal combination that improves the JCT inconsistency for individual DT jobs (§3.3).

We implement TensorShare from scratch and extensively evaluate it with state-of-the-art (SOTA) techniques for gSLA satisfaction. Our evaluation includes experiments on our physical testbed and large-scale simulations using real-world GPU cloud traces containing over 10K DT jobs. To further demonstrate its robustness, we evaluate TensorShare on unseen DT jobs, particularly using LLMs such as OpenLLaMA [20]. We choose LLMs because they have significantly larger model structures compared to existing deep learning models [21]. For instance, OpenLLaMA-7B has a parameter size that is 50.7× larger than that of VGG-16 (a model used for training the δ predictor). In addition, LLMs have complex layers, such as multi-head attention, which require more computation and training time than other deep learning models [22]. By applying TensorShare to these unseen LLM jobs, we demonstrate its ability to generalize the δ predictor for model types not included in the δ predictor training.

The key contributions of this study are as follows:

- Formulate the inconsistent JCT problem for the first time in terms of gSLA.
- Design a δ prediction model with neural architecture search.
- Devise a novel job scheduler for GPU sharing.
- Achieve ∼47.3× better gSLA satisfaction and ∼50× reduced gSLA excess ratio compared to existing techniques.
- Improve the JCT of individual jobs and GPU infrastructure efficiency (GPU time) by ∼60% and ∼44%, respectively, compared to existing techniques.
- Exhibit the effectiveness of TensorShare on unseen LLM jobs, improving gSLA satisfaction and JCT inconsistency by ∼7× and ∼10.3×, respectively.

2. Background and motivating experiment

2.1. Background

DT accelerates model training through multiple GPU workers and parameter servers (PSs),[2] with a worker denoting an entity that performs model training in a distributed manner. To maintain model consistency, GPU workers must synchronize through PSs by exchanging gradients and updated parameters, which necessitates inter-GPU or inter-node communication [19,23]. Typically, DT runs on GPU clouds (e.g., Google Cloud Platform, Microsoft Azure, and Amazon Web Services). Fig. 1 illustrates an example of GPU clouds and their workflow, where multiple users submit DT jobs, specifying the number of GPU workers required for each job [13]. These submitted jobs are added to a "scheduling queue," and the cloud scheduler assigns them to GPUs for execution.

The allocation of GPUs to DT jobs is managed by the cloud scheduler, which employs one of two approaches: (1) *dedicated use* or (2) *GPU sharing*. In "dedicated use", GPUs are allocated exclusively to DT jobs without sharing, a strategy used by most public GPU clouds. Conversely, "GPU sharing" allows multiple DT jobs to use GPUs concurrently. We present the challenges of GPU sharing through motivating experiments in the following subsection.

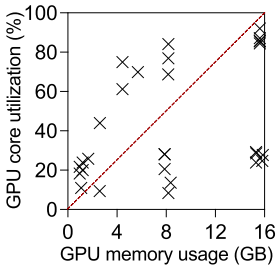


Fig. 2. Resource usage distribution of DT jobs.

Table 1
Datasets and models used for experiments.

Category	Datasets	Models
Image classification	CIFAR-10	DenseNet-40-12, DenseNet-100-12, DenseNet-100-24, ResNet-20, ResNet-32, ResNet-44, ResNet-56, ResNet-110, AlexNet, NASNet
	ImageNet	ResNet-50, ResNet-101, ResNet-152, AlexNet, OverFeat, GoogleNet, InceptionV3, InceptionV4, VGG-11, VGG-16, VGG-19, NASNet, MobileNet
NLP	Europarl	NMTBig, NMTMedium, NMTSmall, Transformer, TransformerANN, TransformerBig

2.2. Motivating experiment

Experimental setup. In this section, we use one GPU machine and one storage machine connected via a 10 GbE switch. The GPU machine has two NVIDIA V100 GPUs, two Intel Xeon Silver 4210 CPUs, and 128 GB of RAM. The storage machine has two Intel Xeon Silver 4210 CPUs, 128 GB of RAM, and a 2 TB SSD. We also evaluate configurations with multiple GPU machines and conduct large-scale simulations in §4.

We use the Kubernetes plugin [24], which enables the Kubernetes scheduler to support GPU sharing. This is the SOTA GPU sharing technique used in real-world clouds [8].[3] We evaluate other SOTA techniques in §4. In our experiments, the Kubernetes plugin places a maximum of two GPU workers on a single GPU, as placing three workers leads to significantly longer JCT—a result also observed in other studies [5,9] (see §5 for more details). The storage machine stores training datasets and computational states of paused models due to preemption (e.g., model parameters). The storage and GPU machines are connected via a network file system.

[2] We focus on data parallel with PS strategy, but we believe our design can be applied to other strategies (please refer to §5).

[3] We recognize the GPU sharing features provided by NVIDIA GPUs, such as MPS and MIG. However, these technologies are known to have significant limitations in used in real-world cloud environments [25,26]. Thus, we exclude them from our evaluation (details in §5).

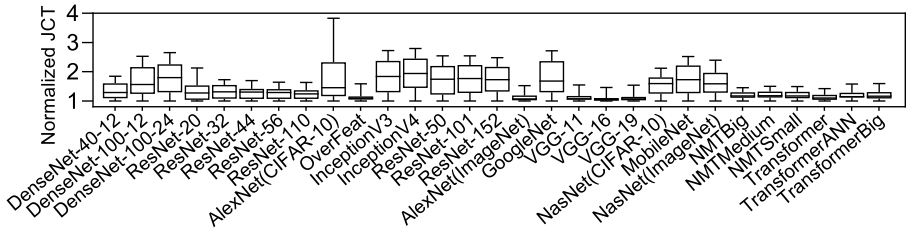


Fig. 3. JCT distribution. Whiskers are the JCT distribution for each job (x-axis) under different GPU sharing jobs.

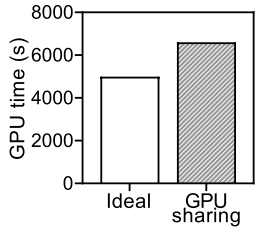


Fig. 4. GPU time.

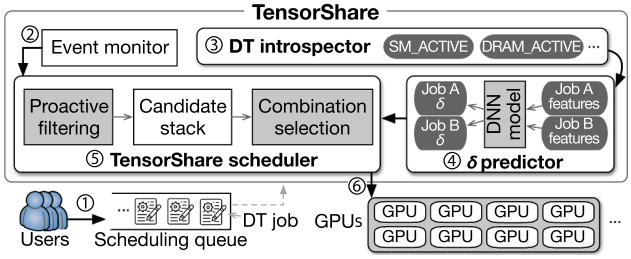


Fig. 5. TensorShare workflow.

As DT jobs, we use representative image classification and natural language processing (NLP) models, as summarized in Table 1. The datasets include CIFAR-10 and ImageNet for image classification models and Europarl for NLP models, resulting in 29 different types of DT jobs. Note that the publicly disclosed workloads in production traces such as Microsoft Philly [13] and Alibaba-PAI [8] include 10 and eight types of workloads, respectively. In contrast, our evaluation encompasses 29 distinct workload types.

In terms of resource usage, our job distribution is designed to be comprehensive. To show this, we execute all the workloads in Table 1 on a V100 16GB GPU without GPU sharing and profile their GPU core utilization and GPU memory usage. In Fig. 2, we plot each workload's resource usage, where GPU memory is x-axis and GPU core utilization is y-axis. It shows that the workloads are broadly distributed across both axes: GPU core utilization ranges from 0% to 100%, and memory usage spans from 0 to 16 GB (the maximum capacity of the V100 16GB GPU). This wide distribution demonstrates that the job types consist of a good mix of memory-intensive and compute-intensive models.

Each DT job undergoes over 700 training iterations, providing sufficient time to measure JCT inconsistency. We use a batch size of 64 for image classification models and 128 for NLP models. To run the DT jobs, TensorFlow 1.13, OpenNMT-tf 1.25.3, Python 3.7, CUDA 11.4, and the NVIDIA GPU driver 470.182.03 are used.

We measure the following items.

- JCT consistency: The distribution of JCT, which is the time required to complete a DT job when another job is running concurrently.
- GPU time: The total sum of each GPU's end-to-end usage duration required to complete the training of all enqueued DT jobs. This metric indicates GPU infrastructure efficiency and is commonly used in other studies [27,28].

JCT consistency. We evaluate JCT consistency in GPU sharing as follows. For each experiment trial, we select a pair (combination) of jobs to run using GPU sharing. For example, if job A (J_A) and job B (J_B), each with two GPU workers, are selected as a combination, we run one worker from both J_A and J_B on a single GPU. Consequently, the two GPUs in our GPU machine run four workers from J_A and J_B. In the experiments, we fix J_A (represented on the x-axis of Fig. 3) while alternating J_B to run alongside J_A. Fig. 3 shows the JCT distribution, where the x-axis represents the 29 job types listed in Table 1 (as J_A), and the y-axis indicates the range of JCT values for J_A when run alongside the 29 job types (as J_B). The total number of job combinations in Fig. 3 is 435.[4] As the JCT varies between DT jobs, we normalize the JCT of each job by its JCT when running without GPU sharing (dedicated use). So, a value of 2 on the y-axis indicates that the JCT of the job on the x-axis under GPU sharing is twice as long as its JCT with dedicated use. This normalized JCT relative to dedicated use is denoted as δ. Each job on the x-axis has 29 corresponding δ values, and each whisker represents the distribution of these 29 δ values.

In Fig. 3, we observe that JCT is highly inconsistent. On average, Kubernetes GPU sharing exhibits a δ of ~1.5× (with the maximum of ~3.7× observed for AlexNet/CIFAR-10). The reason for this inconsistency is the different combinations of DT jobs. Even though the jobs come from the same set (Table 1), the combinations of jobs running together vary due to GPU scheduling. As a result, the interference between jobs changes depending on which jobs are scheduled to share the GPU, as demonstrated in Fig. 3. This experiment shows the necessity of proper scheduling to address JCT inconsistency.

GPU time. We measure GPU time, which represents the end-to-end time required to run 435 DT job combinations, as shown in Fig. 3. In the experiments, we compare the GPU time of GPU sharing with that of an "ideal" scenario. Since two GPU workers share a single GPU, the ideal GPU time for GPU sharing techniques is assumed to be half of the dedicated use, assuming that two identical jobs run without interference. We measure the GPU time for dedicated use and calculate the ideal GPU time as half of that value. Fig. 4 shows the GPU time for both the ideal scenario and GPU sharing (using the Kubernetes plugin). Compared to the ideal, GPU sharing results in 1.32× longer GPU usage. This indicates that GPU sharing introduces additional overhead, thereby reducing efficiency.

3. Tensorshare design

Fig. 5 illustrates the design of TensorShare. User-submitted DT jobs are enqueued in the scheduling queue (① in Fig. 5). The "TensorShare scheduler" determines which DT jobs to share and assigns them to the GPU machines. Also, TensorShare includes an "Event monitor" to track changes in job states and GPU states, activating the TensorShare scheduler as needed. For example, when a DT job completes its training, the

[4] From 29 different jobs, we can create 406 unique combinations of two different jobs ($29 \times 28/2$). Also, there are 29 combinations of two identical DT jobs. Therefore, the total number of possible combinations is 435.

event monitor triggers the TensorShare scheduler (②) to select a new DT job to run.

To decide which jobs to use the GPUs, the TensorShare scheduler requires the δ values for job combinations. These δ values are obtained from the "δ predictor", as follows: To predict δ, we *introspect*[5] each job in the scheduling queue to extract its features (e.g., Job A features in Fig. 5) using the "DT introspector" (③, details in §3.1). The job features include metrics such as SM_ACTIVE and DRAM_ACTIVE, which represent the active duration of the GPU core and memory. Based on these job features, the δ predictor predicts δ values for a combination of jobs (§3.2). For example, in Fig. 5, the δ predictor takes features of Job A and Job B to predict the δ of the two (④). The prediction is done for all job combinations in the scheduling queue.

Based on the predicted δ values, the TensorShare scheduler selects a job combination to run on GPUs (⑤) that (1) satisfies the gSLAs of the jobs and (2) minimizes JCT and GPU time (§3.3). Finally, the selected job combination is executed (⑥). The following subsections provide a detailed explanation of each TensorShare component.

3.1. DT introspector

The DT introspector profiles the GPU-related metrics of DT jobs. The profiling is conducted with dedicated use because individual jobs cannot be reliably characterized under GPU sharing, as DT jobs interfere with each other, causing high variations in GPU metrics. Also, we profile a single worker for each DT job since all workers of a job perform identical calculations in data parallelism, yielding identical values for our GPU metrics.

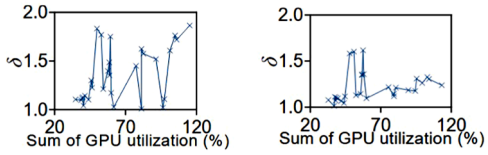
Now we discuss which GPU metrics should be profiled by our DT introspector. Previous studies have recognized the existence of interference when GPUs are shared and have profiled various metrics to analyze this interference. For example, Horus [7] and Lucid [9] use GPU utilization as a metric, which is defined as the ratio of the active duration during which calculations are executed. These studies conclude that when the average or total GPU utilization of DT jobs is high, interference between jobs becomes more severe. In other words, they indirectly analyze interference using GPU utilization [14,29]. To verify this conclusion, we conduct an experiment with two job combinations, shown in Fig. 6a. We measure each combination's GPU utilization and δ and denote the two combinations as c_A and c_B, respectively. The DT jobs in c_A show an average GPU utilization of 15%, while the result for c_B is 42%, which is 2.8× higher than c_A. Based on previous studies, δ is expected to be higher in c_B due to its higher GPU utilization. However, our results show that δ in c_A is actually 19% higher than in c_B, on average (Fig. 6a). This experiment demonstrates that GPU utilization alone has critical limitations in analyzing and predicting δ. Therefore, δ cannot be accurately predicted based solely on GPU utilization.

We present another example using job combinations to illustrate the limitation of other studies that use the sum of GPU utilization to estimate interference. Out of the 435 job combinations ($\{J_A, J_B\}$) from the experiments in Fig. 3, we fix J_A as DenseNet-40-12. Since J_B varies across 29 different DT jobs, this results in 29 unique combinations from the total of 435. For each of these combinations, we compute the total GPU utilization of both DT jobs, J_A and J_B. Note that the GPU utilization for each job is profiled under dedicated use. We then obtain 29 summed GPU utilization values. In Fig. 6b, the x-axis represents these 29 GPU utilization values in ascending order, while the y-axis displays the δ of DenseNet-40-12. Similarly, in Fig. 6c, we present the δ of TransformerBig as the summed GPU utilization increases. Contrary to the assumptions of previous studies, δ values (y-axis) do not increase linearly as GPU utilization (x-axis) increases in either Fig. 6b or Fig. 6c. These figures highlight the limitations of the summed GPU utilization as a predictor of δ. Note that this example is to highlight why a fine-grained approach is necessary. Our δ predictor does not use the sum of GPU utilization (§3.2).

[5] For new jobs in the queue, we perform a few training iterations (e.g., five) to obtain input features. We refer to this operation as "introspect."

	DT jobs	GPU util.	δ
c_A	ResNet-20	11%	1.8
	NMTSmall	19%	1.3
c_B	NasNet	61%	1.5
	VGG-16	23%	1.1

(a) Two job combinations.



(b) DenseNet-40-12. (c) TransformerBig.

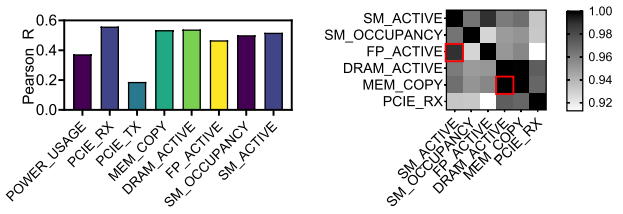
Fig. 6. Limitation of GPU utilization.

Table 2
Fine-grained GPU metrics in job profiling.

Metric	Physical meaning	Unit
SM_ACTIVE	Ratio of active durations in SMs.	%
SM_OCCUPANCY	Ratio of the actively occupied amount of SMs.	%
FP_ACTIVE	Ratio of active durations of 16- and 32-bit floating point commands in SMs.	%
DRAM_ACTIVE	Ratio of active durations of GPU memory interface.	%
MEM_COPY	Ratio of active durations for data read or write from to GPU memory.	%
PCIE_TX	Amount of communication transmitted by GPU over PCIe bus.	Byte/s
PCIE_RX	Amount of communication received by GPU over PCIe bus.	Byte/s
POWER_USAGE	Amount of GPU power consumption.	W

We further investigate the limitations of GPU utilization itself through an in-depth analysis—specifically, why GPU utilization is inaccurate for predicting δ. A single GPU consists of numerous cores (e.g., streaming multiprocessors, SM), and the extent and duration of SM usage vary significantly across DT jobs. However, GPU utilization does not distinguish how individual SMs are utilized; instead, it only represents the ratio of the active duration during which the "GPU device" is in use for a DT job. For example, even if only a single SM is active while thousands of others remain idle, GPU utilization still registers as 100% because the GPU device itself is engaged. As a result, GPU utilization fails to capture the critical variations in computation across SM cores, making it an inadequate metric for analyzing DT job behavior. A study from Microsoft also reported that GPU utilization is an inaccurate and coarse-grained metric [30].

Therefore, we explore alternative fine-grained metrics that are more accurate than GPU utilization measured using NVIDIA's official tool, DCGM [31]. We group the metrics into three categories: (1) GPU memory access (e.g., model and data batch loading), (2) GPU computation, and (3) inter-GPU communication (synchronization) [19,23]. To capture the physical resource interference, we select eight fine-grained metrics: DRAM_ACTIVE and MEM_COPY for memory access; SM_ACTIVE, SM_OCCUPANCY, FP_ACTIVE, and POWER_USAGE for computation; and PCIE_TX and PCIE_RX for communication. Table 2 lists these metrics along with their physical meanings and units. We measure all metrics at 100 ms intervals. Among the eight selected metrics, we identify four metrics—SM_ACTIVE, SM_OCCUPANCY, DRAM_ACTIVE, and PCIE_RX—as particularly effective for δ prediction, which we will discuss in detail in §3.2.



(a) Correlation between fine-grained GPU metric and δ.

(b) Cosine similarity between fine-grained GPU metrics. Darker cells indicate higher similarity.

Fig. 7. Input feature selection.

3.2. δ predictor

We now describe the details of the δ predictor. First, we determine the input features from the fine-grained GPU metrics. Second, we construct the dataset with preprocessing. Finally, we explore the architecture of the δ predictor and conduct a neural architecture search (NAS) to train it.

Input features. We identify input features through feature engineering, which involves: (1) correlation analysis between input features and output features (δ) and (2) similarity analysis between input features. We begin feature engineering with the eight metrics listed in Table 2.

To conduct these analyses, we prepare a dataset consisting of: (1) fine-grained GPU metrics of DT jobs and (2) δ values (output feature). Recently, public cloud providers have released traces on DT workloads and job logs (e.g., the number of GPU workers) [8,13,27,32]. However, to our knowledge, no datasets exist for fine-grained GPU metrics in DT jobs. Therefore, we construct our own dataset as follows. We introspect the 29 DT jobs described in §2.2 to collect fine-grained GPU metrics. Also, we use δ values measured in Fig. 3, which includes 435 job combinations. Since each combination consists of two jobs, we collect data from 870 DT job instances, each producing a corresponding δ value. For each δ, we match the fine-grained GPU metrics of the corresponding DT job, resulting in a dataset with 870 records.

With this dataset, we analyze the relationship between the fine-grained GPU metrics and interference. Specifically, we conduct a correlation analysis between each metric and δ by calculating the Pearson correlation coefficient, where a high value indicates a strong correlation [33,34]. Fig. 7a presents the Pearson coefficients of fine-grained GPU metrics with respect to δ. For computation metrics, SM_ACTIVE, SM_OCCUPANCY, and FP_ACTIVE show high correlation coefficients of 0.52, 0.5, and 0.47, respectively, while POWER_USAGE shows a coefficient of 0.37, which is below 0.4, indicating a weak relationship with δ. For memory, both DRAM_ACTIVE and MEM_COPY show high coefficient values of 0.54 and 0.54, respectively. For communication, PCIE_RX shows a high coefficient value of 0.56, while PCIE_TX shows a value of 0.19. Consequently, we exclude PCIE_TX and POWER_USAGE from the input features as they exhibit low correlations with δ.

Next, we analyze the similarity between the remaining six metrics using cosine similarity, which quantifies the similarity between two metrics in a heatmap. Fig. 7b presents the results: darker cells indicate a higher similarity between metrics. The red-line boxes in Fig. 7b highlight the high similarity between metric pairs. For example, FP_ACTIVE is highly similar to SM_ACTIVE (with a similarity of 0.991), and MEM_COPY is highly similar to DRAM_ACTIVE (0.9994). Previous studies have reported that highly similar input features increase training time and decrease prediction accuracy [35]. Thus, we exclude FP_ACTIVE and MEM_COPY to improve model efficiency. After filtering based on correlation and similarity analysis, we retain the following four metrics as input features for the δ predictor: SM_ACTIVE, SM_OCCUPANCY, DRAM_ACTIVE, and PCIE_RX.

Preprocessing. With the selected input features, we construct the dataset through the following preprocessing steps: (1) normalization, (2) temporal aggregation, and (3) dataset shuffling and splitting. First, we apply min-max normalization to the collected records because input features (e.g., SM_ACTIVE and PCIE_RX) have different units, such as % and bytes/s, as shown in Table 2. Second, we conduct temporal aggregation by sampling the collected input features at 100 ms intervals, followed by averaging them to produce a single value per input feature. Note that output features are calculated per job and thus require no temporal aggregation. Third, before training the δ predictor, we randomly shuffle the records and split the dataset into three subsets: 1) the dataset for training the model, 2) the dataset for validation, and 3) the dataset for testing (evaluation) with 6:2:2 ratio.

For label imbalance, we analyze whether the measured slowdown values (ground truth) in our dataset are balanced. We compute the Gini coefficient, a widely recognized metric for quantifying distributional imbalance. A lower Gini coefficient indicates more balanced distribution, and values below 0.3 are generally considered well-balanced. Our analysis shows 0.167, indicating a low degree of imbalance in the δ value distribution. This result provides strong evidence that the δ values in our dataset are not imbalanced.

Model structure and training. To design the predictor, we explore its structure via NAS, implemented using AutoKeras [36], a widely used tool for NAS. The NAS process navigates various model structures and hyperparameters, including the number of layers, activation functions, optimizers, batch size, learning rate, dropout rate, and batch normalization. Specifically, our NAS process varies the number of layers from one to 19. For each layer, NAS randomly selects hyperparameters. We adopt a Bayesian strategy, which leverages past training results and prediction accuracies of searched models to increase the likelihood of finding an optimal model in subsequent searches. To prevent overfitting and reduce unnecessary computation, we design NAS to limit the number of models explored and apply early stopping during the search. Specifically, NAS explores up to 100 models for each layer, resulting in up to 1900 models. For early stopping, we monitor the prediction error from the model validation and terminate the model search if no improvement is observed for 50 consecutive epochs. Consequently, for each layer, the model search is terminated when either 100 models have been explored or the early stopping condition is met.

We apply our NAS to DNN with multi-layer perceptron, convolutional neural network (CNN), and long short-term memory (LSTM). All accuracy in this paper is measured using symmetric mean absolute percentage error (SMAPE) [37], which is commonly used in related studies [38], unless stated otherwise. Table 3 summarizes the results of the respective accuracy, the number of searched models, and the corresponding NAS training time for CNN, LSTM, and DNN. Among the three algorithms, DNN achieves the best accuracy and shows the shortest training time. Based on these results, we select the MLP-based DNN as the final algorithm, as it outperforms the other deep learning algorithms in both accuracy and efficiency.

Specifically, the best accuracy is achieved with 10 MLP layers, and we choose this model as the δ predictor. Table 4 summarizes the detailed structure of the δ predictor, including each layer's weight, bias, and number of parameters. The δ predictor takes eight input features from the DT jobs, which corresponds to the input dimension of the first layer (MLP_1). Also, since the δ predictor predicts two δ values (one for each job), the output dimension of the final layer (regression_head) is 2. The intermediate dimensions and biases of each layer are determined during the NAS described previously. In total, the model has approximately 110K trainable weights and biases. Each layer employs the ReLU activation function, and the Adam optimizer is used. For hyperparameters, the model is trained with batch size of 128 and learning rate of 0.001, while dropout and batch normalization are disabled, all as determined by NAS. The detailed accuracy and training time of the DNN which is NAS results are reported in §4.4.

Table 3
Comparison of NAS-optimized algorithms.

Algorithm	Best accuracy (SMAPE)	Number of searched models	NAS training time (h)
CNN	16.7%	1747	11.5
LSTM	22.7%	1742	59.2
DNN	4.8%	1748	10.4

Table 4
Layer dimensions of the δ predictor in detail.

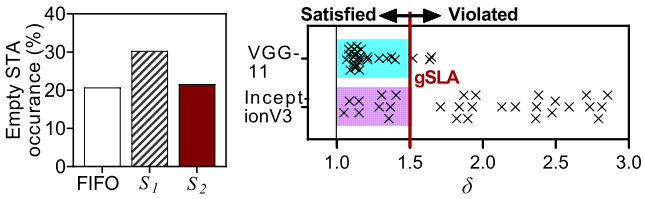
Layers	Weight (in × out)	Bias	Params
MLP_1	8×256	256	2304
MLP_2	256×32	32	8224
MLP_3–7	32×32	32	1056
MLP_8	32×1024	1024	33,792
MLP_9	1024×64	64	65,600
MLP_10	64×16	16	1040
regression_head	16×2	2	34

3.3. Tensorshare scheduler

System model. The TensorShare scheduler determines job combinations for GPUs that satisfy gSLAs. We explain the workflow of the scheduler using the notations to formulate the scheduling problem, such as jobs and interference. We denote the scheduling queue as $SQ = \{J_1, J_2, \ldots, J_n\}$, where J_i represents the i-th DT job, and n is the total number of jobs in the queue. The objective of TensorShare is to identify a combination $c = \{J_A, J_B\}$ that satisfies the gSLAs of the comprised jobs, which is formally described as $\delta_{J_A} \leq \text{gSLA}_{J_A} \wedge \delta_{J_B} \leq \text{gSLA}_{J_B}$. The scheduler is activated by the event monitor in the following cases: (1) when the GPU cluster and the scheduling queue are initialized, and (2) when a DT job completes its training. In the first case (initialization), the event monitor activates the TensorShare scheduler, which selects the first job (J_1) from the SQ. The scheduler then finds another job from the SQ that can run alongside J_1 as a combination.[6] Note that the scheduler prefers fewer GPU machines to run the combination (e.g., running two workers on one machine with two GPUs instead of two machines with one GPU each). This is because a higher number of servers is more prone to networking bottlenecks between workers [27].

In the second case, when a DT job completes its training, the remaining job continues to use the GPU. The event monitor then triggers the TensorShare scheduler, which selects a new job from the SQ and assigns it to the corresponding GPU machine. We use the term J_R to refer to (1) J_1 in the first case and (2) the remaining (still-running) job in the second case. Thus, for n jobs in the SQ, the TensorShare scheduler has n potential combinations, denoted as $c_i = \{J_R, J_i\}$, where J_i is a DT job in the SQ. We represent the set of potential combinations as $C = \{c_i \mid 1 \leq i \leq n\}$. Based on the proposed system model, notations, and objective, we present the modeling analysis for determining the c_i: *proactive filtering* and *combination selection*.

Proactive filtering. The TensorShare scheduler satisfies gSLAs by avoiding violations through proactive filtering, which operates as follows. First, we filter out c_i from C if J_R and J_i have a different number of workers. This is because when jobs have different worker counts, some workers in the larger job cannot share GPUs with workers from the other job. Although the scheduler could attempt to allocate GPUs across different jobs, this approach increases complexity and causes delays. Also, allowing partial workers to either share GPUs or use dedicated GPUs can lead to dedicated GPU workers being blocked by slower (GPU sharing workers), resulting in poor utilization and fairness issues. Thus, only job pairs $\{J_R, J_i\}$ with identical worker counts are considered as candidates for scheduling.



(a) Empty STA occurrence.

(b) STA and sensitivity.

Fig. 8. Job sensitivity of DT jobs.

In addition, the TensorShare scheduler verifies whether a job pair can share GPU without exceeding the memory capacity. An input feature for the δ predictor includes the memory consumption of each job so that the TensorShare scheduler takes advantage of the feature. The scheduler selects job pairs whose combined memory usage fits within the GPU memory. If the combined memory requirement exceeds the available GPU memory, the corresponding c_i is excluded from consideration; if no valid combinations remain, TensorShare executes J_R in dedicated use without sharing.

Next, for the remaining m number of c_i, proactive filtering requests the δ predictor to predict δ. Specifically, with the input features of J_R and J_i of the c_i, the δ predictor generates δ for each of J_R and J_i. We define D^{c_i}, the predicted δ values for c_i, as $D^{c_i} = \{\delta^{c_i}_{J_R}, \delta^{c_i}_{J_i}\}$. Since δ_{J_R} varies depending on J_i, we distinguish the δ of J_R by using a superscript notation ($\delta^{c_i}_{J_R}$). The TensorShare scheduler obtains m sets of D^{c_i}, for all job pairs in C. Based on D^{c_i}, proactive filtering checks whether c_i can satisfy gSLAs. We denote the gSLAs of c_i as $\{\text{gSLA}_{J_R}\ \text{gSLA}_{J_i}\}$. If either $\delta^{c_i}_{J_R}$ or $\delta^{c_i}_{J_i}$ exceeds its respective gSLA, it indicates a gSLA violation, and c_i is excluded from the set of potential combinations (C). If both J_R and J_i satisfy their gSLAs, proactive filtering adds c_i to its "candidate stack" (STA). Thus, the STA is defined as $\text{STA} = \{c_i \mid \delta^{c_i}_{J_R} \leq \text{gSLA}_{J_R} \wedge \delta^{c_i}_{J_i} \leq \text{gSLA}_{J_i}, c_i \in C\}$.

After examining the gSLA satisfactions of all c_i, proactive filtering counts the number of c_i in the STA. If the STA is empty, no valid combinations exist for J_R, so TensorShare assigns J_R to dedicated use (without GPU sharing). If the STA contains exactly one c_i, only one combination is possible for J_R. TensorShare then directly executes c_i and finishes its scheduling. If there are two or more c_i in the STA, TensorShare proceeds to "combination selection" to determine the best c_i for execution.

Combination selection. For combination selection, we design a selection strategy that optimizes two key aspects: (1) JCT reduction for individual DT jobs (users) and (2) GPU efficiency. To minimize JCT, the selection strategy prioritizes job combinations with lower δ values.[7] This strategy, denoted as S_1, is represented as $S_1 = \text{argmin}_{c_i \in \text{STA}}(\delta^{c_i}_{J_R} + \delta^{c_i}_{J_i})$.

Next, we consider GPU efficiency by analyzing the STA in S_1. When the STA is empty, it indicates that J_R must run with the dedicated use to satisfy gSLAs. Running jobs in the dedicated use results in poorer GPU utilization compared to GPU sharing, which reduces GPU efficiency by increasing both the total execution time and the number of GPUs required.

To analyze the efficiency of S_1, we conduct the following experiment to measure the frequency of empty STA occurrences. We compare two strategies: (1) selecting c_i using S_1 and (2) simply choosing c_1 (the first available combination in the STA), known as the first-in-first-out (FIFO) strategy. For each experiment trial, we run the TensorShare scheduler

[6] If the total number of requested GPUs by the enqueued jobs in the SQ is less than the total number of unused GPUs, the TensorShare scheduler can optionally allocate GPUs without sharing (dedicated use)

[7] Identifying the optimal combination to minimize JCTs for both currently enqueued jobs and future incoming jobs is nearly NP-hard. Therefore, our selection process follows a greedy approach, selecting the best combination based on the currently enqueued DT jobs in the SQ.

50 times per strategy. During these runs, DT jobs are randomly generated from Table 1 and placed into the SQ. We repeat the trial 20 times, resulting in 1000 scheduling rounds per strategy. From these 1000 executions, we calculate the ratio of empty STA occurrences. The first two bars in Fig. 8a, labeled FIFO and S_1, present the experiment results. With S_1, the empty STA occurs 1.5× more frequently than with FIFO. This indicates that 1.5× more DT jobs fail to utilize GPU sharing when using S_1 compared to FIFO.

To reduce the occurrence of an empty STA in S_1, we conduct the following analysis. Fig. 8b shows the δ values (measured in Fig. 3) for two exemplar DT jobs: VGG-11 and InceptionV3. Each ×-marked point in the figure represents one of the 29 distinct δ values for each job. InceptionV3 shows a wider range of δ values compared to VGG-11, with the standard deviations of 0.59 and 0.16, respectively. Assume that the gSLA for both jobs is set to 1.5, as indicated by the red vertical line in Fig. 8b. VGG-11 can pair with 26 jobs (corresponding to ×-marked points in the blue-colored region), allowing it to achieve $\delta <$ gSLA. In contrast, InceptionV3 can pair with only nine jobs (purple-colored region), which is 2.9× less than VGG-11.

This suggests that InceptionV3 has fewer opportunities to pair with other DT jobs in combinations that satisfy the gSLAs. Therefore, when both VGG-11 and InceptionV3 are candidates to run as a combination, it is more advantageous to prioritize running InceptionV3 first. Doing so leaves more opportunities for VGG-11 to pair with other jobs later, as VGG-11 is more likely to find compatible combinations that meet gSLA requirements.

We incorporate this observation into S_1 and derive an improved strategy, S_2. We quantify the range (variation) of δ values of J_i by its standard deviation, which we refer to as Sensitivity_{J_i}. S_2 selects the job with higher sensitivity first and is defined as $S_2 = \text{argmin}_{c_i \in \text{STA}}((\delta^{c_i}_{J_R}/\text{Sensitivity}_{J_R}) + (\delta^{c_i}_{J_i}/\text{Sensiti} - \text{vity}_{J_i}))$. To evaluate the effectiveness of S_2, we conduct the same experiment as before, measuring the occurrence of an empty STA, now applying S_2. The third bar in Fig. 8a shows the empty STA occurrence for S_2. From 1000 scheduling rounds, S_2 significantly reduces the high occurrence of empty STA observed in S_1, bringing it to a level comparable to FIFO (with only a 0.9% difference from FIFO). As S_2 considers both JCT reduction and GPU efficiency, the TensorShare scheduler uses S_2 for its combination selection.

4. Evaluation

We implement TensorShare using approximately ∼2.2K lines for the TensorShare scheduler and ∼1.5K lines for the DT introspector and the δ predictor training. TensorShare is designed to operate with Kubernetes, which runs each PS and GPU worker as a separate container. We conduct the following experiments with our TensorShare prototype.

4.1. Evaluation setup

We evaluate TensorShare across four different evaluation settings: (1) end-to-end evaluation (physical testbed), (2) end-to-end evaluation (simulation), (3) micro-evaluation, and (4) unseen job evaluation. Each evaluation is described below.

End-to-end evaluation (physical testbed). For our physical testbed experiments, we use two GPU machines and one storage machine. The machine specifications and experimental configurations are identical to those in §2.2. These machines are managed via Kubernetes, and container allocation (including GPU workers and PSs) follows the default settings of Kubernetes (e.g., least-allocated policy [39]). A similar policy is used for the simulation experiments (explained below). We compare TensorShare against the following baseline techniques:

- Dedicated use: A GPU scheduling strategy commonly used in existing GPU clouds and production GPU clusters [27,40], where DT jobs run without GPU sharing [13].
- GPU sharing: A Kubernetes plugin that enables GPU sharing without considering JCT increase or interference [8,24].
- Reactive GPU sharing: A GPU sharing approach with preemption, proposed in a previous study [14]. This technique pauses a running job when a severe JCT increase (gSLA violation) is detected.

We evaluate TensorShare using six key metrics: (1) gSLA violation rate, (2) gSLA excess ratio, (3) JCT, (4) GPU time, (5) job throughput, and (6) power consumption. The gSLA violation rate is the ratio of jobs that violate their gSLA to the total number of jobs. The gSLA excess ratio quantifies the extent to which δ exceeds the gSLA for DT jobs, calculated as $100 \times (\delta - \text{gSLA})/\text{gSLA}$. The JCT measures the total time taken for each DT job to complete. GPU time represents GPU efficiency and is defined as the total duration of GPU usage required to complete all enqueued DT jobs in the SQ [27,28]. Job throughput is the number of DT jobs completed per unit time. Power consumption refers to the total energy consumed by GPUs while all enqueued DT jobs in the SQ complete their execution. We measure six metrics in the physical testbed, whereas in the simulation experiments, we do not report the power consumption as the simulator does not support it. Since δ is only defined when GPUs are shared, it cannot be calculated for dedicated use. Therefore, dedicated use is excluded from the measurement of gSLA violation rate and gSLA excess ratio.

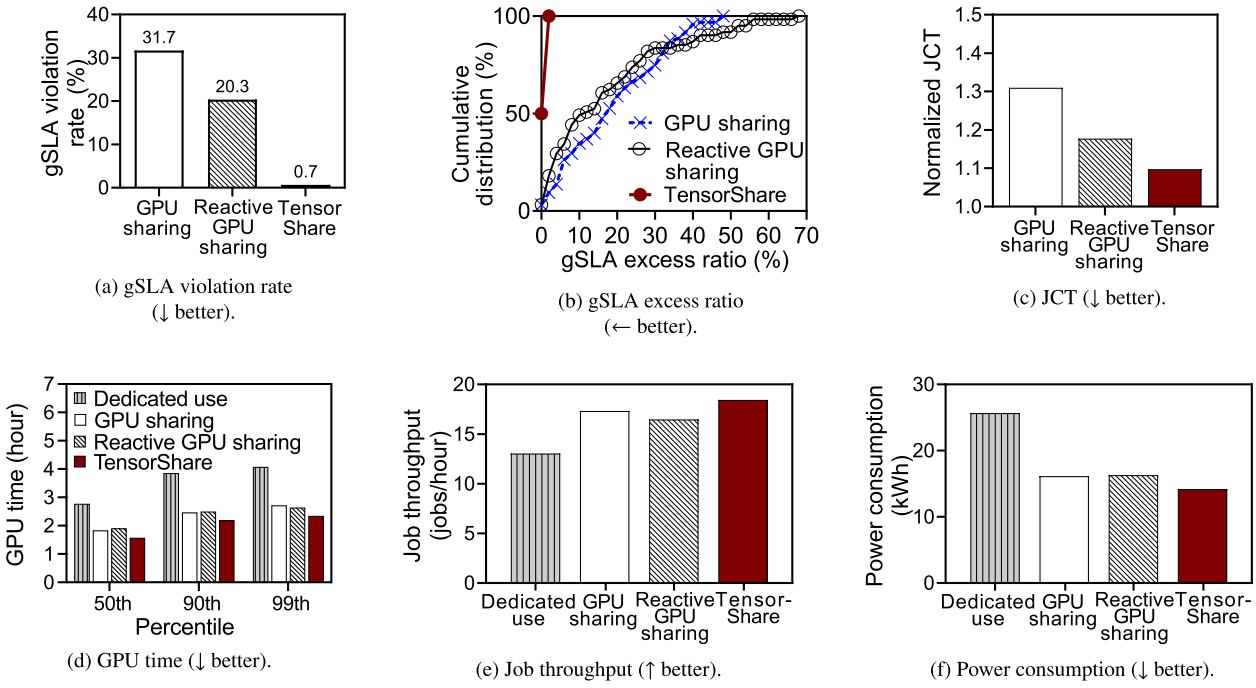
For the workload, we randomly select a job from the 29 DT jobs (Table 1) and enqueue it in the SQ until a total of 10 jobs are enqueued. We run the workers for a job on the same GPU machine to ensure consistent evaluation results and avoid potential network bottlenecks when using multiple servers. In the dedicated use setting, jobs are executed sequentially without GPU sharing. In the GPU sharing setting, two jobs are selected sequentially from the queue and run as a pair. When one of the two jobs completes, the scheduler pairs the remaining (still-running) job with the next job in the queue. In contrast, TensorShare schedules jobs based on its combination selection strategy (§3.3). Each experiment is repeated 30 times, resulting in a total of 300 DT jobs.

For gSLA assignment, we randomly assign gSLA values between 1.0 and 2.0 following a uniform distribution. The lower bound (1.0) corresponds to the training speed of a job when GPUs are not shared. The upper bound (2.0) is based on the assumption that each GPU is shared by two DT jobs, meaning each job uses approximately half of the GPU's resources, leading to an expected 2× increase in JCT. This assumption aligns with prior studies on resource sharing [28].

End-to-end evaluation (simulation). For the simulation experiments, we evaluate larger-scale configurations using an open-source simulator [41], widely used in the research community [7,27]. We extend this simulator by integrating TensorShare into the simulation framework. The simulation experiments are conducted on 32 machines, each equipped with eight GPUs, resulting in a total of 256 GPUs. This setup is comparable in scale to those used in previous studies [9,15,32].

We use Microsoft Philly traces [13], which include DT job information from GPU clouds. The dataset consists of four different traces: Trace 1 (1494 DT jobs), Trace 2 (5755 DT jobs), Trace 3 (991 DT jobs), and Trace 4 (2636 DT jobs). Each trace includes key job attributes such as arrival time, duration, and the number of GPU workers, which we utilize in our simulations. Since the model types of the jobs are unknown, we follow the common practice of prior studies [15,28,32] by randomly assigning job types from a set of representative models (listed in Table 1). DT jobs are enqueued into the SQ according to their arrival times in the trace. Note that the simulations run jobs from the traces with a varying number of workers (up to 8).

For the simulation experiments, we compare TensorShare with three baseline techniques: dedicated use, GPU sharing, and reactive GPU sharing, similar to the physical testbed experiments. In addition, we compare TensorShare with *Horus* [7], as its design is available as an open-source simulator, enabling direct comparison. Horus is a scheduling technique that identifies high GPU or GPU memory utilization as an indicator of significant GPU interference. To mitigate interference, it schedules



(a) gSLA violation rate (↓ better).

(b) gSLA excess ratio (← better).

(c) JCT (↓ better).

(d) GPU time (↓ better).

(e) Job throughput (↑ better).

(f) Power consumption (↓ better).

Fig. 9. End-to-end evaluation (physical testbed) results.

high-utilization jobs to avoid GPU sharing. However, Horus does not predict δ. We evaluate TensorShare using the same five metrics as in the physical testbed experiments.

Micro-evaluation. We also conduct component-level experiments of TensorShare in the physical testbed to better understand its overhead, performance, and generalizability. The following evaluation aspects are analyzed:

- TensorShare overhead: Measure (1) introspection times across 29 distinct DT jobs and (2) the scheduling delay as the number of jobs increases.
- δ prediction accuracy and training time: Compare predicted δ values against ground truth using percentage error and evaluates the training time required for the δ predictor using NAS.
- Generalizability of the δ predictor on new hardware: Evaluate the prediction accuracy of the δ predictor fine-tuned via transfer learning on different hardware setups (e.g., H100 and A100).

Unseen job evaluation. Lastly, we evaluate TensorShare on unseen DT jobs that were never used in training the δ predictor. We assess the generalization ability of the δ predictor on these unseen jobs and examine whether TensorShare can still satisfy gSLA constraints. For unseen jobs, we use LLMs with entirely different model structures, datasets, and hyperparameters. Specifically, we evaluate five open-source LLMs: OpenLLaMA-7B [20], MPT-7B [42], Falcon-7B [43], RedPajama-7B [44], and FlanT5-7B [45]. These models are supported by OpenNMT-py [46], a widely used library for benchmarking LLMs. Please note that LLMs are both compute-intensive and memory-intensive, and differ significantly between themselves in terms of architecture, dataset, and hyperparameters.

We fine-tune these models on the Alpaca dataset [47], which consists of 52,000 input instructions and output values. The Alpaca dataset is a widely used open-source dataset for LLM training. Unlike previous experiments, these LLMs are executed using PyTorch, making them completely unseen in the context of prior evaluations. The models are configured with an input token size of 256, the maximum size that can be loaded within the memory of V100 GPUs. We conduct the following evaluations in the physical testbed:

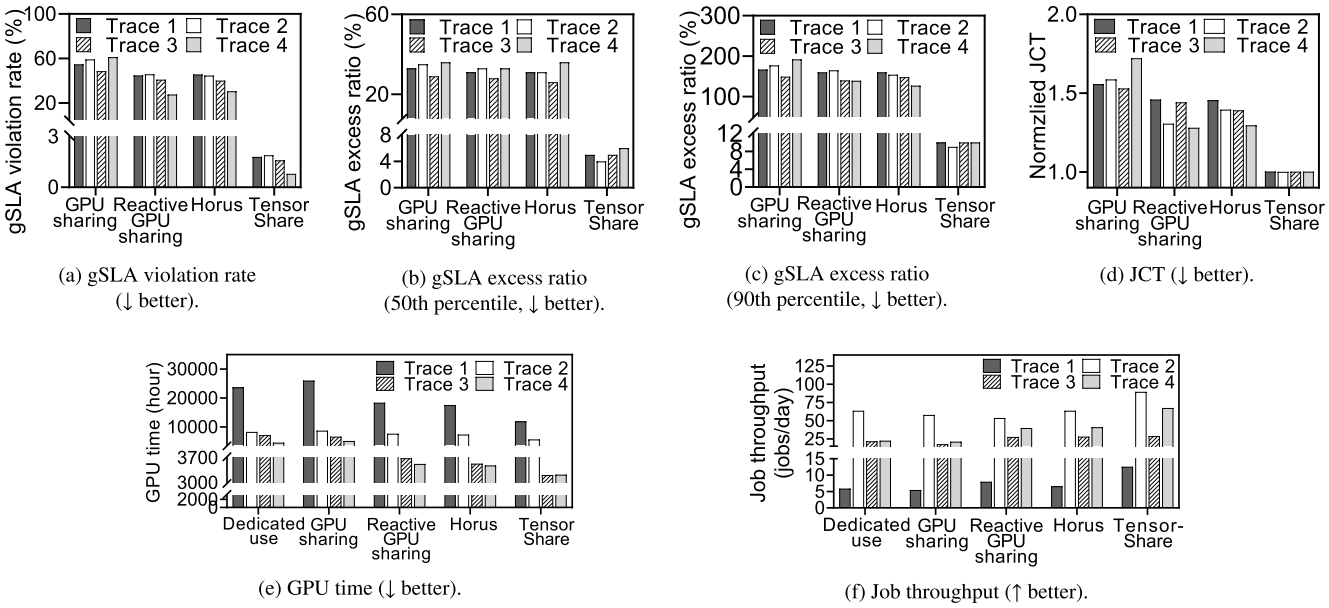
- Generalizability of the δ predictor: We evaluate the accuracy of the δ predictor across 160 job combinations, ensuring that each combination includes at least one unseen job.
- gSLA violation rate, JCT, and GPU time: We enqueue 100 DT jobs in the SQ and apply the TensorShare scheduler. Half of the jobs are unseen, while the other half are seen jobs (from Table 1). This mix of seen and unseen jobs broadens the scope of the evaluation and enhances its realism, covering diverse model types such as image classification, natural language processing, and LLMs.

4.2. End-to-end evaluation (physical testbed)

gSLA violation rate. Fig. 9a shows the gSLA violation rate (y-axis) for each technique (x-axis). TensorShare outperforms the other techniques, achieving an extremely low gSLA violation rate of only 0.7%. Compared to GPU sharing and reactive GPU sharing, TensorShare reduces the gSLA violation rate by 47.3× and 30.3× on average, respectively. This demonstrates that TensorShare effectively meets the gSLA requirements for DT jobs. Note that the 0.7% violation rate observed in TensorShare is attributed to prediction errors in the δ predictor.

gSLA excess ratio. Fig. 9b presents the CDF of the gSLA excess ratio, indicating how severely each job exceeds its gSLA. At the median (50% on the y-axis), TensorShare exceeds the gSLA by only 1%, demonstrating that the violation is very minor and negligible. In contrast, GPU sharing and reactive GPU sharing exceed the gSLA by 17.7% and 12%, respectively—16.9× and 11.5× worse than TensorShare. At the tail-end (99.9% on the y-axis), the gSLA excess ratios for TensorShare, GPU sharing, and reactive GPU sharing are 1.3%, 48.2%, and 67.9%, respectively, meaning that TensorShare outperforms GPU sharing and reactive GPU sharing by factors of 36× and 50× at the tail-end. These results indicate that even for small number of jobs that violate their gSLA (corresponding to the 0.7% in Fig. 9a), TensorShare maintains a minimal gSLA excess ratio of around 1% across both the median and tail-end cases.

JCT. Fig. 9c illustrates the JCT for each GPU sharing technique. The JCT values are normalized relative to dedicated use. The bars represent the average normalized JCT for each technique. Among the techniques, GPU sharing exhibits the highest (worst) JCT, followed by reactive GPU



(a) gSLA violation rate (↓ better).

(b) gSLA excess ratio (50th percentile, ↓ better).

(c) gSLA excess ratio (90th percentile, ↓ better).

(d) JCT (↓ better).

(e) GPU time (↓ better).

(f) Job throughput (↑ better).

Fig. 10. End-to-end evaluation (simulation) results.

sharing, with TensorShare achieving the lowest JCT. Compared to GPU sharing and reactive GPU sharing, TensorShare reduces the JCT increase relative to dedicated use by factors of 3.2× and 1.8×, respectively. These results highlight the efficiency of TensorShare's selection strategy in reducing JCT while maintaining GPU sharing effectiveness.

GPU time. Fig. 9d compares GPU time across the 50th, 90th, and 99th percentiles. TensorShare achieves the shortest (best) GPU time among all techniques. At the median (50th percentile), TensorShare reduces GPU time by 12% compared to GPU sharing and 45% compared to dedicated use, while at the tail-end (99th percentile), it improves GPU time by 12% over reactive GPU sharing and 43% over dedicated use.

When compared to the ideal GPU time (defined as half of the dedicated use, as explained in §2.2, though not shown in the graph), TensorShare's GPU time at the median is only 12% higher than the ideal value. In contrast, GPU sharing and reactive GPU sharing exceed the ideal by 32% and 37%, respectively. These results indicate that TensorShare effectively optimizes both GPU infrastructure utilization and computation time, enhancing overall efficiency.

Job throughput. Fig. 9e shows the number of jobs per hour in the physical testbed experiments. Even while satisfying the gSLA of individual jobs, TensorShare achieves the highest through-put—1.42× compared to dedicated use, 1.07× compared to GPU sharing, and 1.12× compared to reactive GPU sharing. The results reveal that TensorShare effectively improves cluster-wide resource utilization.

Power consumption. Fig. 9f presents the power consumption measured on our physical testbed. TensorShare reduces power consumption by 45%, 12%, and 13% compared to dedicated use, GPU sharing, and reactive GPU sharing, respectively. The results show that TensorShare lowers power consumption, which improves the GPU cluster's energy efficiency.

4.3. End-to-end evaluation (simulation)

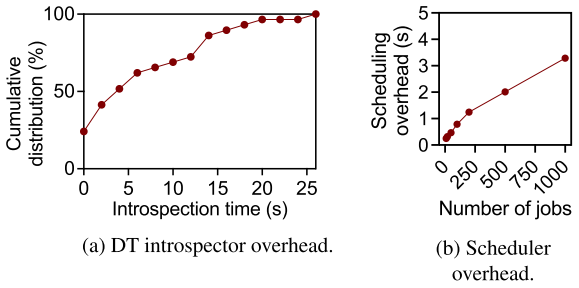
gSLA violation rate. Fig. 10a shows the gSLA violation rate obtained from the simulations. Each bar represents the violation rate per trace. Across all four traces, TensorShare maintains a low gSLA violation rate of ∼1.9% (1.5% on average). In contrast, GPU sharing shows the highest violation rate, with an average of 56%, while reactive GPU sharing and Horus exhibit average violation rates of 39.8% and 40.3%, respectively. Thus, TensorShare improves the gSLA violation rate by 36.7×, 26.1×, and 26.4× compared to GPU sharing, reactive GPU sharing, and Horus, respectively. Note that all techniques experience higher violation rates in the simulation than in the physical testbed (on average, 19.3% higher for existing techniques), likely due to the larger number of DT jobs and GPUs in the simulation. However, the difference for TensorShare is only 0.8%, which is 24× lower (better) than the other techniques.

gSLA excess ratio. Figs. 10b and 10c present the gSLA excess ratio at the median (50th percentile) and tail-end (90th percentile), respectively. Among the four techniques, TensorShare consistently achieves the lowest excess ratio across all traces, averaging 5% at the median and 9.8% at the tail-end. Compared to GPU sharing, reactive GPU sharing, and Horus, TensorShare reduces the median excess ratio by 6.7×, 6.3×, and 6.2×, respectively. At the tail-end, it further improves the excess ratio by 17.6×, 15.5×, and 15.1× on average.

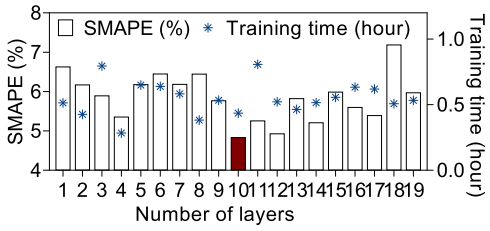
JCT. Fig. 10d presents the normalized JCT relative to dedicated use. The bars represent average JCT values. TensorShare outperforms GPU sharing, reactive GPU sharing, and Horus across all four traces, reducing JCT by 60%, 37%, and 38%, respectively. Compared to dedicated use, the JCT of TensorShare increases by only 2% on average. These results further confirm that TensorShare's selection strategy is highly effective in reducing JCT and improving scheduling efficiency.

GPU time. Fig. 10e shows the GPU time for each GPU sharing technique, with each bar representing the time per trace. TensorShare consistently outperforms all comparative techniques across all traces. On average, TensorShare achieves 41%, 44%, 21%, and 18% shorter (better) GPU time compared to dedicated use, GPU sharing, reactive GPU sharing, and Horus, respectively.

Next, we examine the GPU time of TensorShare per trace. TensorShare achieves the highest reduction in GPU time in trace 1 (43%), followed by trace 3 (32%), trace 2 (29%), and trace 4 (20%), compared to other techniques. Note that the order of improvement aligns with the number of DT jobs in each trace: trace 1 (5755), trace 3 (2636), trace 2 (1494), and trace 4 (991). These results indicate that TensorShare effectively enhances GPU infrastructure efficiency by identifying more optimal job combinations for GPU sharing, particularly as the number of DT jobs in GPU clouds scales up.



(a) DT introspector overhead. (b) Scheduler overhead.



(c) δ predictor accuracy (bars, left y-axis) and training time (dots, right y-axis).

Fig. 11. Micro-evaluation results.

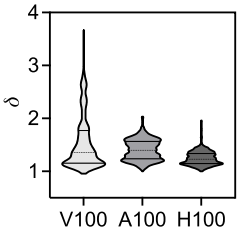


Fig. 12. Distribution of the δ across GPU setups.

Table 5
Comparison of prediction accuracy and training time for transfer learning and full retraining across GPU setups.

GPU setup	Prediction accuracy (SMAPE)		Training time (min)	
	Transfer learning	Full retraining	Transfer learning	Full retraining
H100	4.9%	3.5%	5	444
A100	6.1%	5.8%	3	783

Job throughput. Fig. 10f shows the number of completed jobs per day. TensorShare consistently outperforms all comparative techniques across all traces. On average, TensorShare achieves 2.0×, 2.2×, 1.5×, and 1.5× higher throughput compared to dedicated use, GPU sharing, reactive GPU sharing, and Horus, respectively. This demonstrates that TensorShare effectively enhances resource utilization across the entire cluster.

4.4. Micro-evaluation

TensorShare overhead. We measure TensorShare's overhead in two aspects: (1) DT introspector and (2) TensorShare scheduler. First, note that the DT introspector runs for profiling only when each job enters the SQ, and its execution time varies depending on the DT job type. So, we measure the introspection time of 29 DT jobs used in our evaluation. Fig. 11a shows the cumulative distribution: the median is 3.9 s, ranging from 0.24 to 25 s per job, and the standard deviation is 7.0 s. Such job profiling overhead, including its variability, is within the range reported in other works (standard deviation of 9.5 s) [48].

Next, we analyze the overhead of the TensorShare scheduler. The scheduler examines the SQ to pair jobs that satisfy the gSLA constraint, so its delay increases with the queue length. To assess this at scale, we increase the number of jobs from 10 to 1000. Note that the average queue length in real-world GPU clusters is 117 jobs [49], so our new analysis covers the practical scale. Fig. 11b shows the scheduling delay, which remains under 3.3 s even for 1000 jobs. In the worst case, the combined overhead per job is 28.3 s (25 s for introspection and 3.3 s for scheduling), which remains well below the average waiting time in real-world GPU clusters—approximately 3000 s before execution [49]. We believe this demonstrates the scalability of TensorShare in large clusters.

δ prediction accuracy and training time. Fig. 11c presents the prediction accuracy (bars on the left y-axis) and training time (∗ marks on the right y-axis). The x-axis represents the number of MLP layers, and for each layer count, we show: (1) the best accuracy (bar) obtained from different structural options (e.g., activation function) and (2) the training time (∗ mark) required to achieve the best accuracy. Training is conducted using an Intel Xeon Gold 6342 CPU, and accuracy is evaluated using a dataset portion (20%) not used for training (§3.2). We calculate the accuracy using the symmetric mean absolute percentage error (SMAPE) [37].

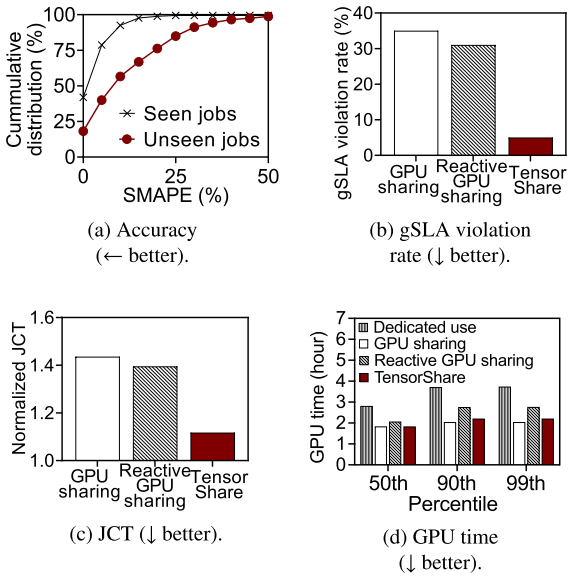
For prediction accuracy, the best error rate is achieved with 10 MLP layers, yielding a SMAPE of only 5%, indicating highly accurate predictions. For training time, it takes 10.4 hours in total to train all models shown on the x-axis and to identify the most accurate model. We believe that training time can be significantly reduced by using GPUs instead of CPUs.

Generalizability of the δ predictor on new hardware. To show the generalizability and robustness of the δ predictor, we configure two additional hardware setups: 1) NVIDIA H100 GPU with AMD EPYC 7J13 CPU and 2) NVIDIA A100 GPU with Intel Xeon Platinum 8480+ CPU, in addition to the V100 GPUs described in §2.1. Note that these setups differ from the V100 GPU setup in terms of GPU architecture, GPU memory capacity, and CPU type, and these are representative GPUs widely adopted in both academia and industry. For dataset, we measure input features and δ for the H100 and A100 setups following the same methodology described in §3.2.

Fig. 12 shows the distribution of the δ, which varies across GPU setups. The variance of δ is different per GPU type; newer-generation GPUs (from left to right on the x-axis) tend to exhibit lower δ values, with standard deviations of 0.49, 0.19, and 0.13 and means of 1.54, 1.41, and 1.25 for V100, A100, and H100, respectively. It is due to their improved computational capability and memory bandwidth.

To handle such variance of interferences per GPU type, we fine-tune the pre-trained δ predictor from the V100 setup (i.e., 10 MLP layers DNN) for each target GPU setup by transfer learning. Specifically, we fine-tune a subset of layers from the last layer (regression_head in Table 4) to MLP_2, while keeping the first layer (MLP_1) frozen to preserve the knowledge from the V100 setup.

To examine how transfer learning effectively enables generalization, we fully retrain the δ predictor from scratch for each new GPU setup. For retraining, we apply the same NAS process used in §3.2, which involves exploring various model architectures and hyperparameters. As shown in Table 5, the training time of transfer learning is 5 min vs. 444 min for full re-training for H100, and 3 min vs. 783 min for A100. We find that this huge reduction is due to the difference in the number of models explored: full retraining explores 1677 and 1711 models for H100 and A100, respectively, while transfer learning explores only the number of fine-tuned layers—9 models for H100 and A100, respectively. This means that our transfer learning searches 186×—190× fewer models than full retraining. Yet, Table 5 shows that the prediction accuracy (SMAPE) of transfer learning differs from the fully retrained δ predictor by only 1.4% and 0.3% for the H100 and A100 setups, respectively.



(a) Accuracy (← better).

(b) gSLA violation rate (↓ better).

(c) JCT (↓ better).

(d) GPU time (↓ better).

Fig. 13. Unseen job evaluation.

These results demonstrate that the δ predictor remains generalizable in terms of accuracy and training time.

4.5. Unseen job evaluation

Generalizability of the δ predictor. We use the δ predictor trained with jobs in Table 1 to evaluate its generalizability for unseen jobs (five LLMs). The input features for unseen jobs are obtained using the DT introspector (§3.1). Fig. 13a presents the CDF of prediction errors, measured using SMAPE. The line with circles represents the δ predictor's accuracy across 160 job combinations, each combination includes at least one unseen job. The line with × marks represents the accuracy of the δ predictor across job combinations of only seen jobs. At the median (50% on the y-axis), the SMAPE values for unseen and seen job combinations are 9% and 3%, respectively. Despite the fact that these LLMs run on a different framework (PyTorch instead of TensorFlow), which may introduce differences that affect the predictor's performance, our δ predictor maintains good accuracy. These results on unseen jobs demonstrate the reasonable generalizability of the δ predictor.

gSLA violation rate. Fig. 13b presents the gSLA violation rate for unseen job combinations. The bars (y-axis) represent the gSLA violation rate. TensorShare demonstrates that only 5 out of 100 jobs (5%) violate their gSLAs, which is a 4% increase compared to seen jobs (Fig. 9a). Despite this slight increase, TensorShare still significantly outperforms other techniques—achieving 7× and 6.2× better gSLA satisfaction compared to GPU sharing and reactive GPU sharing, respectively. These results indicate that TensorShare effectively improves gSLA satisfaction even for unseen jobs. Given that the LLMs have huge differences from the workloads in Table 1, these results show that TensorShare can handle heterogeneous workloads that reflect production environments.

JCT. Fig. 13c presents the JCT of the three techniques on unseen job combinations, using normalized JCT, consistent with §4.2. TensorShare reduces JCT by 28.5% and 24.9% compared to GPU sharing and reactive GPU sharing, respectively. Also, we observe that TensorShare's normalized JCT increases by only 1.8% compared to seen jobs (Fig. 9a), while GPU sharing and reactive GPU sharing experience significantly higher increases of 9.6% and 18.5%, respectively. These results indicate that TensorShare outperforms the other two techniques by 5.4× and 10.3× in terms of JCT consistency.

GPU time. Fig. 13d presents the GPU time of the three techniques on unseen jobs, showing 50th (median), 90th, and 99th percentile values, consistent with Fig. 9a. At the median, TensorShare achieves the shortest GPU time, outperforming GPU sharing and reactive GPU sharing by 1% and 11%, respectively. However, at the 90th and 99th percentiles (tail-end cases), GPU sharing shows 7% lower GPU time on average compared to TensorShare. The increase in TensorShare's GPU time at the tail-end is due to inaccurate δ predictions, which occasionally result in suboptimal scheduling decisions. Specifically, we observe that in 6.2% of prediction trials, the scheduler overestimates δ, predicting higher values than the actual δ. As a result, TensorShare often schedules jobs to run alone instead of sharing GPUs to meet the gSLA requirements (§3.3). While this approach increases tail-end GPU time, it ensures that gSLAs remain satisfied. Thus, the higher GPU time at the tail-end represents a trade-off for maintaining gSLA satisfaction, with a 5% gSLA violation rate for unseen job combinations, as shown in Fig. 13b.

5. Discussion

This section discusses possible extensions of TensorShare related to GPU sharing, GPU virtualization, SLA metrics, DT strategies, GPU types, job starvation, prediction accuracy by dataset size, algorithm choice, use of ML, and job execution behavior.

Two jobs for GPU sharing. In this study, we focus on GPU sharing between two jobs. Previous studies have also focused on two-job sharing, as sharing GPUs among three or more jobs leads to significant interference [5,9]. For example, when we test three-job sharing, δ ranges from 1.15 to 6.28, which is 1.9× worse than two-job sharing. Thus, we believe that limiting GPU sharing to two jobs is a reasonable choice. However, if future advancements—such as new GPU architectures—help mitigate severe interference between multiple jobs, TensorShare could be extended as follows. First, the δ predictor can be retrained on an N-job dataset (N > 2) while maintaining the same feature types, model structure, and training methods. Also, when a job that shares GPUs completes its training, the TensorShare scheduler dynamically assigns a new job to share GPUs with J_R. By generalizing J_R to represent a group of jobs that continue running on GPUs, the TensorShare scheduler could efficiently handle multiple concurrent jobs.

GPU virtualization. GPU virtualization creates virtual GPU (vGPU) instances by slicing a physical GPU. A representative technique is NVIDIA's multi-instance GPU (MIG) that divides a physical GPU into up to seven vGPU instances, each with dedicated SM cores and memory. Previous studies [26,50,51] reported that each vGPU instance provides strong performance isolation from other vGPUs because the SM cores and memory inside the vGPU are not shared. So, during distributed training using MIG instances, performance is isolated and no interference occurs as reported in previous works [26,50,51]. In contrast, GPU sharing on which TensorShare proposes allows multiple jobs to run on the same physical GPU or on a vGPU instance without partitioning SM cores and memory. A key idea of TensorShare is to multiplex the SM cores and GPU memory of the physical GPU or vGPU instance between multiple jobs. As the GPU resources are shared, interference arises between jobs, which can increase job completion time.

In short, MIG offers strong isolation among instances; however, it has critical limitations in real-world deployments: MIG is supported only on specific GPU architectures (e.g., A100, H100) and requires static partitions that cannot be dynamically resized after instantiation. Also, workloads running on vGPU instances under MIG can experience performance degradation compared to using the entire physical GPU [25,50]. Consequently, GPU virtualization alone is insufficient to satisfy per-job gSLA requirements in heterogeneous GPU clouds.

We believe TensorShare is a complementary approach to GPU virtualization. Rather than eliminating interference through strict hardware slicing, TensorShare permits interference but proactively controls its impact by introducing the concept of gSLA. Moreover, TensorShare can

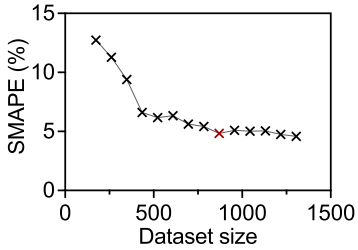


Fig. 14. Prediction accuracy of the δ predictor with varying dataset sizes.

Table 6
Comparison of prediction accuracy and prediction latency between the machine learning approach and simpler analytical models.

Algorithm	Accuracy (SMAPE)	Latency (ms)
Linear regression	17.4%	0.10
Ridge regression	17.3%	0.07
Lasso regression	17.7%	0.08
Elastic net regression	17.7%	0.08
SVR with Gaussian kernel	17.3%	0.67
SVR with linear kernel	11.5%	0.72
SVR with polynomial kernel	10.7%	0.51
MLP-based DNN	**4.8%**	**13**

also operate on top of vGPUs (e.g., MIG instances), as the fine-grained metrics used for δ prediction (e.g., SM_ACTIVE and DRAM_ACTIVE) remain observable in MIG instance. The goal of TensorShare is to offer flexible sharing with user-defined gSLA guarantees and forms a complementary foundation for predictable and efficient GPU clouds.

JCT as gSLA. Since DT jobs often run for days or weeks, JCT serves as a more meaningful measure of training efficiency from the user's perspective than momentary metrics such as per-iteration time. Thus, we define gSLA as the ratio of JCT increase when DT jobs share GPUs. In the future, we plan to expand SLA metrics to other performance aspects, such as availability.

Other DT strategies. This study focuses on data parallelism with PS. However, various other strategies exist, such as model parallelism [22] and all-reduce [52]. We believe that TensorShare is not limited to data parallelism with PS. For instance, the DT introspector can be implemented independently of the DT strategy by leveraging external profiling tools such as DCGM [31]. In future work, we plan to extend TensorShare to support additional DT strategies.

Number of machines for a job. We evaluate TensorShare and other techniques using a single machine to run job workers, a common approach to avoid network bottlenecks caused by varying server counts. Note that server selection for workers has been studied in prior work [27,53,54], and TensorShare can complement these studies.

Job starvation and fairness. Since the TensorShare scheduler prioritizes jobs that satisfy gSLAs, some jobs may experience starvation in scheduling. We have not explicitly addressed starvation in this paper. Previous studies that schedule GPUs without sharing, such as Tiresias [27], Themis [28], and Gavel [5], incorporated aging to mitigate starvation. They gradually raise the priority of jobs that remain in the queue across multiple scheduling rounds, which ensures fairness among jobs. Please note that this aging is added to the scheduling. So TensorShare can incorporate the similar aging mechanism—for instance, by increasing the priority of DT jobs once their gSLA violations exceed a certain threshold. This aging mechanism can mitigate starvation while also preserving TensorShare's gSLA-driven scheduling policy.

Prediction accuracy by dataset size. We vary the number of dataset records used for training the δ predictor from 150 to 1300. Fig. 14 shows the prediction accuracy as the dataset size changes. At 870 records, the error reaches 4.8%, and with more records, the error remains similar (averaging 4.9%). So, we use 870 records for the training dataset.

Algorithm choice and model performance. The effectiveness of deep learning algorithms is task-dependent because they have different characteristics. CNN is known to be effective for image-related task, whereas LSTM is good for time-related task. Different tasks need different algorithms—for our paper, we believe that MLP works well because the input data is structured, not having patterns in 2D or time-series. This is verified by the fact that DNN outperforms CNN and LSTM in the context of δ predictor. There are also reports where DNN works better than CNN and LSTM [55,56].

Use of ML. δ predictor takes DNN, which is more complex than simple analytical models, such as linear regression. To further justify our choice, we conduct additional experiments to explain why machine learning approach is effective over simpler analytical models. We compare the δ predictor (MLP-based DNN) with seven simpler models: linear regression, ridge regression, lasso regression, elastic net regression, support vector regression (SVR) with Gaussian kernel, SVR with linear kernel, and SVR with polynomial kernel. Table 6 shows each model's prediction accuracy (SMAPE) and prediction overhead (latency), indicating that the δ predictor achieves the best accuracy, outperforming other simpler models by up to 3.3×. Although its prediction overhead is higher, we believe prediction accuracy is most important in order to ensure reliable gSLA guarantees. Also, this overhead is tolerable as jobs in production clusters typically wait 3000 s before execution [49], during which TensorShare can operate. Furthermore, by caching prediction results, the TensorShare scheduler can handle 1000 jobs in under 3.3 s (explained in Fig. 11b).

Short and bursty job behavior. One might be curious whether TensorShare works well on short and bursty jobs. TensorShare is designed for distributed training (DT) in GPU clusters that have the following characteristics. First, DT jobs exhibit long execution durations. For example, existing studies analyzing job traces from Microsoft and Alibaba reported that individual jobs ran for several minutes to even weeks or months [8,13]. We measure SM_ACTIVE and DRAM_ACTIVE at 100 ms intervals, and so considering the long DT job execution time, we believe that these two metrics are reasonably reliable indicators for the prediction. Second, users of GPU clusters may submit DT jobs in a bursty manner. However, the cluster scheduler launches jobs only when sufficient GPUs are available. As a result, even if job submissions are bursty, actual job execution is regulated and does not occur in a bursty fashion. So, we believe TensorShare's design is aligned with the cluster scheduler in recent studies on distributed training [7,26,57].

6. Related work

This section presents key related work on TensorShare in terms of GPU sharing and SLA satisfaction.

GPU sharing. Various studies used GPU sharing to improve GPU infrastructure efficiency [5–7,9,14,15,24]. For example, KubeShare [6] and Alibaba GPU plugin [24] enabled GPU sharing within Kubernetes. However, these studies are not aware of interference between jobs that share GPUs. Another study, AntMan [15], considered interference between jobs and managed GPU kernel executions by giving delays. Orion [57] intercepts low-level GPU instructions (kernel executions) and limits the execution of low-priority jobs to reduce interference. Gavel [5] observed the JCT increase in GPU sharing and utilized past job execution logs for scheduling. Horus [7] and Lucid [9] observed the interferences, profiled coarse-grained GPU metrics of the jobs (such as GPU utilization), and indirectly used GPU utilization on the scheduling. Gandiva [14] used reactive GPU sharing that monitors the training speeds

of jobs and preempts the problematic jobs. To our knowledge, however, none of the existing studies satisfy job gSLAs, predict δ, or reduce JCTs and GPU time simultaneously.

SLA satisfaction for DT jobs. Other previous studies defined various SLAs for DT jobs and proposed scheduling techniques to satisfy them [15,16,53,54]. Chronus [53] and ElasticFlow [54] defined SLAs on the training completion deadline. Note that these studies are all based on the dedicated use, not on the GPU sharing. AntMan [15] defined two classes of SLAs: resource-guarantee and opportunistic. To ensure the requested resources of resource-guarantee jobs (e.g., GPU memory amount), AntMan used reactive GPU sharing (preemption) on the opportunistic jobs that share GPUs. Singularity [16] defined similar types of SLAs into three classes: premium, standard, and basic. To provide the best and ideal training speeds for premium jobs, Singularity used reactive GPU sharing (preemption) on the standard and basic jobs. HSM [58] proposed an estimation model with equations for the SM occupancy ratio between GPU sharing jobs and dedicated use jobs. However, it lacks any guarantees on the given gSLAs between jobs. To the best of our knowledge, TensorShare is the first study on scheduling technique that concretely defines gSLA for the JCT increase in GPU sharing and proactively satisfies it without any preemptions.

7. Conclusion

This study proposes TensorShare, a proactive GPU sharing technique designed to satisfy the SLA of DT jobs. TensorShare consists of the DT introspector, the δ predictor, and the TensorShare scheduler. The DT introspector extracts input features for the δ predictor that, in turn, estimates the JCT increase. We find that the δ predictor achieves ∼5% error rates. Based on the δ predictions and job sensitivity characteristics, the TensorShare scheduler proactively filters out gSLA-violating job combinations and selects the best combination that reduces the JCT and GPU time. Our evaluation demonstrates that TensorShare improves gSLA satisfaction in the physical testbed and simulation experiments ∼47.3× and ∼36.7×, respectively. Furthermore, TensorShare improves the JCT and GPU time ∼60% and ∼44%, respectively. In addition, as unseen job evaluation, we experiment with TensorShare on LLM fine-tuning jobs and achieve ∼7× and ∼10.3× improvements in gSLA satisfaction and JCT consistency compared to existing techniques.

CRediT authorship contribution statement

Changyong Shin: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Data curation, Conceptualization; **Younghun Go:** Writing – original draft, Visualization, Validation, Software, Methodology; **Yeonho Yoo:** Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Conceptualization; **Jinwoo Jeong:** Writing – review & editing, Validation, Investigation; **Jaehyun Hwang:** Writing – review & editing, Visualization, Validation, Investigation; **Gyeongsik Yang:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Investigation, Funding acquisition, Conceptualization; **Chuck Yoo:** Writing – review & editing, Writing – original draft, Supervision, Investigation, Project administration, Funding acquisition.

Data availability

Data will be made available on request.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Gyeongsik Yang and Chuck Yoo report financial support was provided by National Research Foundation of Korea. Gyeongsik Yang reports financial support was provided by Institute of Information & Communications Technology Planning & Evaluation. Gyeongsik Yang and Chuck Yoo report a relationship with National Research Foundation of Korea that includes: funding grants. Gyeongsik Yang reports a relationship with Institute of Information & communications Technology Planning & Evaluation that includes: funding grants. Changyong Shin has patent pending to KOREA UNIVERSITY RESEARCH AND BUSINESS FOUNDATION. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (RS-2021-NR060143), by the NRF grant funded by the Korea government (MSIT) (RS-2024-00336564, RS-2023-NR077249), by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by MSIT (RS-2024-00405128), by ICT Creative Consilience Program through IITP grant funded by MSIT (IITP-2026-RS-2020-II201819), and by the Google Cloud Research Credits program. During the preparation of this work, the authors used ChatGPT in order to check grammars or typos. After using this tool/service, the authors reviewed and edited the content as needed and take(s) full responsibility for the content of the published article.

References

[1] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, A.Y. Ng, Deep Speech: Scaling up end-to-end speech recognition, 2014, arXiv:1412.5567

[2] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, Commun. ACM 60 (6) (2017) 84-90. https://doi.org/10.1145/3065386

[3] S. Grigorescu, B. Trasnea, T. Cocias, G. Macesanu, A survey of deep learning techniques for autonomous driving, J. Field Rob. 37 (3) (2020) 362–386. https://doi.org/10.1002/rob.21918

[4] D.W. Otter, J.R. Medina, J.K. Kalita, A survey of the usages of deep learning for natural language processing, IEEE Trans. Neural Netw. Learn. Syst. 32 (2) (2021) 604–624. https://doi.org/10.1109/TNNLS.2020.2979670

[5] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, M. Zaharia, Heterogeneity-Aware cluster scheduling policies for deep learning workloads, in: 14Th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), USENIX Association, 2020, pp. 481–498. https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak.

[6] T.-A. Yeh, H.-H. Chen, J. Chou, Kubeshare: a framework to manage GPUs as first-Class and shared resources in container cloud, in: Proceedings of the 29Th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 173-184. https://doi.org/10.1145/3369583.3392679

[7] G. Yeung, D. Borowiec, R. Yang, A. Friday, R. Harper, P. Garraghan, Horus: interference-Aware and prediction-Based scheduling in deep learning systems, IEEE Trans. Parallel Distrib. Syst. 33 (1) (2022) 88–100. https://doi.org/10.1109/TPDS.2021.3079202

[8] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, Y. Ding, MLaaS in the wild: workload analysis and scheduling in large-Scale heterogeneous GPU clusters, in: 19Th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), USENIX Association, Renton, WA, 2022, pp. 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng.

[9] Q. Hu, M. Zhang, P. Sun, Y. Wen, T. Zhang, Lucid: a non-Intrusive, scalable and interpretable scheduler for deep learning training jobs, in: Proceedings of the 28Th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Association for Computing Machinery, New York, NY, USA, 2023, p. 457-472. https://doi.org/10.1145/3575693.3575705

[10] H. Ballani, P. Costa, T. Karagiannis, A. Rowstron, Towards predictable datacenter networks, SIGCOMM Comput. Commun. Rev. 41 (4) (2011) 242-253. https://doi.org/10.1145/2043164.2018465

[11] F. Xu, F. Liu, H. Jin, A.V. Vasilakos, Managing performance overhead of virtual machines in cloud computing: a survey, state of the art, and future directions, Proc. IEEE 102 (1) (2014) 11–31. https://doi.org/10.1109/JPROC.2013.2287711

[12] K. Jang, J. Sherry, H. Ballani, T. Moncaster, Silo: predictable message latency in the cloud, SIGCOMM Comput. Commun. Rev. 45 (4) (2015) 435-448. https://doi.org/10.1145/2829988.2787479

[13] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, F. Yang, Analysis of large-Scale multi-Tenant GPU clusters for DNN training workloads, in: 2019 USENIX Annual Technical Conference (USENIX ATC 19), USENIX Association, Renton, WA, 2019, pp. 947–960. https://www.usenix.org/conference/atc19/presentation/jeon.

[14] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, L. Zhou, Gandiva: introspective cluster scheduling for deep learning, in: 13Th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), USENIX Association, Carlsbad, CA, 2018, pp. 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao.

[15] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, Y. Jia, Antman: dynamic scaling on GPU clusters for deep learning, in: 14Th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), USENIX Association, 2020, pp. 533–548. https://www.usenix.org/conference/osdi20/presentation/xiao.

[16] D. Shukla, M. Sivathanu, S. Viswanatha, B. Gulavani, R. Nehme, A. Agrawal, C. Chen, N. Kwatra, R. Ramjee, P. Sharma, A. Katiyar, V. Modi, V. Sharma, A. Singh, S. Singhal, K. Welankar, L. Xun, R. Anupindi, K. Elangovan, H. Rahman, Z. Lin, R. Seetharaman, C. Xu, E. Ailijiang, S. Krishnappa, M. Russinovich, Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads, 2022, arXiv:2202.07848

[17] H. Qi, E.R. Sparks, A. Talwalkar, Paleo: a performance model for deep neural networks, in: International Conference on Learning Representations, 2017. https://openreview.net/forum?id=SyVVJ85lg.

[18] G.X. Yu, Y. Gao, P. Golikov, G. Pekhimenko, Habitat: a runtime-Based computational performance predictor for deep neural network training, in: 2021 USENIX Annual Technical Conference (USENIX ATC 21), USENIX Association, 2021, pp. 503–521. https://www.usenix.org/conference/atc21/presentation/yu.

[19] G. Yang, C. Shin, J. Lee, Y. Yoo, C. Yoo, Prediction of the resource consumption of distributed deep learning systems, Proc. ACM Meas. Anal. Comput. Syst. 6 (2) (2022). https://doi.org/10.1145/3530895

[20] X. Geng, H. Liu, OpenLLaMA: An Open Reproduction of LLaMA, 2023, https://github.com/openlm-research/open_llama.

[21] W.X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al., A survey of large language models, arXiv preprint arXiv:2303.18223 (2023).

[22] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, B. Catanzaro, Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, 2020, arXiv:1909.08053

[23] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, C. Guo, A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters, in: 14Th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 2020, pp. 463–479.

[24] AliyunContainerService, AliyunContainerService/gpushare-scheduler-extender: GPU Sharing Scheduler for Kubernetes Cluster, 2023, (accessed: 2024-07-30), https://github.com/AliyunContainerService/gpushare-scheduler-extender.

[25] B. Wu, Z. Zhang, Z. Bai, X. Liu, X. Jin, Transparent GPU sharing in container clouds for deep learning workloads, in: 20Th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), USENIX Association, Boston, MA, 2023, pp. 69–85. https://www.usenix.org/conference/nsdi23/presentation/wu.

[26] Y. Zhao, X. Liu, S. Liu, X. Li, Y. Zhu, G. Huang, X. Liu, X. Jin, MuxFlow: Efficient and Safe GPU Sharing in Large-Scale Production Deep Learning Clusters, 2023, arXiv:2303.13803

[27] J. Gu, M. Chowdhury, K.G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, C. Guo, Tiresias: a GPU cluster manager for distributed deep learning, in: 16Th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), USENIX Association, Boston, MA, 2019, pp. 485–500. https://www.usenix.org/conference/nsdi19/presentation/gu.

[28] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, S. Chawla, Themis: fair and efficient GPU cluster scheduling, in: 17Th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), USENIX Association, Santa Clara, CA, 2020, pp. 289–304. https://www.usenix.org/conference/nsdi20/presentation/mahajan.

[29] H. Albahar, S. Dongare, Y. Du, N. Zhao, A.K. Paul, A.R. Butt, Schedtune: a heterogeneity-Aware GPU scheduler for deep learning, in: 2022 22Nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2022, pp. 695–705. https://doi.org/10.1109/CCGrid54584.2022.00079

[30] Y. Jiang, Y. Xiong, L. Qu, C.L. Luo, C. Tian, P. Cheng, Y. Xiong, Moneo: monitoring fine-Grained metrics nonintrusively in AI infrastructure, SIGOPS Oper. Syst. Rev. 56 (1) (2022) 18-25. https://doi.org/10.1145/3544497.3544501

[31] NVIDIA, Feature Overview - NVIDIA DCGM Documentation latest documentation, 2025, (accessed: 2025-02-17), https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/feature-overview.html#profiling-metrics.

[32] Y. Zhao, Y. Liu, Y. Peng, Y. Zhu, X. Liu, X. Jin, Multi-Resource interleaving for deep learning training, in: Proceedings of the ACM SIGCOMM 2022 Conference, SIGCOMM '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 428-440. https://doi.org/10.1145/3544216.3544224

[33] M.A. Hall, Correlation-based feature selection for machine learning, Ph.D. thesis, The University of Waikato, 1999.

[34] G. Chandrashekar, F. Sahin, A survey on feature selection methods, Comput. Electric. Eng. 40 (1) (2014) 16–28. https://www.sciencedirect.com/science/article/pii/S0045790613003066.

[35] Z. Zhao, L. Wang, H. Liu, Efficient spectral feature selection with minimum redundancy, Proc. AAAI Conf. Artif. Intell. 24 (1) (2010) 673–678. https://doi.org/10.1609/aaai.v24i1.7671

[36] H. Jin, Q. Song, X. Hu, Auto-keras: an efficient neural architecture search system, in: Proceedings of the 25Th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 1946-1956. https://doi.org/10.1145/3292500.3330648

[37] R.J. Hyndman, A.B. Koehler, Another look at measures of forecast accuracy, Int. J. Forecast. 22 (4) (2006) 679–688. https://doi.org/10.1016/j.ijforecast.2006.03.001

[38] R.J. Hyndman, A.B. Koehler, Another look at measures of forecast accuracy, Int. J. Forecast. 22 (4) (2006) 679–688.

[39] K. Authors", Scheduler Configuration | Kubernetes, 2024, (accessed: 2024-11-18), https://kubernetes.io/docs/reference/scheduling/config/#scheduling-plugins.

[40] V.K. Vavilapalli, A.C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, E. Baldeschwieler, Apache hadoop YARN: yet another resource negotiator, in: Proceedings of the 4Th Annual Symposium on Cloud Computing, SOCC '13, Association for Computing Machinery, New York, NY, USA, 2013. https://doi.org/10.1145/2523616.2523633

[41] "SymbioticLab", Tiresias open-source trace-driven simulator, 2019, (accessed: 2024-02-09), https://github.com/SymbioticLab/Tiresias.

[42] M.N. Team, Introducing MPT-7B: A New Standard for Open-Source, Commercially Usable LLMs, 2023, (accessed: 2024-05-05), https://www.mosaicml.com/blog/mpt-7b.

[43] E. Almazrouei, H. Alobeidli, A. Alshamsi, A. Cappelli, R. Cojocaru, M. Debbah, E. Goffinet, D. Heslow, J. Launay, Q. Malartic, B. Noune, B. Pannier, G. Penedo, Falcon-40b: an open large language model with state-of-the-art performance (2023).

[44] T. Computer, RedPajama-INCITE-7B-Base, 2023, (accessed: 2024-11-05), https://huggingface.co/togethercomputer/RedPajama-INCITE-7B-Base.

[45] H.W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, Y. Li, X. Wang, M. Dehghani, S. Brahma, et al., Scaling instruction-finetuned language models, arXiv preprint arXiv:2210.11416 (2022).

[46] G. Klein, Y. Kim, Y. Deng, V. Nguyen, J. Senellart, A.M. Rush, OpenNMT: Neural Machine Translation Toolkit, 2018, arXiv:1805.11462

[47] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, T.B. Hashimoto, Stanford Alpaca: An Instruction-following LLaMA model, 2023, (https://github.com/tatsu-lab/stanford_alpaca).

[48] T. Um, B. Oh, B. Seo, M. Kweun, G. Kim, W.-Y. Lee, Fastflow: accelerating deep learning model training with smart offloading of input data pipeline, Proc. VLDB Endowment 16 (5) (2023) 1086–1099.

[49] J. Li, H. Xu, Y. Zhu, Z. Liu, C. Guo, C. Wang, Lyra: elastic scheduling for deep learning clusters, in: Proceedings of the Eighteenth European Conference on Computer Systems, 2023, pp. 835–850.

[50] C. Espenshade, R. Peng, E. Hong, M. Calman, Y. Zhu, P. Parida, E.K. Lee, M.A. Kim, Characterizing training performance and energy for foundation models and image classifiers on multi-instance GPUs, in: Proceedings of the 4Th Workshop on Machine Learning and Systems, 2024, pp. 47–55.

[51] M. Lee, S. Seong, M. Kang, J. Lee, G.-J. Na, I.-G. Chun, D. Nikolopoulos, C.-H. Hong, ParvaGPU: efficient spatial GPU sharing for large-scale DNN inference in cloud environments, in: SC24: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2024, pp. 1–14.

[52] A. Sergeev, M.D. Balso, Horovod: fast and easy distributed deep learning in TensorFlow, 2018, arXiv:1802.05799

[53] W. Gao, Z. Ye, P. Sun, Y. Wen, T. Zhang, Chronus: a novel deadline-Aware scheduler for deep learning training jobs, in: Proceedings of the ACM Symposium on Cloud Computing, SoCC '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 609-623. https://doi.org/10.1145/3472883.3486978

[54] D. Gu, Y. Zhao, Y. Zhong, Y. Xiong, Z. Han, P. Cheng, F. Yang, G. Huang, X. Jin, X. Liu, Elasticflow: an elastic serverless training platform for distributed deep learning, in: Proceedings of the 28Th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Association for Computing Machinery, New York, NY, USA, 2023, p. 266-280. https://doi.org/10.1145/3575693.3575721

[55] B.T. Atmaja, M. Akagi, Deep multilayer perceptrons for dimensional speech emotion recognition, in: 2020 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), IEEE, 2020, pp. 325–331.

[56] A. Kadra, M. Lindauer, F. Hutter, J. Grabocka, Well-tuned simple nets excel on tabular datasets, Adv. Neural Inf. Process. Syst. 34 (2021) 23928–23941.

[57] F. Strati, X. Ma, A. Klimovic, Orion: interference-aware, fine-grained GPU sharing for ML applications, in: Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 1075-1092. https://doi.org/10.1145/3627703.3629578

[58] X. Zhao, M. Jahre, L. Eeckhout, HSM: A hybrid slowdown model for multitasking GPUs, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 1371-1385. https://doi.org/10.1145/3373376.3378457