Latency-guaranteed Co-location of Inference and Training for Reducing Data Center Expenses

Guoyu Chen, Srinivasan Subramaniyan, and Xiaorui Wang
The Ohio State University, Columbus, Ohio, USA
{chen.9605, subramaniyan.4, wang.3596}@osu.edu

***Abstract*—Today's data centers often need to run various machine learning (ML) applications with stringent SLO (Service-Level Objective) requirements, such as inference latency. To that end, data centers prefer to 1) over-provision the number of servers used for inference processing and 2) isolate them from other servers that run ML training, despite both use GPUs extensively, to minimize possible competition of computing resources. Those practices result in a low GPU utilization and thus a high capital expense. Hence, if training and inference jobs can be safely co-located on the same GPUs with explicit SLO guarantees, data centers could flexibly run fewer training jobs when an inference burst arrives and run more afterwards to increase GPU utilization, reducing their capital expenses.**

In this paper, we propose GPUColo, a two-tier co-location solution that provides explicit ML inference SLO guarantees for co-located GPUs. In the outer tier, we exploit GPU spatial sharing to dynamically adjust the percentage of active GPU threads allocated to spatially co-located inference and training processes, so that the inference latency can be guaranteed. Because spatial sharing can introduce considerable overheads and thus cannot be conducted at a fine time granularity, we design an inner tier that puts training jobs into periodic sleep, so that the inference jobs can quickly get more GPU resources for more prompt latency control. Our hardware testbed results show that GPUColo can precisely control the inference latency to the desired SLO, while maximizing the throughput of the training jobs co-located on the same GPUs. Our large-scale simulation with a 57-day real-world data center trace (6500 GPUs) also demonstrates that GPUColo enables latency-guaranteed inference and training co-location. Consequently, it allows 74.9% of GPUs to be saved for a much lower capital expense.

I. Introduction

In the past decade, Machine Learning (ML), as well as Deep Learning (DL), has been successfully applied in a wide variety of applications. To facilitate ML computing, new cloud services have been generated, such as ML-as-a-Service (MLaaS) or more generally AI-as-a-Service (AIaaS). For those new cloud services, service providers (e.g., Google, Amazon, or Microsoft) often need to help customers perform both model training and inference processing on large numbers of GPUs and servers. Model training is often conducted in an offline matter and may take a long time to finish without strict latency requirements [1], [2], [3], [4], [5]. On the other hand, inference requests must be processed online in real time with stringent timeliness requirements (e.g., hundreds of milliseconds per request) [6], [7], [8], [9], [10]. Such requirements are often referred to as the Service-Level Objectives (SLOs) in terms of processing latency [11]. Violating SLOs can result in undesired customer and financial loss and severely hurt the service provider's reputation.

Online ML inference requests are bursty by nature. A large number of inference requests (as workload bursts) may often need to be simultaneously handled by an entire data center (i.e., the computing infrastructure behind the cloud). For example, during a large sports event like the FIFA world cup, ML inference requests, such as image recognition (for security or admission), can increase sharply right before the beginning of the games and decrease quickly after the games. More recently, due to the rapidly increasing attention and request bursts, ChatGPT [12] is often found to be "*at capacity*", which can be frustrating to its users. Despite existing research on ML acceleration [13], [14], [15], [16], [17], [18], how to handle the burstiness of ML requests in a cost-efficient way remains an open research problem for data center operators. Currently, in order to deal with such bursty workloads with latency guarantees, a data center often has to over-provision their inference servers, just to prepare for the worst cases that only occur occasionally. Such over-provisioning can result in an unnecessarily high capital expense (CapEx) for data centers and is highly undesirable.

In addition to server over-provisioning, data centers often like to use separate server clusters for ML training and inference [19], despite both use GPUs extensively. The key reason is to minimize the risk of inference SLO violation caused by the GPU resource competition from training. For example, a recent study [19] reports that a major online service provider currently runs more than 10K GPUs for training, and at least 5 times as many GPUs for inference. Because of the burstiness of the inference workloads observed by this provider, they do not want to risk sharing servers between training and inference, which results in a low GPU and server utilization. In another empirical study [20], their analysis of 6000 GPUs in a production MLaaS cluster for two months show that most inference servers have low GPU utilization (less than 18% on average). Hence, if training and inference workloads could be *safely* co-located on the same GPUs and servers with explicit SLO guarantees, data centers could flexibly run more/fewer training jobs when the demands for inference become lower/higher. Then, when a burst of inference requests arrives, more GPU resources on the same servers can be dynamically allocated to inference jobs for ensuring latency. As a result, data centers would not need to

This paper is based upon work supported by the U.S. National Science Foundation under award CNS-2336886.

purchase an unnecessarily large number of GPUs and servers and thus can have a much lower CapEx.

How to effectively co-locate multiple processes (e.g., inference and training) on the same GPU is non-trivial, because today's GPUs are mostly designed to exclusively run a single process at any give time. There are currently two different ways to share GPU resource between different processes: 1) temporal sharing and 2) spatial sharing. Traditional temporal sharing refers to the serialization of different processes on the GPU. For example, after an ML training process is finished, we can start to run another training process on the same GPU by loading its DNN models into the GPU memory. The second way, spatial sharing, refers to running multiple processes simultaneously on the same GPU. Examples are Nvidia's Multi-Process Service (MPS) [21] and AMD's CU Masking [22]. For instance, MPS facilitates the compute kernels submitted from different CPU processes to execute on the same GPU at the same time, thus allowing GPU sharing and better GPU utilization. Specifically, within one Nvidia GPU, the computation resources are represented as threads scheduled by Streaming Multiprocessors (SMs). MPS provides a way to provision the maximum percentage of active threads of each CUDA context, which is created when the workload process starts. Compared with temporal sharing, a recent study shows that spatial sharing can lead to higher GPU utilization [7]. However, though MPS has facilitated GPU spatial sharing, co-locating inference and training processes remains challenging due to the risk of inference SLO violation, because MPS itself is not designed to be aware of any SLO for any latency guarantees.

In this paper, in order to provide explicit SLO guarantees for ML inference requests, we propose GPUColo, a two-tier co-location solution that dynamically adjusts the GPU resources allocated to co-located inference and training processes. In the outer tier, if the measured inference latency becomes longer than the SLO, we exploit existing spatial GPU sharing techniques (e.g., MPS) to dynamically reduce the percentage of active GPU threads allocated to the co-located training processes, so that the inference process can get more GPU resources and run faster to shorten its latency. However, for dynamic GPU resource re-allocation, spatial sharing, like MPS, can often introduce considerable overheads (i.e., seconds of process restarting time) and so cannot be conducted at a fine time granularity. Thus, we design an inner tier that puts training jobs into periodic sleep, so that the inference jobs can quickly get more GPU resources for more prompt latency control. But, as shown later in Section IV-B, periodic sleep can result in lower GPU utilization and has a limited adaption range, so MPS still needs to be performed over a longer time period for better GPU resource utilization. Both our hardware testbed and trace-driven large-scale simulation results show that GPUColo can enable latency-guaranteed inference and training co-location and thus significantly reduces CapEx.

Specifically, this paper makes the following contributions:

- We observe that the current practice uses isolated server clusters for ML inference and training jobs, due to the concerns of inference SLO violation. Thus, we propose to dynamically co-locate inference and training on the same GPUs and servers, in order to increase GPU utilization for reducing data center CapEx.
- In order to provide the desired SLO guarantees, we design GPUColo, a two-tier inference latency control solution that dynamically increases the GPU resource allocated to inference processes when a burst of inference requests arrives and decreases it afterwards to allow a better throughput of ML training. GPUColo conducts both spatial sharing for more efficient co-location and lightweight sleep interval adaption for more prompt latency control with negligible overheads. The trade-offs and design choices are discussed in Section IV.
- We perform experiments on a hardware testbed to show that GPUColo provides the desired inference latency control and achieves higher inference and training throughput than the state-of-the-art solutions and other well-designed baselines. Our large-scale simulation with a 57-day real-world trace from a major cloud service provider also shows that GPUColo outperforms the baselines by saving 74.9% of GPUs for a much lower CapEx.

II. Related Work

MLaaS and AIaaS are becoming more and more widely adopted in the recent years. In the industry, TensorRT [23], TensorFlow Serving [24], and Amazon SageMaker [25] are currently among the major ML model serving systems, but they do not provide any mechanisms to meet the application SLOs. Instead, they require users to self-tune the ML model parameters (e.g., batch size) and configure the hardware to meet the SLOs, which can be difficult for some users.

Latency of training jobs. A lot of recent research studies have tried to address the ML latency issues [1], [3], [4], [18], [17], [9], [26], [5]. Many of those studies focus on accelerating the training processes. For example, Optimus [17] presents an online fitting method to allocate resources based on the speed of ML training jobs. Hypersched [4] increases the training job accuracy within a deadline using automatic hyperparameter search and dynamic resource allocation. SLAQ [18] allocates more CPU-based cluster resources to ML training jobs whose loss convergence rates are higher. Tiresias [1] schedules distributed deep learning jobs on a GPU cluster to reduce job completion time and increase GPU utilization. Themis [3] presents a two-level scheduling algorithm that allows ML workloads to bid on GPU resources based on finish-time fairness. In contrast to those studies, GPUColo focuses on co-location and meeting the *inference* latency SLOs, while also maximizing the training throughput.

Inference latency SLO has also been studied in some recent research [8], [6], [27], [10], [7], [28], [19], [29]. For example, Nexus [8] allocates ML inference workloads in a GPU cluster for SLO control. Lyra [29] proposes cluster-level capacity loaning to reduce ML job completion time (JCT). Clipper [6] changes the batch size of ML inference workloads using a fixed percentage in each step until the SLO is met.

DART [27] schedules DNN inference workloads on a heterogeneous multi-core system for latency guarantees. LaLaRAND [10] is a layer-by-layer resource allocation scheme for DNNs to meet inference latency requirements. OptimML [28] tries to meet both ML inference latency and server power constraints. GPUColo differs significantly from those studies that focus only on inference jobs, because we co-locate both inference and training jobs and control the inference latency on co-located GPUs.

ML job co-location has recently received increasing research attention [2], [9], [19], [26], [29], [7], but existing studies co-locate *either inference or training jobs, in a separate manner*. For training co-location, Gandiva [2] designs a cluster scheduling framework to co-locate training jobs on GPUs, with the considerations of memory and communication intensity, to minimize the interference among different jobs. For inference co-location, PipeSwitch [19] uses a pipeline to schedule model transmission and execution on a GPU to co-locate inference jobs with temporal sharing for fast context switch. Choi et al. [26] have applied MPS to co-locate inference workloads spatially on a GPU based on offline profiling data of ML models. However, all those studies do not provide explicit inference SLO guarantees. The most closely related work is GSLICE [7], which co-locates *only* inference jobs and meets the inference SLO by dynamic GPU resource allocation. *To our best knowledge, GPUColo is the first study that proposes to co-locate both inference and training jobs on the same GPUs for significantly reducing CapEx. GPUColo is also the first to use periodic sleep of training jobs with MPS for a two-tier control solution with acceptable runtime overheads.*

III. Motivation

To motivate the proposed GPU co-location and control solution, we conduct some experiments with a simple example on our hardware testbed (detailed in Section VI), to demonstrate the problems of existing solutions and how the proposed co-location solution can improve them. Our simple example has two inference jobs and two training jobs that are released at the beginning. Our first inference job $I1$ emulates the cloud-based speech-to-text conversion application in [30] with a typical latency (*i.e.*, duration when running alone) of 31s and an SLO of 56s, while the second inference job $I2$ emulates the online video moderation application in [31] with a latency of 36s and an SLO of 61s. Usually, training jobs can last for hours and require much more memory than inference jobs. However, to keep the example simple, we configure each training job to run about 52s and require a memory size about 3 times that of inference. The detailed memory size and GPU utilization of each job are listed in Table I. We assume two GPUs, each of which has 24GB of memory. We now introduce four baselines.

No Co-location is a state-of-the-practice solution used in today's data centers. It first divides all the GPUs and servers into an inference cluster and a training cluster. Because GPUs are traditionally designed to run a single process, each GPU is used to run only a single job at any given time with traditional temporal sharing. Based on this policy, we use GPU1 to run the two inference jobs sequentially, resulting in a 67s total running time. We use GPU2 to run the two training jobs sequentially, and so the total duration is 104s. This result is shown in Figure 1 (upper left).

TABLE I: Job details in the simple motivation example.

Job type	Inference		Training	
Job name	I1	I2	T1	T2
Needed GPU utilization (%)	50	50	50	50
Needed memory size (GB)	2.5	2.5	7	7
Duration when running alone (s)	30	35	52	52
Inference latency SLO (s)	56	61	N/A	N/A

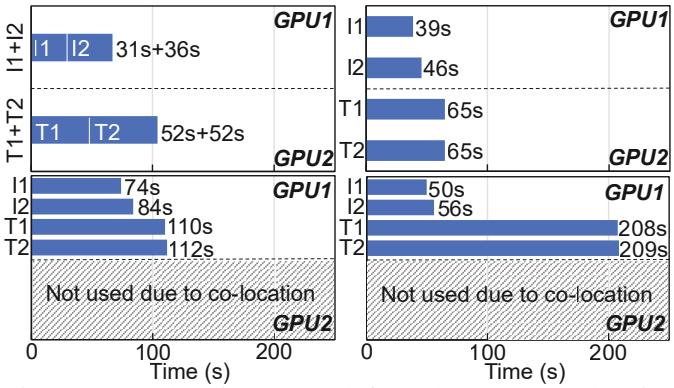


Fig. 1: No co-location (upper left) and Separate co-location (upper right) both need to use two GPUs. Simple co-location (lower left) uses one GPU but misses the inference SLO. Conservative co-location (lower right) uses one GPU and meets the SLO, but leads to long training time.

Separate Co-location is a state-of-the-art solution that is similar to the co-location strategy proposed in [7]. As mentioned before, spatial sharing can result in better GPU utilization than temporal sharing. Hence, jobs can be co-located with spatial sharing on *separate* inference and training GPU clusters. In the process of co-location, the total memory size and GPU utilization should not exceed the capacities of a single GPU (i.e., 24GB memory, 100% utilization). The utilization of each GPU is divided *evenly* among all the co-located jobs. For example, in the simple example, the two inference jobs can each have 50% on GPU1, while the two training jobs can also each have 50% on GPU2. Because each job only needs 50% of GPU utilization, the two jobs on either GPU can run concurrently with spatial sharing without being slowed down due to resource competition. As a result, the jobs can finish much earlier than *No Co-location*. Specifically, the two inference jobs finish by 46s (31% improvement) and the training jobs finish by 65s (37% improvement), as shown in Figure 1 (upper right). Note that the actual duration of each job is longer than their duration numbers in Table I due to the overhead of MPS spatial sharing.

It can be observed that both *No Co-location* and *Separate Co-location* keep inference GPUs separate from training GPUs, because they try to minimize the GPU resource competition from training that might impact the inference latency. However, such a limitation always results in that both GPUs must be used. If we could co-locate both inference and training jobs on the same GPU, we may be able to use

just one GPU instead of two for a lower capital expense. Unfortunately, doing so is non-trivial because sophisticated designs are needed to control the inference latency while maximizing the training throughput. In the following, we test two other baselines that rely on fixed GPU resource allocation for inference and training co-location, to demonstrate common strategies would not result in desired co-location.

Simple Co-location represents a common strategy that mixes inference and training jobs and evenly divides the GPU resource among all co-located jobs on the same GPU. Although *Separate* also evenly divides GPU resource, it does that separately to inference and training jobs in their respective clusters. For the simple example, because the aggregated memory size of all the 4 jobs is smaller than 24GB, *Simple* tries to co-locate all the jobs on one single GPU and evenly allocates 25% GPU utilization to each job. However, since each job demands for 50% GPU utilization, such a co-location would slow down every job. As shown in Figure 1, *Simple* has missed the SLO requirements of 56s for $I1$ and 61s for $I2$ because of its strategy of giving each co-located job an equal share of the GPU resource. One might argue that we could favor inference jobs more by allocating them more GPU resource in a conservative way, which should increase the chance for inference jobs to meet their SLOs.

Conservative Co-location is similar to *Simple* but allocates 40% of the GPU utilization to each inference job and 10% to each training job, totaling in 100% GPU utilization after co-locating the four jobs in Table I. Because each job's demanded GPU utilization is 50%, there will also be slowdown for each job. The result is shown in Figure 1 (lower right). We can see *Conservative Co-location* has an inference latency that is indeed shorter than the SLOs this time, but at the cost of a long training job duration that is about twice that of *Simple*. Such a long training duration is undesired because it can lead to GPU resource waste, service fee increases (for customers), and higher energy costs.

Both *Simple* and *Conservative* can manage to use one GPU with spatial sharing, which helps reduce the data center CapEx. However, a natural problem with spatial sharing is *how to properly allocate GPU resource among co-located jobs*. Both *Simple* and *Conservative* rely on *fixed allocation* without considering the SLO, which results in either SLO violation or a long training duration. Hence, we argue that it can be commonly difficult to have a perfect fixed allocation ahead of time, especially when new jobs can arrive later as a burst. A better strategy should be to dynamically allocate the GPU resource, such that the latency of inference jobs can be controlled exactly to the SLOs, while the throughput of the training jobs can be maximized.

SLO-aware Co-location is an ideal solution that is similar to our proposed solution (without considering overheads). It dynamically adjusts the GPU resource allocation based on the measured inference latency. To facilitate latency control, the SLO of each inference job is divided by its number of batches to have a desired latency for each batch. For example, for $I1$, if it has 100 batches, the desired batch latency SLO is $56/100 = 0.56s$. *SLO-aware Co-location* runs periodically and each of its period includes multiple inference batches. In each period, it measures the average batch latency and compares with the desired batch SLO. If the measured latency is longer than the SLO, more GPU resource is dynamically allocated to the inference job to shorten its latency. If the measured latency is shorter than the SLO, more GPU resource is given to the training jobs to increase their throughput. Hence, *SLO-aware* can start with any initial GPU allocation and settle to the best allocation quickly by adjusting the allocation at runtime.

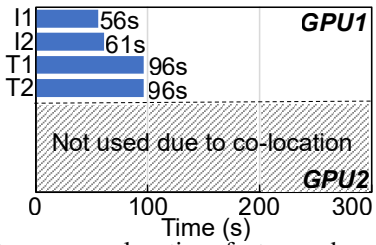


Fig. 2: SLO-aware co-location features dynamic inference latency control (precisely to the desired SLO), which can result in a much shorter training duration.

Figure 2 shows the *numerical analysis result* of *SLO-aware Co-location* as a motivation before we actually implement it. *SLO-aware* can precisely control the two inference jobs to their respective SLOs (*i.e.*, 56s and 61s), while having a training duration (*i.e.*, 96s) much shorter than that of *Conservative*, as shown in Figure 2. Therefore, even though *SLO-aware*, *Simple*, and *Conservative* all can use just one GPU to reduce CapEx, the **dynamic GPU resource adjustment used by *SLO-aware* can have at least two advantages**. First, it can automatically find the best GPU resource allocation by controlling the inference latency precisely to the SLOs. Second, after the two inference jobs finish, *SLO-aware* can dynamically increase the GPU utilization of each training job to 50%, leading to a shorter duration (*i.e.*, 96s). In contrast, *Simple* and *Conservative* still keep it at the initial values of 25% and 10% for each training job, because they do not dynamically change the GPU resource allocation, which is why *SLO-aware* has even a shorter training duration than *Simple*. Those observations have motivated our design of an inference latency controller. Note that dynamic GPU resource adjustment does have one disadvantage because such runtime adjustment commonly has overheads, which must also be carefully considered in our design.

IV. System Design

In this section, we provide a high-level description of the GPUColo system architecture. As discussed before, a significant concern of data center operators is the violation of SLO guarantees. Thus, they tend to 1) separate server clusters used for ML training from those used for inference in order to minimize the competition of computing resources, and 2) over-provision inference servers to prepare for the worst-case scenarios where a large number of inference requests come simultaneously as a burst. Such practices result in low GPU and server utilization and thus a high CapEx. Hence, if training

and inference jobs could be safely co-located on th GPUs and servers with explicit SLO guarantees, data could flexibly run more/fewer training jobs on inferenc when there are fewer/more inference requests, resu fewer GPUs and servers and thus a lower CapEx.

We focus on a typical scenario where a data cent mostly ML or DL workloads needs to run both train inference jobs. Training jobs can be scheduled at a by the data center operator and do not have strict tin requirements, though a high throughput is preferred. In requests have stringent SLO requirements and can continuously received online. The number of reque vary significantly at different times. A job dispatcher to dynamically distribute the incoming training and in jobs to the GPUs on different servers in the data ce shown in Figure 3. We adopt a simple first-fit bin scheme for dynamic job dispatching, which packs ea job based on its demanded GPU and memory utilization and the available resources of the GPUs. It is important to note that job dispatching is not the focus of this paper and our solution can work with other dispatching algorithms as well (which is our future work). The main design contribution of our work is to ensure that the co-located inference jobs meet their latency SLOs even when a burst of requests may arrive at any time, because this is one of the most important concerns of today's data center operators.

At any time, each GPU in the data center can be running three types of job combination: 1) both inference and training, co-located together, 2) either inference or training, alone, and 3) idling or turned off, without any jobs. The proposed latency control solution will be invoked only on those GPUs with co-located jobs. On such GPUs, computing resources are shared between the ML inference and training processes. Because spatial sharing has better GPU utilization [7], we assume that inference and training are co-located with spatial sharing.

A. Overall Architecture

The design objective of GPUColo is to dynamically control the ML inference latency for SLO guarantees on those GPUs that have training and inference jobs co-located. Specifically, GPUColo monitors the average latency of online ML inference requests and compare it with the desired SLO. If the measured latency is longer than the SLO, GPUColo will dynamically allocate more GPU resources to those co-located inference jobs such that they can run faster to have a shorter latency. If the measured latency is shorter than the SLO, more GPU resources are allocated to training jobs for a higher throughput. In the extreme cases, when a huge burst of inference requests arrives, GPUColo can flexibly turn all the GPUs in the data center to run just inference jobs. When no inference request arrives, all the GPUs can run just training jobs. Such a flexibility allows GPUs to be shared for a much higher utilization, resulting in a lower CapEx.

In GPUColo, we deploy one controller on every co-located GPU for the consideration of scalability in a large data center. In this strategy, one controller monitors the average latency

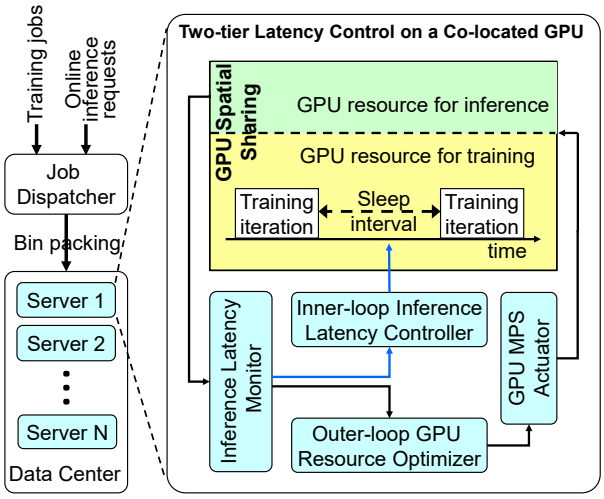


Fig. 3: System Architecture. GPUColo features a two-tier control design. The inner loop controls the inference latency by adjusting the sleep interval of training jobs, while the outer loop optimizes the GPU resource utilization with MPS.

of all the inference tasks on one GPU and adjusts the GPU resource allocated to all the co-located jobs on the same GPU. The GPUColo controller on a GPU is dynamically created whenever the GPU becomes a co-located GPU, i.e., dispatched with both inference and training jobs. It runs periodically to monitor the average latency of all the inference jobs (as controlled variable) on this co-located GPU. The controller dynamically adjusts the GPU resources allocated to both inference and training jobs (as manipulated variable), so that the average latency can converge to the desired SLO.

Note that GPUColo can be applied to GPUs that run distributed training and need synchronization between different GPUs. In that case, if a distributed training task is slowed down by GPUColo, other GPUs that need to synchronize with this task can have their distributed training tasks temporarily suspended. Consequently, those GPUs may be dispatched with more inference tasks or other training tasks.

B. Knobs for GPU Resource Allocation

An important design choice is how to dynamically adjust the computing resources of a GPU. In this paper, we adopt Nvidia's MPS for spatial sharing as an example due to its reported better GPU utilization, *but it is important to know that GPUColo can also be applied to other spatial sharing techniques like AMD's CU Masking [22], which performs similarly to MPS*. As mentioned before, MPS provisions the maximum percentage of active threads of a GPU process *when the process starts*. In other words, MPS is not designed to dynamically adjust the GPU resource of a process in the middle of its running. Hence, if MPS must be used to make adjustment at runtime for a process, the process has to be stopped and restarted with the new MPS percentage. Due to required data copying when a process restarts, dynamically adjusting MPS can introduce an overhead of 5-15 seconds based on our testbed measurements. Therefore, in order not to interrupt any inference jobs that have relatively short durations,

we choose to keep the MPS percentage of inference jobs fixed at 100% and adjust the MPS of co-located training jobs between 1-100%, because training can take hours to finish. Note that MPS allows each process to individually have an MPS cap up to 100%. When the aggregated MPS caps of all jobs exceed 100%, the GPU is oversubscribed, which can result in slowdown of every process. Hence, MPS caps are relative among processes and increasing the cap of a process can help it get more GPU resource, as to be discussed later.

Due to the considerable overhead of MPS adjustment, we propose another lightweight method to promptly adjust the GPU resource among co-located concurrent inference and training jobs. The second knob we choose is to put the training jobs into periodic sleep. For example, after each iteration of a training process (approximately 0.4 to 0.8s for most workloads), the training process can be put into sleep for about 0.2 to 0.8 seconds. Based on our studies, when training jobs are put into sleep, GPU thread resources can be temporarily released and allocated to the spatially co-located inference jobs for a shorter latency. Periodic sleep can be implemented in practice in different ways. If the byte code of the training workload is available, a sleep function can be inserted as part of instrumentation. If the byte code is not available, a more general way is to pause the training process using signal SIGSTOP and resume it after the sleep interval using signal SIGCONT. Such a general method takes less than 5ms to pause/resume a GPU process based on our testbed measurements.

In order to fully understand the differences between MPS and periodic sleep, we now perform the following experiments on our hardware testbed (setup introduced in Section VI). To our best knowledge, our study is the first detailed comparison between MPS and periodic sleep. In this experiment, we launch two concurrent tasks on a Nvidia RTX3090 GPU. The inference task runs ResNet-50 while the second task runs an ML training workload. The purpose is to examine the impacts on the inference latency if we use MPS or sleep interval to adjust the GPU resources allocated to training. We test a typical scenario where we reduce the GPU resource of training jobs in the middle of an experiment and increase it back later. The inference latency and the number of active training SMs are measured to show how they change when MPS or sleep interval is adjusted. Note that the number of active SMs is blocked by the PyTorch framework, so we emulate DNN training using some matrix computations in CUDA and any needed data copying is emulated too. The SM numbers are sampled in each training iteration using the *smid* function provided in the Nvidia PTX library.

Figure 4 shows the results of three experiments, where MPS, sleep interval, and both MPS and sleep interval are tested as "MPS", "Sleep", and "Sleep+MPS", respectively. For "MPS" (blue curves with square markers), spatial sharing is used for the two tasks with the same 80% MPS thread percentage. When the MPS for training is reduced to 20% at 20s, the training SM percentage is then reduced from 38.15% to 1.128%. As a result, more GPU resource is allocated to the

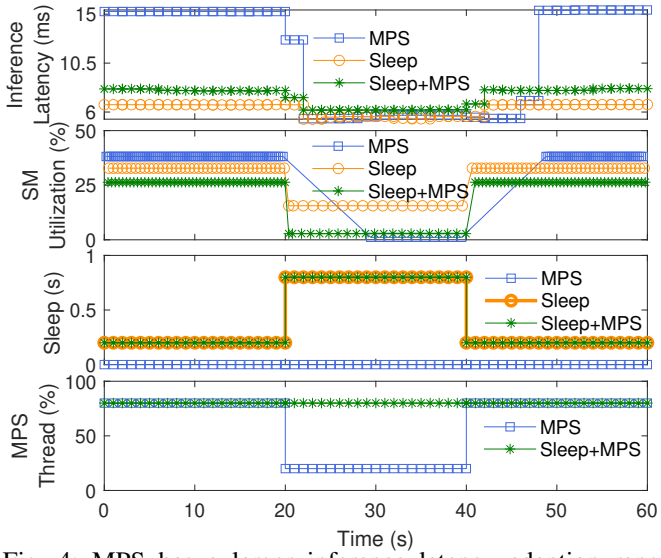


Fig. 4: MPS has a larger inference latency adaption range but a much higher overhead than sleep interval. Sleep+MPS provides a good trade-off to have better GPU co-location.

inference task whose latency is then reduced from 15.1ms to 5.62ms. It is important to note that MPS has an overhead of 7s in this case to stop and restart the training process with the new MPS value, as discussed before. That is why there are no readings of training SM percentage between 20s and 27s, but the inference process is still running in the meantime and its latency can be measured. Hence, MPS adaption cannot be conducted at a fine time granularity. When the training MPS is increased back to 80% at 40s, more GPU resource is allocated back to training, resulting in the increase of inference latency back to 15.1ms.

For "Sleep" (orange curves with circles), temporal sharing is used without MPS. At the beginning, a sleep interval of 0.2s is used, which means that the training process is put into sleep for 0.2s after every iteration. Since each iteration lasts about 0.2s, it is equivalent to 50% of sleeping time. At time 20s, the sleep interval is increased to 0.8s, causing the training SM percentage to reduce from 32.8% to 15.8% and the inference latency to reduce from 6.68 ms to 5.52 ms. There are two important observations: 1) The overhead of periodic sleep is almost negligible (about 2ms) and much shorter than the 7s MPS overhead. 2) The impact of sleep interval on inference latency is much smaller (with a 6.68-5.52=1.16ms difference) than that of MPS (15.1-5.62=9.48ms). This indicates that 1) sleep interval has a much smaller latency adaption range than MPS, and 2) the efficiency of temporal sharing may not be as high as spatial sharing, because the inference job runs faster only by 1.17ms when the training job sleeps 4X time.

We then test "Sleep+MPS" (green curves with stars), where MPS spatial sharing is enabled for concurrent tasks, but only sleep interval is used for GPU resource adjustment. Similar to MPS alone, both jobs start with the same 80% MPS percentage and a sleep interval of 0.2s is initially used. We can see that the inference latency is slightly longer than "Sleep" because only 80% MPS is allocated to the inference job. At time 20s,

the sleep interval is increased to 0.8s with MPS unchanged. Consequently, the SM percentage reduces from 26.3% to 2.89% and the latency is reduced from 7.95 ms to 6.2 ms. Compared with "Sleep", "Sleep+MPS" has a bigger latency adaption range because MPS is more efficient in co-location.

From this set of comparison, we can conclude that MPS has considerable overhead and so cannot be used for fine-grained GPU resource adjustment. Sleep interval has negligible overhead but has a much smaller latency adaption range and is not as efficient as MPS in GPU utilization. Sleep+MPS offers a good trade-off to have a much smaller overhead and a larger range than Sleep for better GPU co-location. Its adaption range can be even larger when MPS is adjusted together with sleep interval (over a long time period). Those observations motivate us to design a two-tier control solution with sleep interval as the inner loop and MPS as the outer loop.

C. Two-tier Control Solution

As shown in Figure 3, our proposed GPUColo two-tier design has 1) an inner-loop inference latency controller that adjusts the sleep interval of the training jobs to dynamically control the latency of the co-located inference jobs, and 2) an outer-loop GPU resource optimizer that changes the MPS thread percentage of the training jobs, at a much coarser time granularity, to optimize GPU resource allocation. The control period of the inner loop is chosen as follows: 1) It is long enough to process multiple ML inference requests within one period for an averaged latency sample to avoid any outliers. 2) It is short enough for timely reaction to any inference bursts. 3) The actuation overhead is acceptable. Based on the average inference request duration and the 2ms overhead of adjusting sleep interval, we use 4s in our testbed as an example. *Note that the control period is configurable based on the typical inference lengths of a data center.* Because the two loops both monitor and control the same inference latency, the period of the outer loop should be chosen to ensure: 1) The MPS overhead is acceptable. 2) The inner loop can settle down within one period of the outer loop, such that the two loops are decoupled and can be designed independently [32]. We use 100s in our experiments based on the analysis of settling time of the inner loop. In practice, those periods should be chosen based on the workload analysis and overhead measurements of a target data center.

For the inner loop, the following steps are invoked at the end of each control period: 1) The *Inference Latency Monitor* in Figure 3 measures the average inference latency in the last control period and sends it to the *Inference Latency Controller*. Note that in order to allow fine-grained latency control, we measure the average latency of each batch in the inference requests. For example, if the inference SLO is 5s and it has 100 batches, we control the latency of each batch to 50ms. 2) The controller compares the measured batch latency with the desired SLO. Based on the difference, the controller calculates the new sleep interval to be used for the training jobs in the next control period. 3) Finally, the new sleep interval value is enforced by adjusting either the parameter of the sleep function inserted in the byte code or the time interval between calling the SIGSTOP and SIGCONT signals.

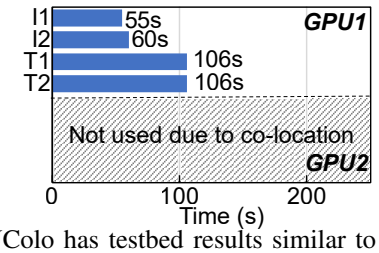


Fig. 5: GPUColo has testbed results similar to SLO-aware in Figure 2. Its two-tier design reduces the MPS overheads.

The *Outer-loop GPU Resource Optimizer* is also invoked periodically but at a much coarser time granularity (e.g., every 25 periods of the inner loop on our testbed), due to the considerable overhead of MPS. The following steps are invoked at the end of every period: 1) The optimizer receives the measured average batch latency from the *Inference Latency Monitor* in the last period after the inner loop settles down. Note the two loops are monitoring the same variable. 2) The optimizer compares the measured latency with the desired SLO. Then, it computes the new MPS percentages to be used for the training jobs in the next period, based on the difference. Because MPS has a high overhead as discussed before, in order to minimize the system actuation overhead, the optimizer computation is invoked only when the difference is bigger than a configurable threshold (e.g., 2%). 3) The new MPS value is sent to the *GPU MPS Actuator* to enforce in the next period. See Section VII for the MPS implementation in our testbed.

In GPUColo, both the inference latency controller and the GPU resource optimizer are designed based on well-established feedback control theory. There are several advantages for such a control-theoretic methodology. For example, the designed system can have guaranteed system stability, better control accuracy, as well as faster convergence and smaller overshoot. Furthermore, the control parameters can be chosen analytically based on control theory, such that the exhaustive manual tuning and testing used in traditional heuristic-based solutions can be avoided. More importantly, when the system models vary online due to workload variations, we can conduct quantitative analysis to ensure stability and control accuracy.

We now compare GPUColo with the ideal solution *SLO-aware* in Section III that does not consider overheads. To reduce the overheads of MPS adjusting and process restarting, GPUColo adopts the aforementioned two-tier control design. Figure 5 shows that GPUColo has slightly higher overheads than *SLO-aware* but indeed can optimize GPU utilization with dynamic GPU resource adjustment.

In the next section, we introduce the design details of the inner loop latency controller. The design of the outer loop is similar and so omitted due to space limitation.

V. Inference Latency Controller

To design the inner-loop inference latency controller, we first need to formulate the control problem. We then establish a mathematical model for the target system between the controlled variable (i.e., inference latency) and the manipulated

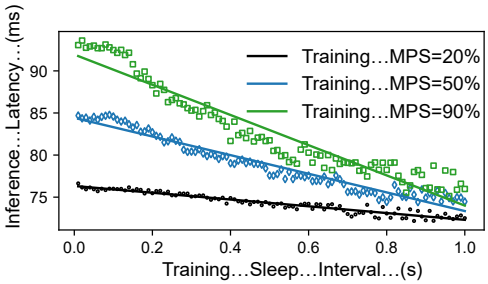


Fig. 6: System model between inference latency and training sleep interval under different training MPS values.

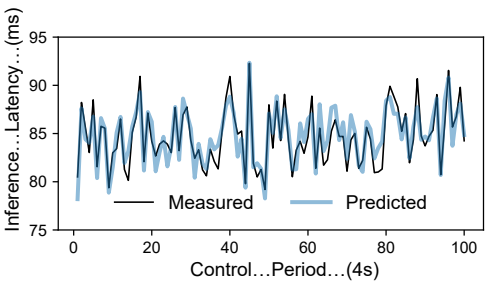


Fig. 7: Comparison between predicted and measured inference latency shows the model is sufficiently accurate.

variable (i.e., sleep interval of training jobs). Based on the system model, we then design the controller and analyze its system stability when the actual model varies due to runtime workload variations.

We first introduce the notation used in this section. T is the control period. L_{ref} is the ML inference latency set point (i.e., SLO). $l(k)$ is the measured average ML inference latency (for a batch) in the k^{th} control period. $e(k)$ is the control error: $e(k) = l(k) - L_{ref}$. $s(k)$ is the sleep interval used for the training jobs in the k^{th} control period. $u(k)$ is the difference between $s(k+1)$ and $s(k)$, i.e., $u(k) = s(k+1) - s(k)$. The goal is to guarantee that the controlled variable $l(k)$ converges to the set point L_{ref} within a finite *settling time*.

A. System Modeling

It is important to model the dynamics of the controlled subsystem for an effective controller design. Unfortunately, a well-established physical equation is usually unavailable for computer systems. Hence, we use a standard approach called *system identification* [33]. Specifically, we run the testbed described in Section VI with different values of $s(k)$ (also under different MPS values), and record the value of $l(k)$ after each control period. Figure 6 plots the relationship between the inference latency and the training sleep interval. We can see that a linear model fits reasonably well for all cases. Based on system identification, we also generate a sequence of pseudo-random digital white noise as control input (*i.e.*, $s(k)$) to stimulate the system and measure the output (*i.e.*, $l(k)$) in each control period.

Figure 7 shows the comparison results between the actual system outputs and the predicted outputs of the linear model. The standard metric $R^2 > 79\%$ indicates that the predicted data is close enough to the measured data from the real system. Note that the controller stability and control accuracy can be guaranteed due to its feedback nature, even if the real system model deviates from the linear model within a certain range. Control-theoretic design methodologies can allow us to analytically derive this range and choose the controller parameters accordingly. Therefore, our system model of ML inference latency is: $l(k) = m \times s(k) + n$, where m and n are the parameters determined by system identification. We can also derive the dynamic model of the inner loop as a difference equation:

$$l(k+1) = l(k) + m \times u(k) \quad (1)$$

B. Controller Design and Analysis

Based on the system model (1), we now apply control theory [33] to design the controller. The design needs to meet the following criteria: 1) Stability: The latency should settle into a bounded range around the SLO set point. 2) Zero steady state error: The latency should be controlled accurately to the SLO. 3) Short settling time: The latency should converge to the SLO within a small number of control periods.

We choose a *Proportional* controller instead of a more complex PID (Proportional-Integral-Derivative) controller because of two reasons: 1) The new training sleep interval to be used in the next control period is $s(k+1) = u(k) + s(k)$, which is already an integral function that helps achieve a zero steady-state error. 2) The derivative term may amplify the noise in ML inference latency. Hence, the time domain form of the designed controller is: $u(k) = \frac{1}{m}(L_{ref} - l(k))$. It is easy to prove that the controller is stable and has zero steady-state error. Due to page limitations, we skip the detailed proofs that can be found in a control textbook [33].

A key advantage of having control theory as a foundation is the analytic assurance of stability and control performance. For example, the real system model can become different from the nominal model (1) used to design the controller, due to workload/hardware variations. Based on the observations of all the possible model variations, we can mathematically analyze the impacts on system stability and control performance by modeling the actual system with variations. Specifically, we first derive the control inputs from the designed controller that use the *nominal model*. We then build the closed-loop system model by substituting the control inputs into the *actual system model*. Finally, we can derive a stability condition of the closed-loop system. Our results show that the system is guaranteed to be stable even when the actual workload/hardware varies within a wide range.

VI. Experimental Setup

Hardware Testbed. Our testbed includes two servers, each with an Intel Xeon R Gold 5215 processor and an Nvidia RTX 3090 GPU. The frequency of the CPU is set at 3.7 GHz, while that of the GPU is set to 2.1 GHz (with a $350W$ power limit). The server OS is Ubuntu 20.04 LTS with CUDA Toolkit 11.6. The ML framework is PyTorch 1.12.1. and the ML workload used for inference and training is Resnet50 and Wide Resnet101, respectively, which are widely adopted in image

classification applications. The inference workloads are configured to a batch size that can vary from 105 to 115. We follow the tutorial in [34] to run the training workload. Each training workload has 100 iterations to run on the testbed. To implement the MPS Actuator, we can configure an environment variable *CUDA_MPS_ACTIVE_THREAD_PERCENTAGE*, based on the value calculated by the GPU resource optimizer, which is limited between 1% and 100%. The training workload is then restarted to enforce the new MPS percentage.

Real-world ML Trace. The PAI trace from Alibaba [20] contains a hybrid of training and inference jobs running the ML algorithms. It is collected from a cluster with over 6500 GPUs in July and August of 2020. Each job record in the trace includes its unique ID, start time, duration, requested GPU memory, requested GPU utilization, and its job status. To facilitate simulation, we filter out jobs whose status is failed or whose GPU memory/utilization is zero. For jobs that have GPU utilization over 100% or GPU memory over 32GB, we follow the common way to split them into sub-tasks with up to 100% utilization and up to 32GB for each sub-task. Because training jobs commonly have higher utilization, larger memory, and longer duration, we use those factors to help define a job as training job, if it is not already marked so in the trace. The PAI trace has been widely used in related work [35], [36], [37], [38], [39], [40].

Trace-driven Simulation. Due to limited size of our hardware testbed, we have developed a simulator in Python to evaluate GPUColo in large scale with the real-world traces. *The simulation accuracy is verified by having the same results with our testbed for those smaller-scale experiments.* Specifically, the simulator includes a simulated clock module, a job dispatcher, a GPU manager, and a controller. The main framework is a *for* loop that simulates 1 second in the trace in one iteration. Before launching, the simulator first sorts all jobs in the trace based on the start time in the ascending order. In each loop, the *job dispatcher* checks the start time of each job and dispatches the job if its start time is equal to or later than the current simulated time. The jobs are dispatched to existing GPUs using the first-fit bin packing algorithm under two constraints: GPU utilization and memory size. If a fit cannot be found, a new GPU is created by the *GPU manager* to host this job. The duration of each dispatched job is deducted in each loop based on its current speed that is determined by its allocated GPU utilization (see Section III for examples), which is in turn decided by the simulated solution. The speed of each training job is also impacted by its sleep interval if the interval is not 0. A finished job is then removed from the job list of the host GPU. Different GPUs can be assigned a different frequency/speed and memory size to simulate a data center with heterogeneous GPUs. To simulate GPUColo, its inner loop algorithm (with a period of 4s) is invoked every 4 iterations to perform the control computation in Section V. The outer loop is invoked every 100 iterations.

Baselines. In addition to the baselines introduced in Section III, i.e., *No Co-location*, *Separate Co-location*, *Simple Co-location*, and *Conservative Co-location*, we also compare GPUColo against a state-of-the-art solution GSLICE [7], which co-locates only inference jobs and meets the inference SLO by dynamically allocating more GPU resource to a job that violates its SLO. GSLICE differs from GPUColo in two major ways: 1) GSLICE keeps inference GPUs separated from training GPUs and so it cannot flexibly use training GPUs for inference when a burst arrives, 2) it relies an ad-hoc method to adapt only MPS, while GPUColo is a control-theoretic solution that features a two-tier design to reduce adaption overheads by putting training tasks into periodic sleep.

The inner loop of GPUColo puts training jobs into periodic sleep with negligible overhead, so it can be invoked every 4s (or shorter) for a prompt response to any workload variations. The outer loop (MPS) has a period of 100s due to the MPS overhead. Both periods are configurable based on the typical workloads. However, GSLICE relies only on MPS, which has an overhead of 5-15s due to the required process restarting. Hence, we cannot run GSLICE every 4s. To determine an appropriate control period for GSLICE, we compute its MPS overhead for different periods from 10s to 100s. We find a period of 91s would give GSLICE and GPUColo approximately the same overheads. However, 91s can be too long a period for ML inference jobs, so we choose 25s that allows GSLICE to respond more promptly.

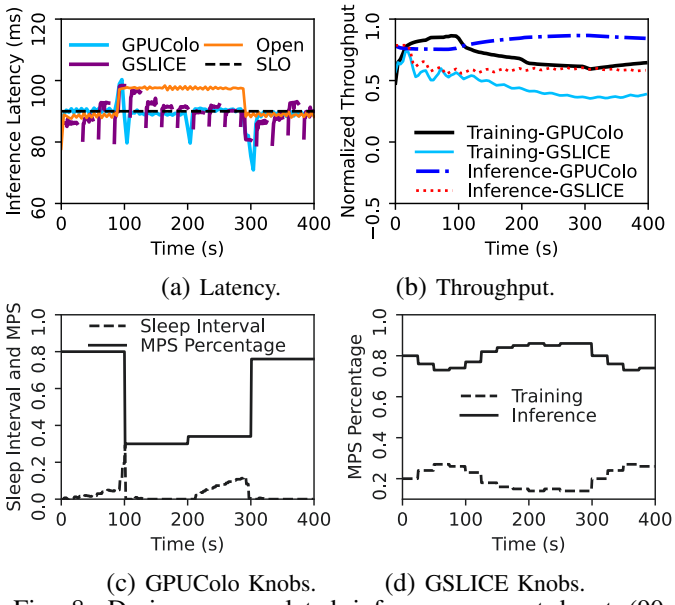
VII. Hardware Testbed Evaluation

We first compare GPUColo against the baseline GSLICE. We then evaluate GPUColo with real-world trace samples.

A. GPUColo Two-tier Control vs. GSLICE

In this experiment, we compare GPUColo and GSLICE when they are used to control one single GPU that co-locates one training job and one inference job. Although GSLICE actually does *not* co-locate training and inference, one might argue that GSLICE could be extended to control a training job just like an inference job. Specifically, GSLICE keeps the total MPS of both jobs at 100% and increases the inference MPS (so reduces the training MPS) if the measured latency becomes longer than the SLO using the algorithm in [7]. In contrast, GPUColo keeps the inference MPS always at 100% and adjusts only the training sleep interval and MPS. The inference latency SLO is set to 90ms for both solutions.

To emulate a typical scenario where the inference workload suddenly increases, the inference batch size is increased from 105 to 115 at 90s and changed back to 105 at 290s. Note that such an increase degree is configured based on the observed workload variations in the PAI traces [20]. As shown in Figure 8a, the open-loop system without any control (i.e., Open) has undesired latency increase as a result of inference request surge and violates the 90ms SLO immediately, until the surge ends at 290s. In contrast, GPUColo reacts by increasing the sleep interval of the training job, as shown in Figure 8c, such that the inference job can promptly get more GPU resource to run faster. Then, at 100s, the outer loop of GPUColo is invoked to optimize the GPU resource by lowering the training MPS from 80% to 30%. As a result of the two-tier control, GPUColo is able to quickly lower the inference latency back to the SLO.



(a) Latency. (b) Throughput.

(c) GPUColo Knobs. (d) GSLICE Knobs.

Fig. 8: During an emulated inference request burst (90s-290s), GPUColo outperforms extended GSLICE by its two-tier control that results in smaller overheads, better control accuracy, and higher inference and training throughput.

In contrast to GPUColo that adjusts only the training MPS, GSLICE adjusts the inference MPS too, causing the inference job to restart every 25s. As a result, it can be observed in Figure 8a that GSLICE has fragmented inference latency values with no latency readings during its restarting times. When the workload increases at 90s, GSLICE reacts by increasing the inference MPS and so decreasing the training MPS, as shown in Figure 8d. GSLICE responds slower than GPUColo because it has a much longer period due to its overhead (25s vs. 4s). As a result of the decreased training MPS, the training throughput is decreased too. Figure 8b shows that, on average, GPUColo has 36.1% higher inference throughput and 51.5% higher training throughput than GSLICE, for two reasons: 1) GSLICE has a much higher overhead because it relies only on MPS, while GPUColo features a two-tier control whose inner loop conducts periodic sleep with negligible overheads. 2) GSLICE adjusts MPS for both inference and training jobs, while GPUColo adjusts only the training job so that the inference throughput is not impacted by restarting.

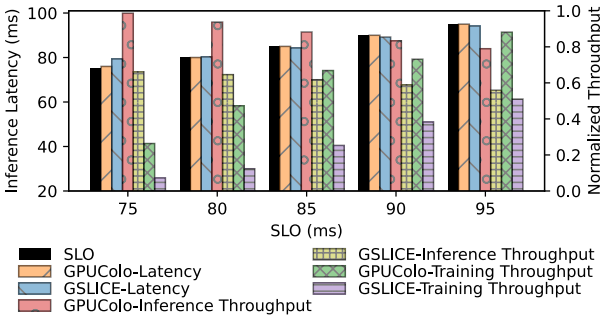


Fig. 9: GPUColo has better latency control accuracy and higher throughput than GSLICE under different SLOs.

To further examine the control accuracy and throughput of the two solutions, we test different SLOs from 75ms to 95ms with a step size of 5ms. We run this experiment with an inference batch size of 105 for 400s and collect the average values of each run by excluding any transient periods. Figure 9 shows that GPUColo accurately controls the latency at the required SLOs, except that its latency is 1ms higher when the SLO is 75ms. In this case, both the training sleep interval and MPS have already been increased to their upper bounds (e.g., 1s and 100%), so the controller saturates and loses its adaption capacity. In contrast, GSLICE often stays unnecessarily lower than the SLO due to its restarting overheads. At 75ms, GSLICE has an even longer latency than GPUColo because it relies only on MPS and so has a smaller adaption range. Figure 9 shows that GPUColo has much higher inference and training throughput than GSLICE due to its two-tier control that has smaller overheads. For example, when the SLO is 80ms, the training throughput of GPUColo is 2.8 times that of GSLICE. Note again that GSLICE is *not* actually designed to control co-located inference and training jobs. It is evaluated here just to show that even an extended GSLICE would not work as well as GPUColo.

B. Evaluation of Real-world Trace Samples

We now test a larger ML job set with 12 jobs sampled from the real-world trace [20] based on their demanded GPU utilization and memory size: 6 inference $\{I1, I2, 3, I4, I5, I6\}$ and 6 training $\{T1, T2, T3, T4, T5, T6\}$. We emulate a typical scenario where inference jobs can dynamically arrive as bursts. As shown in Figure 10, in Burst Stage 1, at time 0s, three training jobs $T1$, $T2$, $T3$ and one inference job $I1$ arrive. In Burst Stage 2, at 1800s, an inference burst is created by launching four inference jobs $I2-I5$ and one training job $T4$. In Burst Stage 3, at 3400s, training jobs $T5$, $T6$ and inference job $I6$ arrive.

Figure 11 presents the average values of inference latency, duration of inference jobs, duration of training jobs, and training job throughput of the examined solutions, all normalized to the results of GPUColo. Similar to the motivation example, *No co-location* and *Separate* must use two GPUs because they keep inference and training jobs separate on different GPUs. Hence, they both have a shorter inference batch latency and a higher training throughput than GPUColo (at the cost of 2X CapEx). *No co-location* has a much longer duration for both inference and training because it must run those jobs sequentially without any spatial sharing. GSLICE must use two GPUs too, because it co-locates only inference jobs and runs training jobs sequentially on separate GPUs. GSLICE can control the inference latency to the SLO but has a longer duration than GPUColo due to its MPS overheads.

Simple, *Conservative*, and GPUColo all use only one GPU for a lower CapEx. Among the three, GPUColo controls the inference latency to the desired SLO with dynamic GPU resource adjustment. *Simple* has much longer inference latency that violates the SLO, because it equally divides the GPU resource in a fixed way. *Conservative* tries to give more GPU

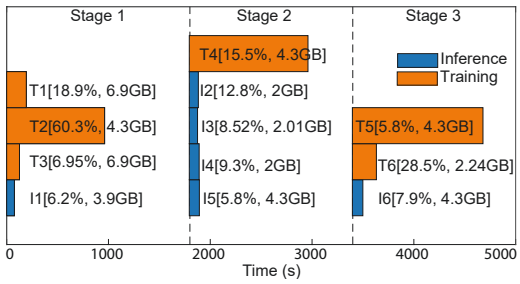


Fig. 10: Arrival time, duration (bar length), and GPU utilization (in brackets) of 12 jobs from the real-world trace.

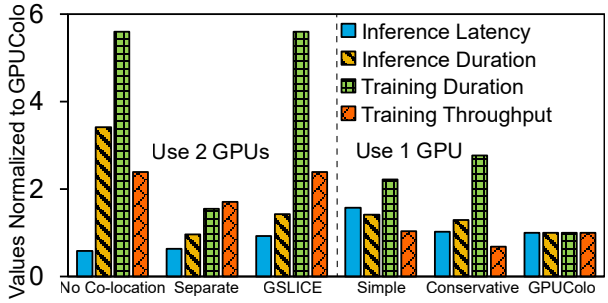


Fig. 11: No Co-location, Separate, and GSLICE must use two GPUs (2X CapEx), while Simple, Conservative, and GPUColo all use just one GPU for co-location. Simple misses the SLO, while Conservative has a lower training throughput.

resource to inference jobs, resulting in a shorter latency but a much lower training throughput. Again, fixed GPU resource allocation can neither lead to good GPU utilization nor adapt to possible inference bursts. Those results are consistent with our motivation example in Section III.

VIII. Large-scale Simulation with Trace

As introduced before, due to the risk of violating the inference latency SLO, data centers do not co-locate inference and training jobs. Now, with the SLO guarantees provided by GPUColo (as shown in Section VII-A), we perform large-scale simulation to test how many GPUs can be safely saved by GPUColo, with the 57-day real-world trace introduced in Section VI. We compare against *No Co-location*, *Separate Co-location*, and GSLICE here because they are the state-of-the-art solutions. *Simple* and *Conservative* are not practical because they rely on fixed GPU resource allocation. Hence,

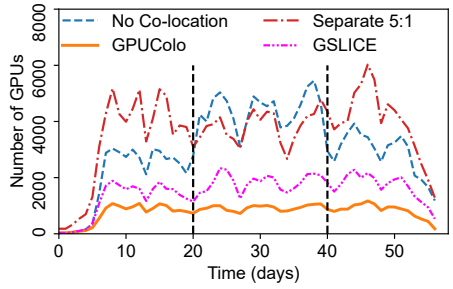
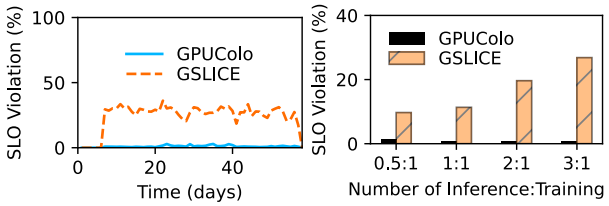


Fig. 12: Large-scale simulation with a 57-day real-world trace shows that GPUColo can save 74.9% of GPUs.



(a) Entire trace. (b) Different job ratios.

Fig. 13: GPUColo has much lower SLO violation rates because it co-locates inference and training jobs.

Simple can violate the SLO, while *Conservative* can have much lower training throughput, as shown in Section VII-B. For *Separate*, we follow the setup of a real data center [19] to have the inference GPUs as 5 times the training GPUs (*i.e.*, 5:1), as discussed in Section I. The initial job ratio of *inference:training* of the trace is 3:1. To simulate an inference request burst, we double the number of inference jobs by duplicating them between days 20 and 40.

Figure 12 presents the daily average number of GPUs used by each solution. GPUColo uses an average of 766 GPUs (over 57 days) to safely co-locate all the jobs in the trace with SLO guarantees. In contrast, *No Co-location* only allows one job to run on any GPU without spatial sharing, resulting in an average of 3059 GPUs. *Separate 5:1* must keep the ratio between inference and training GPUs as 5:1 and it cannot use inference GPUs to run training jobs or vice versa. As a result, it can demand for a lot of unused inference GPUs when the ratio of actually used GPUs is smaller (e.g., 3:1). Hence, it requires even more GPUs than *No Co-location* with an average of 3625 GPUs. GSLICE only co-locates inference jobs, resulting in an average of 1538 GPUs. During the inference job burst between days 20 and 40, *No Co-location* has to use many more GPUs to process the increased inference jobs. GSLICE has smaller GPU increase because it co-locates those inference jobs. GPUColo is not impacted much because *GPUColo is the only solution that co-locates inference and training jobs on the same GPUs*. Overall, GPUColo saves 74.9% and 50.2% of GPUs on average (*i.e.*, 2293 and 772 GPUs), if we compare it with the state-of-the-practice/art solutions *No Co-location* and GSLICE, respectively. If we assume the Nvidia A100 GPU whose price is approximately $10,000, GPUColo can safely reduce $22.9M CapEx for this data center with 6500 GPUs.

To test the SLO guarantees of GPUColo and GSLICE, we also measure the SLO violation by setting the SLO of each task as 0.96 of its length in the trace. Figure 13a shows that GPUColo successfully keeps the SLO violation rate low at 1.1% on average, but GSLICE has 27.2%. During the inference job burst, the violation of GSLICE becomes 28.4% on average. The key reason is that GSLICE co-locates only inference jobs on *GPUs separated* from training jobs. As a result, when all the inference jobs need to speed up, they compete for GPU resources with each other, resulting in SLO violations. In sharp contrast, GPUColo co-locates inference and training jobs together. Hence, if inference jobs need more GPU resources, the training jobs co-located on the same GPUs can be slowed down. We then test different

inference:training ratios. Figure 13b shows that the SLO violation rate of GSLICE does decrease when there are fewer inference jobs, due to decreasing GPU resource competition among inference jobs. This result clearly demonstrates that safely co-locating inference and training jobs *with latency control* can significantly increase GPU resource utilization and reduce CapEx.

IX. Conclusion

The concern of inference latency SLO violation has prevented data centers from co-locating ML inference and training jobs on the same GPUs. In this paper, we have presented GPUColo, a two-tier co-location solution that provides explicit SLO guarantees. In the outer tier, we exploit GPU spatial sharing (e.g., MPS) to dynamically adjust the percentage of active GPU threads allocated to spatially co-located inference and training processes, such that the inference latency can be guaranteed. Because MPS can introduce considerable overheads and so cannot be conducted at a fine time granularity, we design an inner tier that puts training jobs into periodic sleep, so that the inference jobs can quickly get more GPU resources for more prompt latency control. Our hardware testbed results show that GPUColo can precisely control the inference latency to the desired SLO, while maximizing the throughput of the training jobs co-located on the same GPUs. Our large-scale simulation with a 57-day real-world data center trace also demonstrates that GPUColo enables latency-guaranteed inference and training co-location. Consequently, it allows 74.9% of GPUs to be saved for a much lower capital expense.

References

[1] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. H. Liu, and C. Guo, "Tiresias: A gpu cluster manager for distributed deep learning." in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[2] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[3] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient gpu cluster scheduling," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[4] R. Liaw, R. Bhardwaj, L. Dunlap, Y. Zou, J. E. Gonzalez, I. Stoica, and A. Tumanov, "Hypersched: Dynamic resource reallocation for model development on a deadline," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2019.

[5] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *SOSP*, 2019.

[6] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system." in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[7] A. Dhakal, S. G. Kulkarni, and K. Ramakrishnan, "Gslice: controlled spatial sharing of gpus for a scalable inference platform," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*, 2020.

[8] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A gpu cluster engine for accelerating dnn-based video analysis," in *SOSP*, 2019.

[9] P. Jain, X. Mo, A. Jain, H. Subbaraj, R. S. Durrani, A. Tumanov *et al.*, "Dynamic space-time scheduling for gpu inference," in *NeurIPS*, 2018.

[10] W. Kang, K. Lee, J. Lee, I. Shin, and H. S. Chwa, "Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks," in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2021.

[11] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency," in *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*, 2017.

[12] OpenAI, "Gpt-4 technical report," 2023.

[13] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "Deepcpu: Serving rnn-based deep learning models 10x faster," in *USENIX ATC*, 2018.

[14] E. Nurvitadhi *et al.*, "Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic," in *FPL*, 2016.

[15] S. Rajbhandari, Y. He, O. Ruwase *et al.*, "Optimizing cnns on multicores for scalability, performance and goodput," in *ASPLOS*, 2017.

[16] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram *et al.*, "Eie: Efficient inference engine on compressed deep neural network," in *ISCA*, 2016.

[17] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[18] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "Slaq: Quality-driven scheduling for distributed machine learning," in *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, 2017.

[19] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "Pipeswitch: Fast pipelined context switching for deep learning applications," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2020.

[20] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters," in *NSDI*, 2022.

[21] Nvidia multi-process service. [Online]. Available: https://docs.nvidia.com

[22] N. Otterness and J. H. Anderson, "Exploring amd gpu scheduling details by experimenting with "worst practices"," in *RTNS*, 2021.

[23] The NVIDIA TensorRT website. [Online]. Available: https://developer.nvidia.com/tensorrtg

[24] The Tensorflow Serving website. [Online]. Available: https://www.tensorflow.org/tfx/guide/serving

[25] The Amazon SageMaker website. [Online]. Available: https://aws.amazon.com/pm/sagemaker

[26] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Multi-model machine learning inference serving with gpu spatial partitioning," *arXiv preprint arXiv:2109.01611*, 2021.

[27] Y. Xiang and H. Kim, "Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019.

[28] G. Chen and X. Wang, "Performance optimization of machine learning inference under latency and server power constraints," in *the 42nd IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2022.

[29] J. Li, H. Xu, Y. Zhu, Z. Liu, C. Guo, and C. Wang, "Lyra: Elastic scheduling for deep learning clusters," in *EuroSys*, 2023.

[30] Google Cloud. Speech-to-text. [Online]. Available: https://cloud.google.com/speech-to-text/docs/async-time-offsets

[31] AWS Machine Learning Blog. How to decide between amazon rekognition image and video api for video moderation.

[32] X. Wang, X. Fu, X. Liu, and Z. Gu, "Power-aware cpu utilization control for distributed real-time systems," in *RTAS*, 2009.

[33] G. F. Franklin *et al.*, *Digital control of dynamic systems*. A-W, 1998.

[34] The Training Tutorial website. [Online]. Available: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

[35] W. Gao, P. Sun, Y. Wen, and T. Zhang, "Titan: a scheduler for foundation model fine-tuning workloads," in *Proceedings of the 13th Symposium on Cloud Computing (SoCC)*, 2022.

[36] K. Zhang, P. Wang, N. Gu, and T. D. Nguyen, "Greendrl: managing green datacenters using deep reinforcement learning," in *Proceedings of the 13th Symposium on Cloud Computing (SoCC)*, 2022.

[37] E. Meskar and B. Liang, "Fair multi-resource allocation in heterogeneous servers with an external resource type," *IEEE/ACM Transactions on Networking*, 2022.

[38] J. You, J.-W. Chung, and M. Chowdhury, "Zeus: Understanding and optimizing gpu energy consumption of dnn training," in *NSDI*, 2023.

[39] Z. Yang, Z. Ye, T. Fu, J. Luo, X. Wei, Y. Luo, X. Wang, Z. Wang, and T. Zhang, "Tear up the bubble boom: Lessons learned from a deep learning research and development cluster," in *ICCD*, 2022.

[40] B. Cai, Q. Guo, and X. Dong, "Autoinfer: A self-driving management for resource efficient, slo-aware machine learning inference in gpu clusters," *IEEE Internet of Things Journal*, 2022.