InSS: An Intelligent Scheduling Orchestrator for Multi-GPU Inference with Spatio-Temporal Sharing

ABSTRACT

As the applications of AI proliferate, it is critical to increase the throughput of online DNN inference services. Multi-process service (MPS) improves the utilization rate of GPU resources by spatial-sharing, but it also brings new challenges. First, there is interference between co-located DNN models deployed on the same GPU, affecting both the inference latency and throughput. Second, inference tasks arrive dynamically online, and each task needs to be served within a bounded time to meet the service-level objective (SLO). Scheduling decisions (including batch size, model placement, resource allocation) must be adjusted in a very short time to meet SLO and throughput requirements. To address the above two challenges, we propose an Intelligent Scheduling orchestrator for multi-GPU inference servers with spatio-temporal Sharing (*InSS*), aiming to maximize the system throughput. *InSS* exploits two key innovations: i) An interference-aware latency analytical model which estimates the GPU execution latency and queuing latency. ii) A two-layer intelligent scheduler based on Soft Actor-Critic Discrete and binary search which jointly optimizes the model placement, GPU resource allocation and adaptively decide batch size by coupling the latency analytical model model. Our prototype implementation on eight DNN models and four GPUs show that *InSS* can improve the throughput by up to 189% compared to the state-of-the-art GPU schedulers, while satisfying SLOs.

1 INTRODUCTION

As more and more AI applications enter people's daily lives, the demand of GPU resources for inference services is also increasing rapidly. A recent survey shows that the resource usage of inference services in China's AI-related industries has accounted for more than 55% of the total GPU usage, and the proportion will continue to increase in the future [50]. However, the practical problem faced by many internet companies like Google, Alibaba which run multiple DNN models to support their low-latency products (*e.g.,* voice assistant [3], recommendation system [22]) is that the GPU utilization rate of online inference services is generally low. The high throughput cannot be achieved while ensuring the service-level objective (SLO) [2, 47]. Therefore, it is of great significance to improve the efficiency of GPU usage.

Temporal sharing [38] and spatial sharing [48] are the two most common ways of sharing resources. Temporal sharing enables each DNN model to occupy all GPU resources at one time, and realizes time sharing through scheduling. Spatial sharing slices a GPU into multiple pieces, and each piece runs a DNN model. In addition, by combining multiple interference tasks which request the same DNN model through dynamic batch, the utilization of GPU resources can be further improved [11]. NVIDIA Multi-Process Service (MPS) is a logical resource partition mechanism that enables multiple DNN models to be executed concurrently on the same GPU [34]. MPS assigns a limited percentage of GPU resource (*e.g.,* 30%) to each DNN model, potentially producing more thorough resource usage and better system throughput.

Although MPS brings opportunities to improve GPU resource utilization, it also brings new challenges. *First*, there is interference between co-located DNN models running on the same GPU [1]. This is because MPS only isolates streaming multiprocessors, while other resources like L2 Cache and PCIe bandwidth are shared by co-located DNN models [45]. Such interference affects the GPU execution latency[1], leading an increased task latency[2] and a lower system throughput. The experiments in Sec. 2.2 indicate that the interference can increase the latency by up to 18.7%. *Second*, inference tasks have SLOs and must be completed within the required time. In practical, inference tasks arrive dynamically online, MPS-based spatio-temporal sharing decisions and batch sizes must be adjusted in a very short time to meet the SLO and throughput requirements.

To improve the utilization of GPU resources, some works apply batch processing [1, 6, 10, 11, 16, 21, 31], and some studies enables spatial sharing [7, 12, 23, 25, 27] or temporal sharing [17, 29, 38, 43, 46] of GPU resources. But most of them do not consider the interference, *e.g., GSLICE* [14], *Clipper* [10]. Some works [5, 20, 40] takes into account interference, but need to store all the latency data under different settings, which has a large overhead and is not suitable for online tasks. Only *iGniter* [45] came up with an interference model, but it aims to minimize the cost. In addition, *iGniter* ignores the queuing latency in batching and its resource allocation heavily relies on the accuracy of latency prediction, leading to poor performance with the dynamic online workloads. We will discuss detailedly in Sec. 7.

[1]GPU execution latency, *a.k.a.*, inference latency, describes how long a batch of the model spends running on the GPU.

[2]Task latency refers to the duration between the arrival of a task and the return of its inference result.

To deal with the above two challenges, we propose and implement an Intelligent Scheduling orchestrator for multi-GPU inference servers with spatio-temporal Sharing (*InSS*) based on an interference-aware latency analytical model. To the best of our knowledge, *InSS* is the *first work* that characterizes the task latency while considering interference and batch queuing latency, and can make adaptive spatio-temporal and batch processing adjustments for online dynamic inference tasks in a very short time. *InSS* guarantees the SLO requirement while maximizing the system throughput. We make the following contributions.

- *InSS* builds an analytical performance model to predict the task latency. We first analyze the GPU execution latency by considering the inference among co-located DNN models. We characterize the impact of PCIe bandwidth, cores and L2 Cache, and propose a light-weighted prediction model. We then model the queuing latency in batch processing based on the batch size and the workload arrival rate. The estimated GPU execution latency and the queuing latency are then used to calculate the task latency.
- *InSS* employs a two-layer intelligent scheduler, *TS*, to jointly optimize GPU provisioning (spatio-temporal sharing) and batch size decision. TS leverages a Soft Actor-Critic-Discrete (SAC-D) based approach to decide the model placements and resource allocation. *TS* then inputs the predicted task latency to the binary search-based algorithm *BDA* which can find the optimal batch size to maximize the system throughput while satisfying constraints.
- We implement a prototype of *InSS* with eight DNN models and four GPUs. We observe: i) *InSS* provides an accurate light-weighted prediction model for the inference task, with an error rate of less than 4% in the worst case. ii) *InSS* can improve the throughput by 28.1%, 47.7%, and 189.7%, compared to *Gpulet* [5], *GSLICE* [14] and *iGniter* [45], respectively. iii) *InSS* always maintains excellent service quality, with its SLO violations consistently below 1%, while the SLO violations of the three baselines reach 4.4%, 4.5%, and 3.8%, respectively. Moreover, the prototype of *InSS* is open-sourced on GitHub.

2 BACKGROUND AND MOTIVATION

2.1 Improve GPU Utilization

2.1.1 Batch-aware Execution. Batching is a common way to increase throughput and GPU utilization [1, 31, 37]. It waits for tasks to arrive and accumulate to the required amount to start the batch execution. The reasons why it can improve GPU utilization are: i) The unique characteristics of GPU resources enable the creation of extra computing threads, allowing for the parallel processing of multiple requests. ii) It reduces the number of data transfers between the CPU and GPU, thereby reducing the PCIe bandwidth utilization time. To show the performance of batch execution, we experimented with 4 different DNN models (AlexNet, ResNet50, VGG16, and MnasNet). The specific configuration of experiments is presented in Sec. 6.

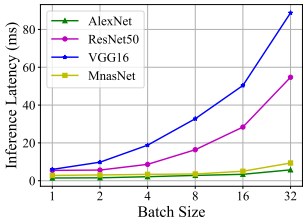

Figure 1: Latency under different batch sizes.

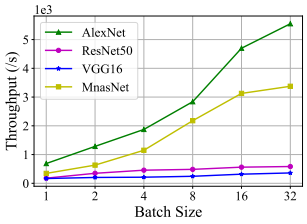

Figure 2: Throughput under different batch sizes.

As shown in the Fig. 1 and Fig. 2, it is the inference latency and system throughput of a batch with different DNN models under different batch size settings. It can be seen that as the batch size increases, both throughput and latency increase non-linearly. However, inference tasks have SLO requirements, the system cannot wait indefinitely to collect a heavy batch. Therefore, it is challenging to choose a suitable batch size for executing inference tasks with each model. In this study, we develop an adaptive batching method to leverage batching's benefits while limiting the latency to stay within the SLO in Sec. 5.3.

2.1.2 Spatial Sharing Execution. Given the latency limit of inference tasks, the batch size cannot be arbitrarily adjusted, making it difficult to significantly enhance GPU utilization only through batch processing. GPU suppliers such as NVIDIA have developed several mechanisms to realize resource spatial sharing: i) Multi-stream processing [32], a software-based programming model, enables different streams to run simultaneously and share the underlying GPU resources like Streaming Multiprocessors (SMs). But the resource allocation is uncontrollable, and there will be pseudo-parallel [14]. ii) Multi-Process Service (MPS) [34], a logical resource partition mechanism, can allocate SMs to different processes in percentages. For example, 40%, 60% to two parallel processes. However, in this way, the remaining GPU resources are still shared (such as L2 Cache and memory bandwidth). Therefore, there is interference when multiple models run in parallel. iii) Multi-Instance GPU (MIG) [33], a physical resource partition method, realizes the division of physical resources at the hardware layer and ensures the complete isolation of all resources. There is no interference when multiple processes run in parallel. But still, complete resource isolation results in a high temporal overhead for resource reallocation. In summary, we choose MPS technology to achieve relatively flexible and controllable resource spatial sharing.

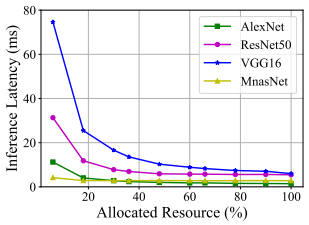

Figure 3: Inference latency under different amounts of allocated resource.

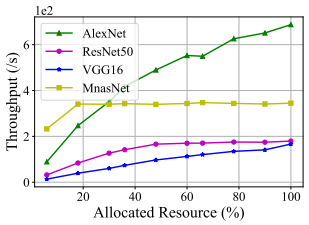

Figure 4: Throughput under different amounts of allocated resource.

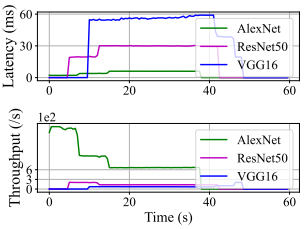

Figure 5: Inference latency and throughput with GPU default mode.

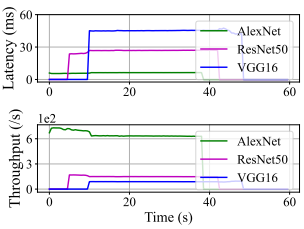

Figure 6: Inference latency and throughput with MPS mode.

We plot the inference latency and throughput for processing a task (batch size = 1) with a variety of DNN models under different GPU resources (%) in Fig. 3 and Fig. 4. As can be seen, throughput and latency are significantly impacted by the amount of resources allocated, and their relationship is also non-linear. Furthermore, as the resources provided to a model reach a certain value (*e.g.*, 60% GPU for ResNet50), the advantage of increasing resources for throughput and latency become significantly less pronounced. It is obvious that in the majority of circumstances, allocating a whole GPU resource to a model task is wasteful. Additionally, it can be discovered that employing a portion of a GPU's resources can meet the task's SLO and provide sufficient throughput. Concurrently, the residual resources of the GPU can be leveraged for additional computations, allowing for the concurrent execution of multiple DNN models under a given resource constraint on a single GPU. GPU resource spatial sharing through the MPS mechanism is necessary and can significantly raise GPU utilization. Consequently, it is vital and challenging to allocate the proper amount of GPU resources to each DNN model.

2.2 Interference among Co-located DNN Models

When the isolation of SM resources is achieved in MPS, other resources of the GPU are shared. So there is interference between co-located models on the same GPU. We experimented three DNN models under different GPU working modes: i) GPU default mode, multiple processes can use the GPU simultaneously. ii) MPS mode, limit each model to just using a fixed amount of GPU resources. Specifically, The resource allocation for AlexNet, ResNet50, and VGG16 is set to 30%, 30%, and 40% GPU, respectively. We begin with AlexNet and then launch ResNet50 after 5s and VGG16 after 10s. The results are plotted in Fig. 5 and Fig. 6. Under the specific resource configuration in MPS, the latency of the DNN model calculation is more stable and controllable. In addition, it can also be found that when more models are added to parallel computing, the execution latency increases, which proves that interference does exist. Furthermore, the MPS model's interference is less than that of the GPU's default mode.

To further examine the impact of interference, Fig. 7 presents the results of ResNet50 in additional experiments with different system configurations. Here, we use AlexNet (A) and VGG16 (V) as co-located models, and the batch size for both is fixed to 4. As for resource allocation, ResNet50 is fixed to be allocated with 40% GPU, while X' indicates that model X is allocated with 30% GPU and X* indicates allocated with 60% GPU. The increased latency refers to the percentage increase in latency for ResNet50 in the current deployment compared to its performance when run alone. It can be observed that changes in batch size, allocated resources, and number of co-located models have a significant impact on interference values, with a latency increase of up to 18.7%. Therefore, the interference cannot be neglected. Similar to [5, 45], we consider the impact of three factors: PCIe bandwidth, the number of cores, and L2 Cache for interference. In Sec. 4, we develop a well-designed model for interference-aware latency prediction.

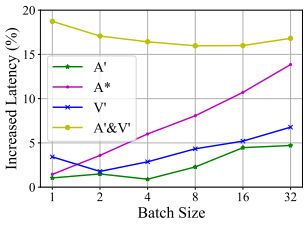

Figure 7: Increased latency with different co-located models and batch sizes.

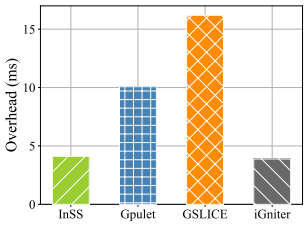

Figure 8: Scheduling overhead with different algorithms.

2.3 Scheduling Overhead

Considering the above factors, the most similar works are *Gpulet* [5], *GSLICE* [14], and *iGniter* [45]. Due to the low-latency requirements of inference tasks, the scheduling algorithm for inference services needs to make decisions in a short time to minimize the impact of scheduling on inference tasks. We evaluate the scheduling overhead of *InSS* and the three algorithms. As shown in Fig. 10, *Gpulet* and *GSLICE* have relatively high scheduling overhead, which is larger than the inference latency of many tasks and cannot meet the SLOs. Although the scheduling overhead of *iGniter* is similar to *InSS*, it has two main drawbacks: i) It is not aiming to maximize the throughput. ii) It requires extremely high

precision in latency prediction, which is demonstrated in Sec. 6. Therefore, differing from these algorithms, we introduce *InSS*, a fault-tolerant and efficient scheduling orchestrator, that achieves predictable task latency and maximizes system throughput while meeting task SLOs.

3 *INSS*: ARCHITECTURE & DESIGN

From Sec. 2, we can efficiently raise GPU utilization by leveraging the MPS mechanism and batching. Inspired by this observation, we introduce a scheduling orchestrator, *InSS*, for online DNN inference serving, which determines the optimal configuration for a GPU cluster (*i.e.,* model deployment, resource allocation, and batch size decision) to maximize cluster throughput while guaranteeing the SLO requirement of tasks.

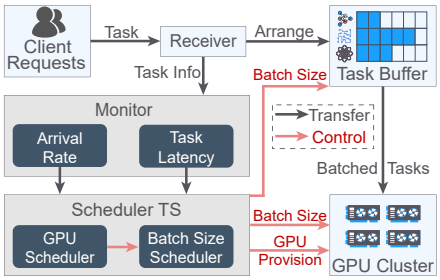

Figure 9: Overall architecture of *InSS*.

3.1 Architecture and Workflow

Fig. 9 illustrates the overall architecture of *InSS*. It mainly consists of four components: a task buffer, a monitor, a scheduler, and a GPU cluster. The following stages describe how inference tasks are handled in more detail:

i) *Task Buffer*. Clients continuously generate tasks and submit them to the system for processing. After accepting the tasks, the system places them in the *task buffer*. Since tasks are performed in batches, arriving tasks are queued in the *task buffer* until the previous batch is finished and the current batch is assembled.

ii) *Monitor*. The *monitor* gathers empirical data and makes some predictions. The monitor first monitors the task arrival process to predict the workload arrival rate. Then, a latency prediction model is designed to capture the relationship between system configuration, workload, and inference latency. Latency prediction includes two time metrics: **GPU execution latency** and **task latency**[3].

iii) *Scheduler*. Using workload arriving time, task latency prediction model, and the information of tasks as input, the *scheduler* combines reinforcement learning and traditional optimization methods to determine the optimal configuration for the system (model deployment, resource allocation, and batch size decision) to maximize the cluster throughput while ensuring that the task SLO requirements are not violated. The optimal batch size is communicated to the task buffer, while the model placement and resource allocation are sent to the *GPU cluster*.

iv) *GPU cluster*. According to the *scheduler*'s parameters, the *task buffer* groups tasks into batches and sends the data to the *GPU cluster* for processing after a batch has been assembled.. The *GPU cluster* deploys models and reallocates GPU resources according to the configuration determined by the *scheduler*, and performs inference computation after receiving data.

[3]The definition is introduced in Sec. 1

3.2 System Model

We model the system to clearly describe the orchestrator.

System Overview. We consider a system for supporting DNN inference services on a GPU cluster equipped with k GPUs. Each GPU can simultaneously execute the workload of multiple DNN models by utilizing the MPS mechanism to spatial share the computing resources. Denote the computing power of the GPU k as C_k. Let M_k represents the maximum memory limit of GPU k. Let $\mathcal{X}$ denote the integer set $\{1, 2, \ldots, X\}$.

The system provides inference services for I types of models (*e.g.,* AlexNet, ResNet50, and VGG16). Each DNN model $i \in \mathcal{I}$ is well-trained and pre-stored in the system, and can be deployed to all GPUs when needed. The memory usage of a DNN model during inference on a GPU is affected by changes in the batch size due to the allocation of memory for intermediate computations and the storage of data. Specifically, a larger batch size may require more memory to store intermediate computation results and a larger amount of data. Therefore, denote $m_i(b_i)$ as the the memory footprint of the DNN model i with a batch size of b_i. Let p_i^d and p_i^f represent the input data size and the output data size of the DNN model i when the batch is 1, respectively. Denote λ_i as the arriving rate of DNN model i's workload, which is predicted by the monitor based on historical task arrival statistics. Each model workload has its own SLO, which is a measure of the maximum allowable inference latency it takes for a model to generate an output in response to an input. Let SLO_i indicates the SLO requirement of DNN model i. Over a time span $\mathcal{T}$ (*e.g.,* several minutes), a set of inference tasks using model i arrive randomly online and request for processing, denoted as $\mathcal{J}_i$. Each inference task $j \in \mathcal{J}_i$ with DNN model i can be represented by a tuple $\{a_{i,j}, d_{i,j}, T_{i,j}\}$, where $a_{i,j} \in \mathcal{T}$ indicates the arrival time of task j, $d_{i,j}$ denotes the data size of task j, and $T_{i,j}$ denotes the task latency of task j.

Decision Variables. i) $y_{i,k} \in \{0, 1\}$, a binary variable which represents whether DNN model i is deployed at GPU k; ii) $\rho_{i,k}$, the ratio of computing resources allocated to DNN model i at GPU k; iii) $b_i \in \mathcal{N}^+$, the batch size of DNN model i.

3.3 Problem Formulation

Our objective is to maximize system throughput while ensuring the SLO requirement of tasks. Let $h_{i,k}$ denote the throughput of DNN model i on GPU k. The online problem for DNN inference services on a GPU cluster can be formulated as follows.

$$\text{maximize} \sum_{\forall i \in \mathcal{I}, \forall k \in \mathcal{K}} y_{i,k} h_{i,k} \tag{1}$$

subject to:

$$\sum_{i \in \mathcal{I}} y_{i,k} \rho_{i,k} \leq 100\%, \forall k \in \mathcal{K}, \tag{1a}$$

$$\sum_{i \in \mathcal{I}} y_{i,k} m_i(b_i) \leq M_k, \forall k \in \mathcal{K}, \tag{1b}$$

$$T_{i,j} \leq SLO_i, \forall i \in \mathcal{I}, \forall j \in \mathcal{J}_i, \tag{1c}$$

$$\sum_{\forall k \in \mathcal{K}} y_{i,k} h_{i,k} \geq \lambda_i, \forall i \in \mathcal{I}, \tag{1d}$$

$$\sum_{\forall k \in \mathcal{K}} y_{i,k} = 1, \forall i \in \mathcal{I}, \tag{1e}$$

$$y_{i,k} \in \{0,1\}, b_i \in \mathcal{N}^+, \forall i \in \mathcal{I}, \forall k \in \mathcal{K}, \tag{1f}$$

where constraint (1a) guarantees that the computing resource allocated to all DNN models at each GPU does not exceed the resource capacity. The memory capacity of each GPU for model placement is formulated by constraint (1b). Constraint (1c) ensures that the task latency of each task does not exceed its SLO. Constraint (1d) makes sure that the throughput of each DNN model after deployment can meet the arriving rate of its workload. Constraint (1e) means that each model is only deployed on one GPU.

Challenges. i) Given the values of $\rho_{i,k}$ and b_i, problem (1) can be reduced to a 0-1 knapsack problem, which is known to be NP-hard [15, 18]. Hence, even in the offline setting, problem (1) is NP-hard, making it challenging to solve using conventional optimization methods. ii) We consider a realistic scenario where the arrival process of inference tasks is unknown. iii) $T_{i,j}$ is an unknown parameter that is nonlinearly correlated with multiple decision variables.

4 PREDICTION OF TASK LATENCY

Our task latency prediction model is developed with the consideration of heterogeneity in DNN models and GPU devices, as well as the potential interference caused by co-located models. The prediction model consists of two major components:

i. First, we build an analytical performance model to estimate the GPU execution latency. It explicitly captures the influence of heterogeneous system configuration and the performance interference among co-located DNN inference workloads.
ii. We model the queuing delay due to waiting for batch processing. The estimated GPU execution latency and queuing delay are then used to calculate the task latency to check whether its SLO requirement is met.

4.1 GPU Execution Latency Prediction

GPU Execution Latency of a DNN Model. The execution of the DNN model workload on GPU can be divided into three sequential steps: data uploading, GPU execution, and result feedback. Therefore, the GPU execution latency of a batch with DNN model i on GPU k can be calculated as: $t_{i,k}^{inf} = t_{i,k}^{d} + t_{i,k}^{c} + t_{i,k}^{f}$, where $t_{i,k}^{u}$ is the data upload latency from CPU to GPU, $t_{i,k}^{c}$ is the GPU computing latency, and $t_{i,k}^{f}$ is the result feedback latency from GPU to CPU. Then, the throughput of model i on GPU k can be formulated as: $h_{i,k} = b_i / t_{i,k}^{inf}$.

Data Interaction Latency. Inference input data needs to be uploaded from the CPU to the GPU for computing. The calculated results need to be transferred from the GPU to the CPU to be returned to the task generator. The data interaction between the CPU and GPU device is typically carried out through the PCIe bus. Therefore, we can use the data volume and the available PCIe bandwidth to estimate the time it will take to transfer data between two devices. Let B_k indicates the available PCIe bandwidth of GPU k. The data upload latency and the result feedback latency of a batch with DNN model i at GPU k can be obtained by $t_{i,k}^{d} = \frac{p_i^d * b_i}{B_k}$, and $t_{i,k}^{f} = \frac{p_i^f * b_i}{B_k}$.

GPU Computing Latency. According to the GPU execution workflow, the GPU scheduler must schedule each core of a DNN model workload to compute unit SMs before performing computations. The GPU computing phase consists of core scheduling and core running. Therefore, the computing latency of a batch with DNN model i at GPU k can be formulated as: $t_{i,k}^{c} = t_{i,k}^{s} + t_{i,k}^{r}$, where $t_{i,k}^{s}$ and $t_{i,k}^{r}$ denote the GPU scheduling latency and the GPU running latency of a batch with DNN model i at GPU k, respectively.

GPU Scheduling Latency. Denote the number of cores for a batch with DNN model i as $n_{i,k}$. The GPU scheduling latency $t_{i,k}^{s}$ is essentially linear with the number of cores it uses $n_{i,k}$, which can be estimated as: $t_{i,k}^{s} = (o_i^{sch} + int_j) \cdot n_{i,k}$, where o_i^{sch} indicates the scheduling latency of each core for DNN model i executing alone at GPU k, and int_j denotes the increased scheduling latency caused by the interference among co-located models at GPU k. The interference on scheduling latency is connected to the number of co-located models. The scheduling time of a DNN model with different number of co-located models is plotted in Fig. 10. Combined with the data, int_j can be calculated as:

$$int_j = \begin{cases} 0, & \sum_{\forall i \in \mathcal{I}} y_{i,k} <= 1, \\ \alpha_k^s \sum_{\forall i \in \mathcal{I}} y_{i,k} + \beta_k^s, & \sum_{\forall i \in \mathcal{I}} y_{i,k} > 1. \end{cases}$$

where α_k^s and β_k^s are two linear coefficients of interference on scheduling latency.

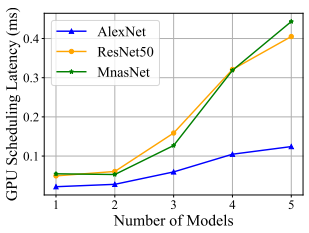

Figure 10: GPU scheduling latency under different number of models.

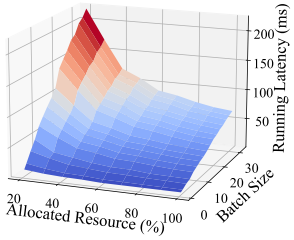

Figure 11: GPU running latency under different resources and batch sizes.

GPU Running Latency. In this phase, we primarily focus on the effects of the L2 Cache, which is shared in co-located DNN models when the MPS mechanism is employed. To illustrate the relationship between L2 Cache and interference, we use a system statistic called GPU L2 Cache utilization to represent the demand on L2 Cache for model workloads. Higher L2 Cache utilization means more competition for a fixed amount of L2 Cache resources on the GPU, which leads to longer GPU running time. GPU runtime latency may be estimated similarly to GPU scheduling latency by: $t^r_{i,k} = o^r_{i,k}(1 + \alpha^{cache}_{i,k} \sum_{\forall i \in \mathcal{I} \setminus i} l_{i,k} y_{i,k})$, where $o^r_{i,k}$ denotes the GPU running time for a batch of DNN model i executing alone at GPU k, $l_{i,k}$ indicates the GPU L2 Cache utilization of DNN model i executing alone at GPU k, and $\alpha^{cache}_{i,k}$ is the coefficient of interference on model i's GPU running time.

Obviously, the GPU execution time of models is proportional to the batch size and the amount of resources allocated. Fig. 11 depicts the DNN model's GPU running time under different configurations. It has been observed that the GPU running latency is generally inversely related to the amount of allocated resources and rapidly grows with batch size. However, the relationship between these variables is complex and nonlinear, and it is not feasible to derive an analytical model for it. As a result, we utilized Locally Weighted Regression (LWR) [9] to effectively capture the relationship between the allocated resource, batch size, and GPU running time. Then, the GPU running time can be obtained by $o^r_{i,k} = LWR(b_i, \rho_{i,k} \cdot C_k)$.

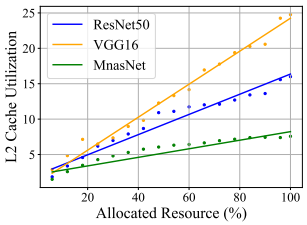

Figure 12: L2 Cache utilization.

In general, higher GPU computing capability results in more significant L2 Cache competition (*i.e.,* larger L2 Cache utilization). We plot the L2 Cache utilization under different allocated GPU resources in Fig. 12. The dots in the figure indicate the measured values of L2 Cache utilization. According to the data, the L2 Cache utilization $l_{i,k}$ can be predicted as: $l_{i,k} = \alpha^u_{i,k}\rho_{i,k} + \beta^u_{i,k}$, where $\alpha^u_{i,k}$ and $\beta^u_{i,k}$ are two coefficients for model i at GPU k. We also plot the curves fitted by the formula in Fig. 12, and it is evident that the fit is excellent.

Prediction Overhead. In the above prediction model, parameters and coefficients need to be measured before the system operates. For each GPU, the available PCIe bandwidth is obtained by transferring inference data from CPU to GPU. The parameters $\{o^{sch}_i, n_{i,k}\}$ are invariant for a given GPU device and a specific DNN model. They can be obtained by utilizing Nsight Systems [36] and Nsight Compute [35] to analyze the DNN model run data just once. Moreover, the model coefficients $\{\alpha^s_j, \beta^s_j, \alpha^{cache}_{i,k}\}$ require data collected by launching multiple DNN models concurrently. We collect five sets of data for these coefficients in the experiment. As for coefficients $\{\alpha^u_{i,k}, \beta^u_{i,k}\}$, the fitting data is collected when the DNN model runs alone on the GPU. We measure the running data under 16 configurations (4 options of resource allocation and 4 options of batch size) and fit all coefficients $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ using the least squares method [30]. Moreover, to utilize LWR, we also measure running data under 16 configurations and made predictions for each configuration to obtain data for GPU running latency prediction. In general, the process of getting the prediction model's parameters and coefficients takes a few minutes.

4.2 Task Latency Prediction

Arriving Rate. Clients submit continuous inference jobs to the receiver online, who then adds them to the task buffer. Our system processes requests sequentially in a FIFO (first in first out) manner. Prior work observes that the arrival rate of tasks for DNN inference service follows a Poisson distribution [51]. In this paper, we assume that the data size of each task is a batch of size 1. In practical applications, tasks with a large amount of data can be broken down into numerous smaller tasks under this setting. In this way, the arrival rate of each model workload λ_i can be obtained by fitting a Poisson distribution. The monitor predicts the arriving rate of model workload in the future based on the historical arrival data.

Task Latency. The latency of a task starts from its submission to the completion of the result delivery. Since the data size of the result is relatively small, the transmission latency of the result delivery is ignored [28]. Then, the task's queuing latency in the task buffer must be taken into account. Let $G_i(t)$ denote the queued workload of model i at t (before the task arrives). The queued workload of DNN model i after task j is $W_{i,j} = G_i(a_{i,j}) + d_{i,j}$. There are two situations in which tasks are queued in the task buffer: i) The queued workload is sufficient (*i.e.,* $W_{i,j} \geq b_i$), the current task must wait for the

processing of the tasks that arrived earlier. ii) The queued workload is less than that required for batch assembly (*i.e.,* $W_{i,j} < b_i$), The current task must wait for the arrival of the subsequent tasks until the queued workload equals the size of a batch. The task latency of task j with DNN model i is:

$$T_{i,j} = \begin{cases} \sum_{\forall k \in \mathcal{K}} y_{i,k} exec_k + \lceil W_{i,j}/b_i \rceil \sum_{\forall k \in \mathcal{K}} y_{i,k} t_{i,k}^{inf}, & W_{i,j} \geq b_i, \\ max(\sum_{\forall k \in \mathcal{K}} y_{i,k} exec_k, wait_{i,j}) + \sum_{\forall k \in \mathcal{K}} y_{i,k} t_{i,k}^{inf}, & W_{i,j} < b_i, \end{cases} \tag{2}$$

where $exec_k$ denotes the remaining DNN model execution time of the workload *currently* being executed at the GPU k. $wait_{i,j}$ indicates the queuing latency for sufficient workloads to be completed for assembling the batch, which is calculated by: $wait_{i,j} = \frac{b_i - W_{i,j}}{\lambda_i}$.

5 INTELLIGENT SCHEDULING

5.1 Main Idea

Considering the temporal overhead of GPU resource reallocation, the batch size is first adjusted to accommodate changes in workload. When there is a significant change in models' workload, resource reallocation is carried out and then calculate batch size. Therefore, we first decompose problem (1) into two sub-problems: GPU provisioning (spatio-temporal sharing) and batch size decision. Then, a two-layer intelligent scheduler, *TS*, is proposed to address them jointly. *TS* comprises the following steps:

i. Initially, *TS* transforms the GPU provisioning subproblem into an Markov Decision Process (MDP), and leverages a SAC-D-based approach to solve it. In our scenario, the *scheduler* is regarded as an agent, which can dynamically obtain the state information s_t from the *monitor* and *GPU cluster*. Given the state s_t, the agent then makes the discrete action to decide the model placements $\{y_{i,k}\}_{i \in \mathcal{I}, k \in \mathcal{K}}$ (temporal sharing) and resource allocation $\{\rho_{i,k}\}_{i \in \mathcal{I}, k \in \mathcal{K}}$ (spatial sharing) based on its own policy π.

ii. After the GPU provisioning is determined, *TS* devises the binary search-based algorithm *BDA* to address the subproblem of batch size decision. *BDA* enables efficient search of the state space, and find the optimal batch size $\{b_i\}_{i \in \mathcal{I}}$ that maximizes throughput while satisfying constraints.

iii. After the *task buffer* and *GPU cluster* receive the decision and perform the DNN inference services, the environment updates the state to s_{t+1} and feedbacks the reward r_t to the agent. The tuple $\{s_t, a_t, r_t, s_{t+1}\}$ is then stored in the replay buffer for sampling by the agent to update its actor and critic networks. The agent continues to make decisions with state s_{t+1} until the DRL network training is completed. The trained network can be used for online instant GPU provisioning decisions.

5.2 Solution for GPU provisioning

Problem Transformation. To address the GPU provisioning subproblem, we first reformulates it as an MDP. An MDP is characterized by a tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma\}$, where $\mathcal{S}$ denotes the set of environment states, $\mathcal{A}$ indicates the action space, $\mathcal{P}(s'|s,a)$ represents the probability of transitioning between states, $\mathcal{R}$ denotes the reward function, and $\gamma \in [0,1]$ is the discount factor used for trading off future rewards to present rewards. In our scenarios, the *scheduler* is considered as an agent. The MDP of GPU provisioning subproblem can be designed as follows.

i) *State.* At each decision step, denoted as t, the agent observes state information from the current environment. In the considered system, the state can be formulated as: $s_t = \{\mathcal{M}_t, \mathcal{W}_t\}$, where $\mathcal{M}_t$ indicates the state of all GPUs and $\mathcal{W}_t$ denotes the information of all DNN models' workload. Specifically, the state of each GPU k include the used memory for DNN model placement of previous action $u_k = \sum_{\forall i \in \mathcal{I}} y_{i,k} m_i(b_i)$, *i.e.,* $\mathcal{M}_t = \{u_k\}_{k \in \mathcal{K}}$. This information is crucial for the agent to track the memory usage of GPUs and avoid memory overflow, which can lead to performance degradation or even system failure. The information of each DNN model's workload includes: the arriving rate of workload λ_i, the amount of queued workload G_i, *i.e.,* $\mathcal{W}_t = \{\lambda_i, G_i\}_{i \in \mathcal{I}}$. These variables reflect the current state of the DNN models' workload and provide the agent with insight into GPU resource demands. It is worth noting that other information in our system is fixed and therefore does not need to be included in the state.

ii) *Action.* Given the observed state s_t, the agent determines the action a_t for all DNN models at decision step t, which includes: the model placements $\{y_{i,k}\}_{i \in \mathcal{I}, k \in \mathcal{K}}$ and resource allocations $\{\rho_{i,k}\}_{i \in \mathcal{I}, k \in \mathcal{K}}$. Due to the fact that the allocated resources are in units of GPU SMs, which are discrete variables rather than continuous ones, and the decisions of model placement are also discrete, we discretized the resource allocation variables in order to obtain more accurate solutions. Denote U_k as the amount of GPU k's resource units, *i.e.,* $\rho_{i,k} * U_k \in \mathcal{U}_k$. Therefore, the action is defined as $a_t = \{y_{i,k}, \rho_{i,k}\}_{i \in \mathcal{I}, k \in \mathcal{K}}$.

iii) *Reward.* After taking the action a_t based on the observed state s_t, the agent will receive a reward r_t from the environment to evaluate the quality of the action. In accordance with the objective of problem (1) and the fundamental logic of MDP, The reward function is formulated to incentivize actions that lead to higher throughput. Furthermore, consider the constraints (1b) and (1c), a reward function is developed to incentivize the agent to choose actions that satisfy to the constraints. Let $v_{i,t}$ denote the percentage of inference tasks that violate SLO requirements at decision step t. For

ease of model comparison and selection, as well as facilitating learning and optimization, we normalize the throughput of our models. Specifically, we divide the throughput of the model by the throughput H_i when allocated 100% GPU and the batch size is 1. Denote $\hat{h}_{i,k}$ as the normalized throughput of DNN model i at GPU k, *i.e.,* $\hat{h}_{i,k} = h_{i,k}/H_i$. We formulate our reward function as follows,

$$r_t = \sum_{i \in \mathcal{I}} r_{t,i}, \text{ where } r_{i,t} = \sum_{\forall k \in \mathcal{K}} y_{i,k} \hat{h}_{i,k} - \omega v_{i,t}, \tag{3}$$

where $r_{t,i}$ indicates the reward of DNN model i, and ω is the penalty factor. Specifically, when the constraints (1b) and (1c) are satisfied, the reward is the sum of the normalized throughput achieved by DNN models. When constraint (1b) is violated, the model cannot be deployed. Therefore, all the inference tasks that arrive at the current decision step t of the model cannot be processed, which is considered as violating the SLO requirement. We use a penalty term $\omega v_{i,t}$ to reflect the violation of the SLO requirement. In this way, the actor is incentivized to select actions that result in higher throughput while ensuring the constraints of the problem are satisfied. It is worth noting that when solving this MDP, the value of $v_{i,t}$ can be either the actual value obtained from running the inference service, or the simulated value calculated by the task latency prediction model. The specific set of parameters are listed in Sec. 6

Challenge and Solution. Accurately modeling the state of the environment in GPU clusters is challenging due to the limited knowledge of transition probability. Moreover, the high-dimensional continuous state spaces and high-dimensional discrete action spaces pose a challenge in achieving tractable convergence performance [49]. Thus, conventional dynamic programming solutions are inadequate in solving the MDP problem. To address these challenges, DRL has emerged as a popular approach for solving MDP problems. In the context of solving MDP problems with high-dimensional tuple information and multiple discrete actions, the choice of DRL algorithm plays a critical role in achieving fast training speed and exploration capability. In this paper, we adopt the SAC-D algorithm, which is an off-policy actor-critic method that employs soft policy updating based on the maximum entropy RL orchestrator [8]. This algorithm has been shown to effectively address MDP problems and can be applied in large-scale networks without requiring statistics on network dynamics. Detailed information about the SAC-D algorithm is presented in Sec. 5.4.

5.3 An Adaptive Batching Method

Upon obtaining the DNN model placement and resource allocation strategy using the SAC-D-based algorithm, the original problem (1) can be reduced to a batch size decision subproblem as follows:

$$\text{maximize} \sum_{\forall i \in \mathcal{I}, \forall k \in \mathcal{K}} y_{i,k} h_{i,k} \tag{4}$$

$$\text{subject to:} \sum_{i \in \mathcal{I}} y_{i,k} m_i(b_i) <= M_k, \forall k \in \mathcal{K}, \tag{4a}$$

$$T_{i,j} <= SLO_{i,j}, \forall i \in \mathcal{I}, \forall j \in \mathcal{J}_i, \tag{4b}$$

$$\sum_{\forall k \in \mathcal{K}} y_{i,k} h_{i,k} >= r_i, \forall i \in \mathcal{I}, \tag{4c}$$

$$s_i \in \mathcal{N}^+, \forall i \in \mathcal{I}, \tag{4d}$$

where constraints (4a), (4b) and (4c) are the same as constraints (1b), (1c) and (1d).

Based on the results presented in Sec. 2, it can be observed that the task latency, model throughput, and model memory usage all increase as the batch size increases. Therefore, we assume that these parameters are sorted in ascending order relative to the batch size. By leveraging this observation, we propose *BDA* to dynamically adjust the batch size using a pseudo-binary search approach during runtime. *BDA* can effectively search the decision space in a few steps and find the most suitable batch size.

Algorithm 1 *BDA*: Batch Decision Algorithm

Input: $y_{i,k}, \rho_{i,k}, \lambda_i, T_{i,j}, \forall i \in \mathcal{I}, \forall j \in \mathcal{J}, \forall k \in \mathcal{K}$;
Output: $\{b_i\}_{i \in \mathcal{I}}$, flag;
1: Initialize $b_i = default, flag = false, \forall i \in \mathcal{I}$;
2: **for** model i in $\mathcal{I}$ **do**
3: $minB = 1, maxB = 32, flag_i = false$;
4: **while** $minB \leq maxB$ **do**
5: $mid = \lfloor (minB + maxB)/2 \rfloor$;
6: **if** constraints (4a) and (4b) not satisfied **then**
7: $maxB = mid - 1, flag_i = true$;
8: **else if** constraints (4c) is satisfied **then**
9: $b_i = mid, minB = mid + 1$;
10: **else** break;
11: **end if**
12: **end while**
13: **if** $flag_i == false$ **then**
14: return $\{b_i\}_{i \in \mathcal{I}}, false$;
15: **end if**
16: **end for**
17: return $\{b_i\}_{i \in \mathcal{I}}, true$;

Algorithm Details. Our binary-search based batch decision algorithm *BDA* is shown in Alg. 1. *BDA* takes GPU provisioning, model arrival rate, and predicted task latency as inputs and outputs the batch size and decision status. Let $flag_i$ indicate whether the algorithm *BDA* can find the optimal batch size that satisfies the constraints for model i. Line 1 initializes the decision variables. For each model i, the algorithm performs a binary search over the batch size range from 1 to 32 (lines 2-16). Current batch size has three non-overlapping cases. i) Any of constraints (11a) and (11b) is not satisfied, indicating that the batch size is set too large, BDA updates the $maxB$ to be the $mid - 1$ (lines 6-7). ii) All

constraints are satisfied, the algorithm updates b_i to the current batch size, $flage_i$ to true and increases the batch size to search for larger throughput (lines 8-9). iii) None of the batch sizes satisfy the constraints for a given model, *BDA* stops searching for that model and returns false (lines 10-16). Finally, line 17 returns the result values.

5.4 *TS*: Intelligent Scheduler

We introduce a two-layer intelligent scheduler, *TS* , which is based on SAC-D to address the problem (1). *TS* comprises four key components that work together to improve performance. *i)* An actor-critic architecture with an actor network, Q-networks, and target Q-networks. The clipped double Q-networks technique is applied to both Q-networks and target Q-networks to avoid overestimation of Q-values [4]. *ii)* A off-policy with experience replay technique to expedite the efficiency of convergence. The agent's experiences are stored into replay buffer, and a subset of them is randomly sampled to update the network parameters during the training phase. *iii)* Discrete action space suited for our MDP problem. To address decision elements, the actor network's output layer employs discretization, while the Q-networks output Q-values for each possible action. *iv)* An entropy regularization term to ensure exploration and stability. Our agent aims to find a policy π^* that maximizes both cumulative reward and entropy objective:

$$\pi^*_\phi = \arg\max_\pi \sum_{t\in\mathcal{T}} \mathbb{E}_{(s_t,a_t)\sim\zeta_\pi}[\gamma^t(r_t + \tau\mathcal{H}(\pi(.|s_t)))], \quad (5)$$

where ζ_π represents the distribution of trajectories induced by policy π, the temperature parameter τ is utilized to balance the entropy and reward objectives, and $\mathcal{H}(\pi(.|s_t)$ denotes the entropy of policy π at state s_t. Moreover, as mentioned in Sec. 5.2, *TS* also performs normalization on the state and reward to make the training of the neural network more stable and improve the algorithm's robustness.

Parameter Update. In the training phase of our algorithm, the agent selects a mini-batch of experiences $\mathcal{F}_j$ randomly from the replay buffer $\mathcal{F}$ to update the parameters of the actor network and critic networks. For a given transition $\{s_j, a_j, r_j, s_{j+1}\}_{j\in\mathcal{F}_j}$, the Q-networks $\{Q_{\theta_1}, Q_{\theta_2}\}$'s parameters can be trained independently by minimize the soft Bellman residual:

$$J_Q(\theta_v) = \mathbb{E}_{(s_j,a_j)\sim\mathcal{F}_j}[\frac{1}{2}(\hat{Q}_j - \min_{v=1,2} Q_{\theta_v}(s_j))^2], v = 1, 2, \quad (6)$$

with target Q-values computed by target Q-networks $\{Q_{\theta'_1}, Q_{\theta'_2}\}$:

$$\hat{Q}_j = r_j + \gamma(\min_{v=1,2} Q_{\theta'_v}(s_{j+1}) - \tau\log\pi_\phi(s_{j+1})). \quad (7)$$

Using the Stochastic Gradient Descent (SGD) method, the parameters $\{\theta_1, \theta_2\}$ can be optimized in the direction of $\nabla_{\theta_v} J_Q(\theta_v)$.

Utilizing the Q-values obtained from Q-networks, the objective function to update the actor network is defined as:

$$J_\pi(\phi) = \mathbb{E}_{s_j\sim\mathcal{F}_j}[\pi_\phi(s_j)^T[\tau\log(\pi_\phi(s_j)) - \min_{v=1,2} Q_{\theta_v}(s_j)]]. \quad (8)$$

The temperature parameter τ can be updated by minimizing the entropy objective function, which can be expressed as:

$$J(\tau) = \pi_\phi(s_j)^T[-\tau(\log(\pi_\phi(s_t)) + \hat{H})], \quad (9)$$

where $\hat{H}$ is a constant and indicates the target entropy. The parameters of the actor network and the temperature are updated using the SGD method similar to the Q-networks.

Algorithm 2 Intelligent Scheduler (*TS*)

1: Initialize actor network with parameter ϕ, Q-networks with parameters θ_1, θ_2, and target Q-networks with $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$,
2: Initialize an empty replay buffer $\mathcal{F}$, and hyperparameters γ, τ, κ ℓ_Q, ℓ_π, ℓ_τ, $\hat{\mathcal{H}}$;
3: **for** *episode* = 1 to E **do**
4: Initialize the environment, and receive the initial state s_1;
5: **for** t = 1 to T **do**
6: Select action with actor's policy $a_t \sim \pi_\phi(s_t)$;
7: $b, flag = BDA(a_t)$;
8: Apply the actions $\{a_t, b\}$ in the environment, receive the reward r_t, and observe the next state s_{t+1};
9: Store $\{s_t, a_t, r_t, s_{t+1}\}$ into replay buffer $\mathcal{F}$;
10: Sample a mini-batch of experiences $\{s_j, a_j, r_j, s_{j+1}\}_{j\in\mathcal{F}'}$ from replay buffer $\mathcal{F}$ Randomly;
11: Update Q-networks: $\theta_v \leftarrow \theta_v - \ell_Q\nabla_{\theta_v} J_Q(\theta_v), v = 1, 2$;
12: Update policy weight: $\phi \leftarrow \phi - \ell_\pi\nabla_\pi J_\pi(\phi)$;
13: Update temperature: $\tau \leftarrow \tau - \ell_\tau\nabla_\tau J(\tau)$;
14: Update target Q-networks: $\theta'_v \leftarrow \kappa\theta_v + (1-\kappa)\theta'_v, v = 1, 2$.
15: **end for**
16: **end for**

Algorithm Design. Our proposed intelligent scheduler, *TS*, is presented in Alg. 2. The workflow of *TS* is as follows:

Initialization Phase. Firstly, the parameters of the actor network, Q-networks, and target Q-networks are initialized (lines 1). Moreover, line 2 initializes an empty experience replay buffer $\mathcal{F}$ and related hyperparameters. *TS* iterates E episodes. At the start of each episode, the environment is initialized and the agent observes the initial state s_1 (line 4).

Environment Phase. Based on state information $s_t = \{\{C_s, B_s, u_k\}_{k\in\mathcal{K}}, \{\lambda_i, G_i\}_{i\in\mathcal{I}}\}$, the agent decides the discrete actions of GPU provisioning $a_t = \{y_{i,k}, \rho_{i,k}\}_{i\in\mathcal{I},k\in\mathcal{K}}$, with its actor policy π_ϕ (line 6). Then, the decisions of batch size $b = \{b_i\}_{i\in\mathcal{I}}$ for each DNN model, is obtained by algorithm *BDA* (line 7). After receiving the actions $\{a_t, b\}$ from the agent, *task buffer* and *GPU cluster* apply them to provide DNN inference service. Following a decision step, the agent receives the corresponding reward r_t and observes the next state s_{t+1} (line

8). Finally, the transition tuple $\{s_t, a_t, r_t, s_{t+1}\}$ is stored into replay buffer $\mathcal{F}$ in line 9.

Training Phase. During the training stage, the agent executes the subsequent procedures to update networks. Firstly, line 10 samples a mini-batch of experiences $\mathcal{F}_j$ randomly from the replay buffer $\mathcal{F}$. Then, the parameters of Q-networks θ_v, the policy weight ϕ and the temperature τ are updated by minimizing Eq. (6), Eq. (8) and Eq. (9), with learning rates l_Q, l_ϕ and l_τ, respectively. Finally, the parameters of target Q-networks are updated by employing a soft-updating method, *i.e.*, $\theta'_v \leftarrow \kappa\theta_v + (1-\kappa)\theta'_v, v = 1, 2$, where τ denotes the update factor.

5.5 Implementation

Prototype. We implemented *InSS* based on a GPU inference service software prototype [24], which contains over 20k lines of C++ code. We used Python to implement the scheduling algorithm for *InSS*'s *scheduler TS*, which was designed based on SAC-D, resulting in over 1k lines of code. Furthermore, the DNN modes were implemented using PyTorch, which is widely adopted in the ML communities. We also designed C++ interfaces to integrate the scheduler module and deploy DNN models. The complete source code of *InSS* prototype is available on GitHub[4].

Workflow. Prior to the execution of *InSS*, data collection conducts on different types of GPU devices, as mentioned in Sec. 4. Then, *InSS* is launched to provide DNN inference services. *InSS* periodically executes the scheduler *TS*. Specifically, when the model arrival rate changes, *TS* first runs the *BDA* algorithm to adjust the batch settings (case 1). When *BDA* returns $false$ (case 2), it indicates that the current model deployment and resource allocation cannot meet the task requirements. Therefore, the actor network is triggered to perform resource reallocation and calculate the batch size once again.

GPU Resource Reallocation. NVIDIA's MPS allows the allocation of specific computational resources to processes. However, changing the resource allocation requires creating a new inference service process. Creating a new inference service process involves starting new process, loading kernels, loading model, and model warming up. Therefore, adjusting batch settings (case 1) can be done in real-time, while resource reallocation (case 2) utilizes a shadow mechanism, which involves preparing a new process, activating it, and finally killing the original process. In this way, we hide the overhead of preparing a new process, although running the shadow and original processes concurrently incurs memory overhead. The current inference tasks will not be interrupted.

6 EVALUATION

[4]https://github.com/code-InSS/InSS

6.1 Experiments Setup

GPU Cluster. Our GPU cluster consists of three inference servers, each of which is a desktop PC equipped with 12 CPU cores, 16GB RAM, and a dual-port 1GbE NIC. One of the servers is equipped with two NVIDIA RTX 2060 Super GPUs, which have 2176 CUDA cores, 34 SM counts, 8000MB GDDR6, 4MB L2 Caches, and 448.0 GB/s memory bandwidth. The other two servers are each equipped with one NVIDIA RTX 2060 GPU, which has 1920 CUDA cores, 30 SM counts, 6000MB GDDR6, 3MB L2 Caches, and 336.0 GB/s memory bandwidth. All of these GPUs support MPS technology. In addition, another desktop PC serves as both the *client* and the *scheduler*, which does not provide DNN inference service. It is responsible for generating and sending inference task requests, as well as running the scheduling program of the *scheduler*. All desktop PCs can communicate with each other.

Workload. Eight widely-known DNN models are selected for providing inference services, *i.e.*, AlexNet (A) [26], ResNet50 (R) [44], VGG16 (V) [39], MnasNet (M) [41], MobileNet (M1) [19], EfficientNet (E) [42], Resnet18 (R1), and VGG19 (V1). Both training and inference stages utilize the widely used ImageNet dataset [13]. This dataset contains more than 1 million images, each with a resolution of 224x224 pixels and 3 channels. Based on the GPU execution latency of DNN models and the SLO requirements presented in [5], we set the SLO requirements for the eight selected models to [25, 80, 150, 50, 60, 40, 60, 200] ms. To model the workload arrival rate for each DNN model, we adopt a Poisson random distribution to sample inter-arrival time. This approach is supported by prior study [51] that shows a Poisson distribution can effectively approximate real-world arrival rates.

Algorithm Networks. Our DRL framework employs a four-layer neural network structure for both the actor and critic networks of the agent, which include an input layer, an output layer, and two hidden layers. The hidden layers consist of 512 and 256 neurons, respectively. We set the penalty factor of reward to $\omega = 100$. other parameters of *TS* are listed in TABLE 1, where ι represents the output dimension of the actor network. Our DRL framework is also implemented using PyTorch.

Table 1: Parameter Setting of *TS*

Parameter	Value	Parameter	Value
Number of episodes E	1500	Number of steps T	200
Replay buffer size $\lvert\mathcal{F}\rvert$	10000	Mini-batch size $\lvert\mathcal{F}_j\rvert$	100
Learning rate $\ell_Q, \ell_\pi, \ell_\tau$	0.0001	Discount factor γ	0.99
Temperature initial τ	1.0	Target entropy $\hat{H}$	-log(1 + ι)
Soft update factor κ	0.01	Optimizer	Adam
Hidden layer act.	ReLU	Actor output act.	Softmax

Baselines. To evaluate the performance of *InSS*, we compare it with the following three baselines.

- *Gpulet* [5]: It divides a GPU into two pieces. It allocates GPU resources by maximizing request throughput, and places DNN models on the most suitable GPU. The batch decision is made by traversing to find the maximum batch size with a latency less than SLO/2.
- *GSLICE* [14]: This strategy adjusts the allocation of GPU resources and batch sizes by traversing the average latency and throughput of the DNN models.
- *iGniter* [45]: It calculates the batch size and minimum resource allocation based on parameters of latency prediction and arrival rate. It then adjusts the allocation of resources and model deployments by prediction of interference.

6.2 Validation of the Latency Prediction Model

Accurate latency prediction is crucial for scheduling DNN inference tasks. Here, we evaluate the performance of our GPU execution prediction model.

Table 2: Observed and predicted GPU execution latency with different batch sizes

Model / Batch size		1	4	8	16
R	obs (ms)	8.690	23.285	43.422	71.456
	pre (ms)	8.905	23.888	43.405	71.396
	error (%)	**2.474**	**2.588**	**0.041**	**0.084**
A&R*	obs (ms)	2.848	5.674	8.898	10.191
	pre (ms)	2.894	5.750	8.975	10.487
	error(%)	**1.620**	**1.327**	**0.868**	**2.909**
V&A*&R*	obs (ms)	18.196	62.009	120.382	176.308
	pre (ms)	18.207	59.984	117.064	170.879
	error (%)	**0.060**	**3.265**	**2.756**	**3.080**

Impact of Batch Size. Table 2 presents the GPU execution latency of ResNet50, AlexNet, and VGG16 with different batch sizes and co-located DNN models. In Table 2, all DNN models are deployed with an equal allocation of 30% GPU resources. Here, we use X* to indicate that DNN model X is fixedly scheduled, *i.e.,* its batch size is fixed at 4. It can be observed that *InSS* can accurately predict GPU execution latency with a low prediction error ranging from 0.041% to 2.588% for ResNet50, 0.868% to 2.909% for AlexNet, and 0.060% to 3.265% for VGG16. Additionally, as the number of co-located models increases, the prediction accuracy tends to decrease due to more complex resource contention.

Impact of Allocated Resources. The prediction results of MnasNet, Resnet18, VGG19, EfficientNet, and MobileNet are presented in Table 3. The batch size of all DNN models is fixed at 4. Let X* denote DNN model X with a fixed resource allocation of 18% GPU. The results show that with the change of allocated resources, the prediction error of InSS ranges from 0.214% to 3.659% for Mnasnet, 0.910% to 3.861% for Resnet18, and 0.758% to 2.870% for VGG19. To further validate the impact of co-located models with different allocated resources on the current model, we also tested the fourth group of data in Table 3, which shows an error range of 1.372% to 3.818%. This confirms that the more resources are allocated, the greater the interference to the co-located model, and *InSS* can predict such changes, as we mentioned in Sec. 4. It is worth noting that the prediction results for the other models are similar, we do not present redundant data for all models.

Table 3: Observed and predicted task inference latency with different allocated resources

Models / Resource(%)		6	24	42	60
M	obs (ms)	13.686	4.241	3.468	3.409
	pre (ms)	13.185	4.101	3.461	3.395
	error (%)	**3.659**	**3.316**	**0.214**	**0.413**
R1&M*	obs (ms)	43.016	11.446	7.422	5.483
	pre (ms)	41.909	11.586	7.136	5.433
	error (%)	**2.574**	**1.228**	**3.861**	**0.910**
V1&R1*&M*	obs (ms)	358.270	91.690	53.947	39.411
	pre (ms)	355.554	90.465	53.071	38.279
	error (%)	**0.758**	**1.336**	**1.624**	**2.870**
E*&M1	obs (ms)	15.515	16.170	16.958	17.516
	pre (ms)	16.108	16.416	16.725	17.114
	error (%)	**3.818**	**1.522**	**1.372**	**2.294**

6.3 Evaluation Results

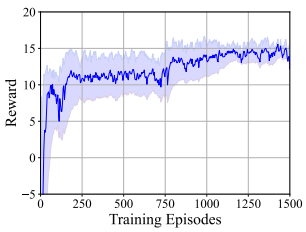

Figure 13: Convergence performance of *TS*.

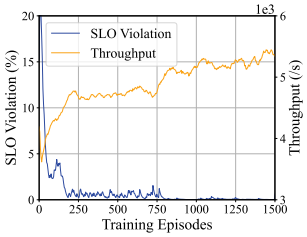

Figure 14: Convergence performance details of *TS*.

Convergence of TS. The convergence performance of the scheduler, *TS*, is presented in Fig. 13. The plot shows that the reward value gradually increases with an increase in the number of training episodes until it reaches a relatively stable value. It validates that *TS* converges after parameter iterations for 800 episodes. To provide further analysis on the convergence effect of *TS*, we have plotted the SLO violation and throughput for each iteration in Fig. 14. The results demonstrate that with an increase in the number of episodes, the SLO violation decreases, while the throughput gradually increases. These observations further reinforce the notion that *TS* exhibits a strong convergence effect, a high throughput with less than 1% SLO violation.

System Throughput. After well offline training, we evaluate the performance of *InSS* on a system equipped with

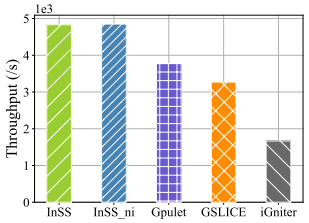

Figure 15: Throughput of different algorithms.

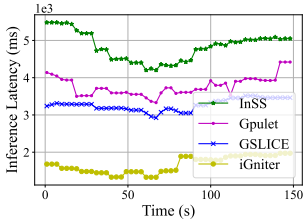

Figure 16: Tthroughput over time of different algorithms.

4 GPUs for providing inference services to 6 DNN models. As shown in Fig. 15, we compared the average system throughput of *InSS* with three baselines. *InSS-ni* represents *InSS* without interference prediction. It can be observed that our proposed *InSS* achieves higher throughput. Specifically, compared with *Gpulet*, *GSLICE*, and *iGniter*, *InSS* achieved an average improvement of 28.1%, 47.7%, and 189.7%, respectively. Moreover, the difference in throughput performance between *InSS* and *InSS-ni* is not significant. However, the impact of interference prediction is mainly reflected in SLO violations, which is described in detail later. To further demonstrate the performance of *InSS*, we plotted the system throughput over time in Fig. 16. The results show that *InSS* adjusts its decision based on the dynamic arrival of tasks and maintains high throughput.

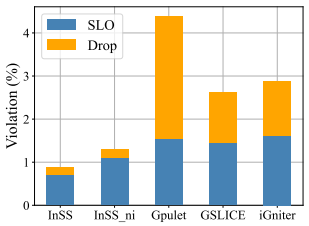

Figure 17: SLO violation under different algorithms.

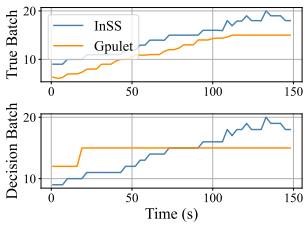

Figure 18: Comparison of decision and true batch over time.

SLO Violation. To examine the quality of inference service provided by the system, we present the SLO violations of different algorithms in Fig. 17. In the figure, "SLO" indicates the percentage of tasks whose task latency exceeds its SLO, and the percentage of tasks dropped by the system due to long queuing latency that cannot meet the SLO is labeled as "Drop". It can be seen that *InSS* guarantees the service quality of 99% for inference tasks. Although this impact may seem small, we argue that considering interference is necessary since a scheduler must ensure SLO at all times. The reason for SLO violation in *iGniter* is that its resource allocation heavily relies on the accuracy of latency prediction. When there is a bias in the prediction, the violation will increase accordingly. In contrast, the GPU provisioning of *InSS* is computed based on SAC-D and is not solely dependent on latency prediction. Therefore, it has higher fault tolerance. There are two reasons why the SLO violation of the *Gpulet* is relatively high. Firstly, *Gpulet* does not consider interference, which may cause task latency beyond the SLO. Secondly, its batch decision simply satisfies GPU execution time < SLO/2 and sets a timeout of SLO/2. Thus, its batch decision is not precise, and the calculated throughput cannot be met, resulting in task drops. To provide a clearer illustration, we plot the decision and true batch size of ResNet50 under *InSS* and *Gpulet* over time in Fig. 18. It shows that the batch decisions made by *Gpulet* are not useful in practice, as the true batch size is determined by the task arrival rate and the half of model's SLO. In contrast, the decision batch and true batch values for *InSS* are almost identical. To further demonstrate the necessity of considering the queuing latency, we applied the BDA algorithm to *Gpulet*. The results of *BDA-Gpulet* achieved a throughput of 4188, which is 11% higher than the original *Gpulet* while maintaining a similar SLO violation. This is because the precise batch decisions lead to more accurate timeout values, which in turn increase the queuing latency and further improve throughput. *GSLISE* also faces the same two problems as *Gpulet*, but due to the different resource allocation strategies, their SLO violations differ in different scenarios.

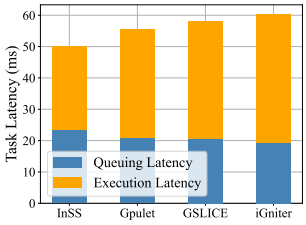

Figure 19: Task latency of ResNet50 under different algorithms.

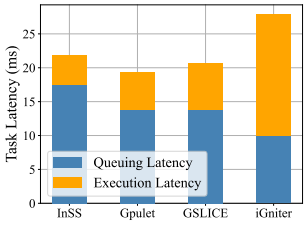

Figure 20: Task latency of MnasNet under different algorithms.

Task Latency. Fig. 19 and Fig. 20 illustrates the average task latency of ResNet50 and MnasNet under different algorithms, respectively. It can be observed that *InSS* has longer queuing latency but lower execution latency compared to baselines. This is attributed to the capability of *InSS* to accurately predict both queuing latency and GPU execution lantency, enabling it to make superior batch decisions. Moreover, *InSS* effectively allocates resources while balancing throughput and latency.

Heavy Workload. In our experiments, we find that as the workload increased, both *Gpulet* and *GSLICE* required more GPUs. Therefore, we test the performance of serving 8 models simultaneously. While running, *Gpulet* and *GSLICE* both use 5-6 GPUs, while *iGniter* uses only 3 GPUs. We configured *InSS* with 4 (*InSS_4*) and 5 (*InSS_5*) GPUs. The system's average throughput and total SLO violation data are presented in Fig. 21 and Fig. 22, respectively. From the figures, we can observe that *InSS_4* achieves comparable throughput to *Gpulet* and *GSLICE* while using fewer GPUs, and *InSS_5* outperforms

baselines in terms of system throughput. Compared to the baselines, *InSS* can improve the throughput by up to 180%. Moreover, the system running on *InSS* offers the best service quality. The SLO violation of *InSS* is minimal and always less than 1%.

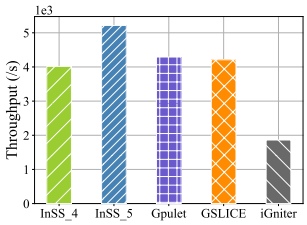

Figure 21: Throughput of different algorithms.

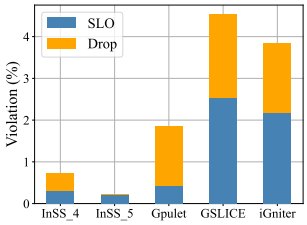

Figure 22: SLO violations under different algorithms.

Furthermore, we simply compare the allocated GPU resources of *InSS* with three baselines in Fig. 23. It can be seen that *InSS* efficiently utilizes the configured GPU resources. Combined with the data in Fig. 21 and Fig. 22, it can be concluded that *InSS* achieves higher efficiency in GPU resource allocation, enabling it to better utilize the limited GPU resources.

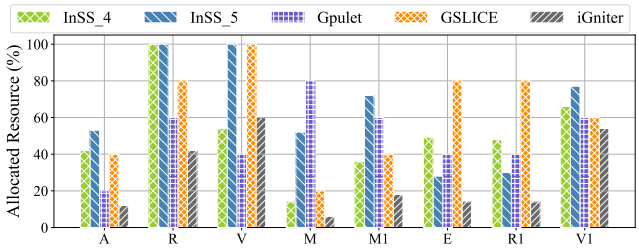

Figure 23: Task latency of ResNet50 under different algorithms.

7 RELATED WORK

Inference with batching. In this scenario, each GPU serves only one DNN inference at a time, hence the focus is primarily on batching [1, 6, 10, 16, 21]. *Ebird* [11] improves the response time and throughput by eliminating unnecessary waiting, enabling overlap, and elastically organizing inference requests using an elastic batch scheduler. *BatchDVFS* [31] combines dynamic batching and DVFS techniques to control power consumption and improve throughput in DNN inference on GPUs. However, using only batch processing has limited improvement in system throughput and resource utilization efficiency.

Inference with GPU temporal sharing. Temporal scheduling is common in GPU inference [29, 43, 46]. *Nexus* [38] enhances GPU cluster performance and utilization with batching-aware scheduling and early drop mechanisms, using a fixed batch size for each epoch. *Clockwork* [17] uses predictive execution times to order user requests and proposes fine-grained request-level scheduling to meet their SLO. Temporal sharing can improve GPU utilization but not as effectively as spatial sharing, and it also incurs costs for model switching and memory usage. *InSS* employs temporal and controllable spatial sharing to enhance GPU utilization while considering constraints on SLO and memory usage.

Inference with GPU spatial sharing. There are some efforts on GPU resource spatial sharing [7, 12, 23, 25, 27]. *Gpulet* [5] supports up to two co-located models sharing one GPU and builds a linear regression model for predicting the latency increases based on L2 Cache and DRAM bandwidth, but it requires a large amount of data analysis and has heavy overhead. *GSLICE* [14] focuses on resource allocation for GPUs and does not consider performance interference between co-located models. *iGniter* [45] is designed to minimize GPU usage by calculating batch size and minimum resource allocation using two formulas and deploying models based on interference prediction. However, scheduling overhead, co-located performance interference, and queuing latency are all vital factors for task latency. The prior works above have not comprehensively addressed them. In this work, we combine batch processing and GPU spatial-temporal sharing techniques and propose an orchestrator that achieves predictable task latency, efficient resource utilization, high throughput, and excellent service quality.

8 CONCLUSION

This paper proposes and implements *InSS*, an intelligent scheduling orchestrator on multi-GPU Inference servers for achieving high throughput via spatio-temporal sharing. We first present an analytical performance model which predicts the GPU execution latency based on the inference of co-located DNN workloads on the same GPU and compute the queuing model. *InSS* is built on the interference-aware analytical model to intelligently decide the model placement, GPU resource allocation and adjust the batch size, with the goal of maximize the system throughput. Extensive prototype experiments verify the efficiency of *InSS*. *InSS* can increase the throughput by up to 189%, while satisfying the SLOs.

REFERENCES

[1] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *Proc. of SC*.

[2] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. 2017. Real-time video analytics: The killer app for edge computing. *computer* (2017).

[3] Michael Braun, Anja Mainz, Ronee Chadowitz, Bastian Pfleging, and Florian Alt. 2019. At your service: Designing voice assistant personalities to improve automotive user interfaces. In *Proc. of CHI*.

[4] Shaotao Chen, Xihe Qiu, Xiaoyu Tan, Zhijun Fang, and Yaochu Jin. 2022. A model-based hybrid soft actor-critic deep reinforcement learning algorithm for optimal ventilator settings. *Information Sciences* (2022).

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

[6] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. 2021. Lazy batching: An sla-aware batching system for cloud machine learning inference. In *Proc. of IEEE HPCA*.

[7] Marcus Chow, Ali Jahanshahi, and Daniel Wong. [n.d.]. KRISP: Enabling Kernel-wise RIght-sizing for Spatial Partitioned GPU Inference Servers. In *Proc. of IEEE HPCA*.

[8] Petros Christodoulou. 2019. Soft Actor-Critic for Discrete Action Settings. *CoRR* (2019).

[9] William S Cleveland and Susan J Devlin. 1988. Locally weighted regression: an approach to regression analysis by local fitting. *Journal of the American statistical association* (1988).

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

[11] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Mingyi Guo. 2019. Ebird: Elastic batch for improving responsiveness and throughput of deep learning services. In *Proc. of IEEE ICCD*.

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

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

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

[15] Arnaud Fréville. 2004. The multidimensional 0–1 knapsack problem: An overview. *European Journal of Operational Research* (2004).

[16] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency RNN inference with cellular batching. In *Proc. of EuroSys*.

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

[18] Yi-Chao He, Xi-Zhao Wang, Yu-Lin He, Shu-Liang Zhao, and Wen-Bin Li. [n.d.]. Exact and approximate algorithms for discounted {0-1} knapsack problem. *Information Sciences* ([n. d.]).

[19] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

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

[21] Yoshiaki Inoue. [n.d.]. Queueing analysis of GPU-based inference servers with dynamic batching: A closed-form characterization. *Performance Evaluation* ([n. d.]).

[22] Folasade Olubusola Isinkaye, Yetunde O Folajimi, and Bolande Adefowoke Ojokoh. [n.d.]. Recommendation systems: Principles, methods and evaluation. *Egyptian informatics journal* ([n. d.]).

[23] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. 2018. Dynamic space-time scheduling for gpu inference. *arXiv preprint arXiv:1901.00041* (2018).

[24] Casys Kaist. 2022. *glet*. https://github.com/casys-kaist/glet.git.

[25] Yunseong Kim, Yujeong Choi, and Minsoo Rhu. 2022. PARIS and ELSA: an elastic scheduling algorithm for reconfigurable multi-GPU inference servers. In *Proc. of DAC*.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Proc. of NIPS* (2012).

[27] Baolin Li, Viiay Gadepally, Siddharth Samsi, and Devesh Tiwari. 2022. Characterizing multi-instance gpu for machine learning workloads. In *Proc. of IEEE IPDPSW*.

[28] Ming Liu, Tao Li, Neo Jia, Andy Currid, and Vladimir Troy. 2015. Understanding the virtualization" Tax" of scale-out pass-through GPUs in GaaS clouds: An empirical study. In *Proc. of IEEE HPCA*.

[29] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient GPU cluster scheduling. In *Proc. of USENIX NSDI*.

[30] Steven J Miller. 2006. The method of least squares. *Mathematics Department Brown University* (2006).

[31] Seyed Morteza Nabavinejad, Sherief Reda, and Masoumeh Ebrahimi. 2022. Coordinated batching and dvfs for dnn inference on gpu accelerators. *IEEE Transactions on Parallel and Distributed Systems* (2022).

[32] NVIDIA. 2015. *CUDA Multi-Streams*. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

[33] NVIDIA. 2020. *NVIDIA Multi Instance GPU (MIG)*. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

[34] NVIDIA. 2020. *NVIDIA Multi-Process Service (MPS)*. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[35] NVIDIA. 2021. *NVIDIA Nsight Compute*. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html.

[36] NVIDIA. 2021. *NVIDIA Nsight Systems*. https://developer.nvidia.com/nsight-systems.

[37] Zhao-Wei Qiu, Kun-Sheng Liu, and Ya-Shu Chen. 2022. BARM: A Batch-Aware Resource Manager for Boosting Multiple Neural Networks Inference on GPUs With Memory Oversubscription. *IEEE Transactions on Parallel and Distributed Systems* (2022).

[38] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proc. of ACM SOSP*.

[39] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[40] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. 2021. Serving DNN models with multi-instance gpus: A case of the reconfigurable machine scheduling problem. *arXiv preprint arXiv:2109.11067* (2021).

[41] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware

neural architecture search for mobile. In *Proc. of IEEE CVPR*.

[42] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *Proc. of ICML*. PMLR.

[43] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *Proc. of USENIX OSDI)*.

[44] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proc. of IEEE CVPR*.

[45] Fei Xu, Jianian Xu, Jiabin Chen, Li Chen, Ruitao Shang, Zhi Zhou, and Fangming Liu. 2022. iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in the Cloud. *IEEE Transactions on Parallel and Distributed Systems* (2022).

[46] Peifeng Yu and Mosharaf Chowdhury. 2020. Fine-grained GPU sharing primitives for deep learning applications. *Machine Learning and Systems* (2020).

[47] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving.. In *Proc. of USENIX ATC*.

[48] Wei Zhang, Quan Chen, Ningxin Zheng, Weihao Cui, Kaihua Fu, and Minyi Guo. 2021. Toward QoS-awareness and improved utilization of spatial multitasking GPUs. *IEEE Trans. Comput.* (2021).

[49] Weiting Zhang, Dong Yang, Haixia Peng, Wen Wu, Wei Quan, Hongke Zhang, and Xuemin Shen. 2021. Deep Reinforcement Learning Based Resource Management for DNN Inference in Industrial IoT. *IEEE Transactions on Vehicular Technology* (2021).

[50] Xu Zhang, Zheng Zhao, Qing An, Yuan Lin, Liang Zhi, and Yi Chu. 2023. *Meituan Visual GPU Inference Service Deployment Architecture Optimization Practice.* https://tech.meituan.com/2023/02/09/inference-optimization-on-gpu-by-meituan-vision.html.

[51] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. 2016. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. *ACM SIGARCH Computer Architecture News* (2016).