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

Ziyi Han, Ruiting Zhou, *Member, IEEE,* Chengzhong Xu, *Fellow, IEEE,* Yifan Zeng, and Renli Zhang

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 unique challenges. First, interference between co-located DNN models deployed on the same GPU must be accurately modeled. Second, inference tasks arrive dynamically online, and each task needs to be served within a bounded time to meet the service-level objective (SLO). Third, the problem of fragments has become more serious. To address the above three 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 task latency. ii) A two-stage intelligent scheduler is tailored to jointly optimize the model placement, GPU resource allocation and adaptively decides batch size by coupling the latency analytical model. Our prototype implementation on four NVIDIA A100 GPUs shows that *InSS* can improve the throughput by up to 86% compared to the state-of-the-art GPU schedulers, while satisfying SLOs. We further show the scalability of *InSS* on 64 GPUs.

Index Terms—DNN inference, GPU Resource Management, Online Scheduling

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 [66]. 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 [4], 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], [61]. Therefore, it is of great significance to improve the efficiency of GPU usage.

To enhance GPU utilization, batch processing [39] and resource sharing [57] are two key strategies. By combining multiple inference tasks that request the same DNN model through dynamic batch, the utilization of GPU resources can be improved [11]. Temporal sharing [45] and spatial sharing [63] 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. Among existing spatial sharing methods, NVIDIA Multi-Process Service (MPS) allocates a limited percentage of GPU resources (e.g., 30%) to each DNN model, which has the potential to achieve more comprehensive resource utilization and enhance system throughput.

MPS-based spatio-temporal resource sharing introduces unique challenges when applying it to inference services on GPU clusters. *First*, there is interference between co-located DNN models running on the same GPU [1]. This is because MPS only isolates streaming multiprocessors (SMs), while other resources like L2 Cache and PCIe bandwidth are shared by co-located DNN models [55]. Given the compute-intensive nature of inference tasks, they are especially sensitive to interference. As indicated in Sec 2.2, interference can increase the latency by up to 22.7%. Due to significant differences in hardware and memory architecture between CPUs and GPUs, interference considerations based on the CPU cannot be directly applied to the GPU cluster. Therefore, it is essential to formulate an interference model tailored to the unique hardware architecture and characteristics of GPUs. *Second*, inference tasks must be completed within the required time to meet SLOs. Therefore, striking a balance between parallel processing (batch processing) and achieving low-latency responses is crucial for maintaining consistent service quality. In practice, inference tasks arrive dynamically online, requiring rapid adjustment in decisions to adapt to environment changes without compromising the quality of inference services. *Third*, the fragment issue is a crucial consideration in resource allocation [56]. Implementing fine-grained GPU resource allocation based on MPS introduces a more diverse range of partitioning choices. In contrast to the allocation of a small number of CPU cores and coarse-grained resource allocation, the allocation pro-

- *Z. Han, R. Zhou, Y. Zeng and R. Zhang are with the School of Cyber Science and Engineering, Wuhan University, Wuhan, China. E-mail: {ziyihan, ruitingzhou, yifanzeng, zhang_rl}@whu.edu.cn.*
- *R. Zhou is also with the School of Computer Science and Engineering, Southeast University, Nanjing, China. E-mail: ruitingzhou@seu.edu.cn.*
- *C. Xu is with the Faculty of Science and Technology, University of Macau, Taipa 999078, China E-mail: czxu@um.edu.mo.*

This work is supported in part by the NSFC Grants (62072344, U20A20177and 62232004).
(Corresponding author: Ruiting Zhou.)

cess is more intricate, resulting in a more complex fragment issue. Hence, classical cloud resource sharing approaches are not applicable to GPU clusters with MPS-based spatio-temporal sharing.

To improve the utilization of GPU resources, some works apply batch processing [1], [7], [11], [18], [24], [39], and some studies enable spatial sharing [8], [13], [25], [27], [29] or temporal sharing [20], [34], [45], [52], [59] of GPU resources. But most of them do not consider the interference and resource fragments, *e.g., GSLICE* [15]. Some works [6], [23], [47] take into account interference but require storing latency data under various settings, incurring significant overhead and unsuitability for online tasks. Only *iGniter* [55] proposes a light-weighted interference model, relying on quadratic functions for scheduling, which poses scalability and replaceability challenges. Moreover, existing approaches mainly employ simple greedy strategies or tackle scheduling through solving constrained combinatorial optimization problems [6], [15], [19], [60], which exhibit low efficiency and overlook the need for low scheduling overhead. Furthermore, their approaches can only be applied to small-scale clusters, resulting in limited scalability.

To this end, we propose and implement an Intelligent Scheduling orchestrator for multi-GPU inference servers with spatio-temporal Sharing (*InSS*) to maximize the system throughput while guarantees the SLO requirement. To the best of our knowledge, *InSS* is the **first work** that characterizes the task latency while considering interference and batch queuing latency, and show the capability of deep reinforcement learning (RL) to automatically learn the implicit relation between coupled scheduling decisions and make decisions rapidly for inference tasks with low-latency requirements in online services. Considering the time and resource costs of training RL models in real clusters, a prediction model is designed to obtain interference-aware GPU execution latency[1] and queuing-aware task latency[2]. Due to the large decision space and distinctive nature of decision variables, the learning process demands considerable time for convergence. To address this, *InSS* discretizes decision variables into both discrete and continuous components and employs a two-stage intelligent scheduler. Specifically, We make the following contributions.

First, *InSS* builds an analytical performance model to predict task latency. It first presents light-weighted prediction model for GPU execution latency which captures the influence of heterogeneous system configuration and the interference among co-located DNN inference workloads. It also models the queuing delay due to waiting for batch processing which is then used to calculate the task latency. The prediction model exhibits *excellent scalability*, requiring only a few sets of data to derive latency for co-located models. This is because interference-related resource utilization is model-specific and can be independently predicted for each model. Only two coefficients for interference between co-located models are required to fit.

Second, *InSS* employs a two-stage intelligent scheduler, *TS*, to jointly optimize model placement (discrete variables), resource allocation and batch size (continuous variables). In the global stage, *InSS* presents a Soft Actor-Critic-Discrete (SAC-D) based approach to learn the deployment for each model. This framework is adaptable to hierarchical RL based on GPU grouping, providing scalability in large-scale clusters. In the local stage, for each GPU, *InSS* utilizes Twin Delayed Deep Deterministic policy gradient algorithm (TD3) to learn the resource allocation and batch size for the deployed models. The design of the *action output layer of TD3* is carefully tailored to meet resource capacity constraints and mitigate resource fragments effectively. Through offline training, *InSS* can adapt to dynamic workloads and interference resulting from different co-located models, enabling swift scheduling adjustments during online service.

Third, we implement a prototype of *InSS* with ten DNN models and four NVIDIA A100 GPUs. We observe: i) *InSS* provides an accurate light-weighted prediction model for inference tasks, with an error rate of less than 5% in the worst case. ii) *InSS* can improve the throughput by 52.8%, 26.4%, and 86.1%, compared to *Gpulet* [6], *GSLICE* [15] and *iGniter* [55], respectively. iii) The SLO violation of *InSS* is always less than 0.7%, while that of the three baselines are 1.9%, 1.7%, and 1.1%, respectively. iv) *InSS* show scalability to large-scale clusters, achieving up to 66.0% throughput improvement while meeting 99% of SLO requirements.

2 BACKGROUND AND MOTIVATION

2.1 Spatial Sharing Execution

TABLE 1
Latency (ms) of MPS and MIG.

Mechanism	Allocation	Reallocation	model load & warm
MPS	1.909	1.909	2616.495
MIG	334.236	505.176	2965.799

To enhance GPU utilization, GPU suppliers like NVIDIA have developed multiple mechanisms to achieve controllable resource spatial sharing: i) Multi-Process Service (MPS) [41], [64], a logical resource partition mechanism, distributes SMs to different processes in specified percentages, such as allocating 40% and 60% to two parallel processes. However, this approach still involves shared utilization of remaining GPU resources, including L2 Cache and Dynamic Random-Access Memory (DRAM). Consequently, interference occurs when multiple models run in parallel. ii) Multi-Instance GPU (MIG) [27], [40], a physical resource partition method, achieves the partitioning of physical resources at the hardware layer, ensuring the full isolation of all resources. This ensures no interference during concurrent processes. However, full resource isolation results in significant temporal overhead for resource reallocation. Table 1 illustrates the latency differences in resource allocation and model loading under different mechanisms on NVIDIA A100 GPU. It is observed that the resource allocation latency of MIG significantly surpasses that of MPS. The reallocation process in MIG requires the prior removal of existing instances, introducing additional delays and causing the suspension of inference services. In contrast, MPS enables continuous inference services during resource reallocation, which is described in Sec. 5.5. In summary, we choose MPS mechanism to achieve controllable and flexible resource spatial sharing and continuous inference services.

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.

We further plot the inference latency and throughput for processing tasks (batch size = 8) with a variety of DNN models under different GPU resources (%) in Fig. 1. As can be seen, throughput and latency are notably influenced by the allocated resources, and their relationship is non-linear. Furthermore, when resources allocated to a model reach a certain value (*e.g.*, 60% GPU for ResNet50), the incremental benefits of additional resources on throughput and latency diminish considerably. It becomes evident that allocating an entire GPU resource to a model is often inefficient. Additionally, partial allocation of GPU resources can meet the task's SLO and deliver ample throughput. Simultaneously, the remaining GPU resources can be utilized for additional computations, enabling the concurrent execution of DNN models within a given resource constraint on a single GPU. Employing GPU resource spatial sharing through the MPS mechanism is imperative and significantly enhances GPU utilization. Therefore, the challenge lies in determining the optimal allocation of GPU resources for each DNN model.

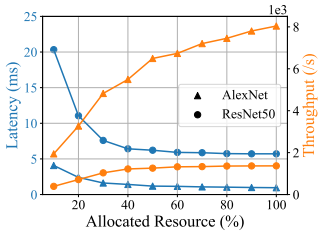

Fig. 1. Inference latency of different amounts.

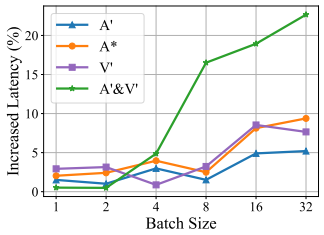

Fig. 2. increased latency of different configurations.

2.2 Interference among Co-located DNN Models

When MPS achieves the isolation of SMs, other resources of the GPU are shared, leading to interference between co-located models on the same GPU. To assess the impact of interference, Fig. 2 displays the results of experiments with ResNet50 under different system configurations. Here, AlexNet (A) and VGG16 (V) are used as co-located models, both with a fixed batch size of 8. For resource allocation, ResNet50 consistently receives 40% of the GPU, while X' denotes a 30% allocation for model X, and X* indicates a 60% allocation of the GPU. The increased latency refers to the percentage rise 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 the number of co-located models significantly impact interference values, resulting in a latency increase of up to 22.7%. Thus, interference is a non-negligible factor. Based on the inference workflow and experimental data, we consider the influence of three factors: PCIe bandwidth, L2 Cache, and DRAM. In Sec. 4, we develop a well-designed model for interference-aware latency prediction.

2.3 Scheduling Overhead

DNN inference tasks demand low latency and high real-time performance. Therefore, the scheduling mechanism should aim to make decisions in a short time, ensuring prompt responses to environmental changes while minimizing its impact on inference tasks.

We evaluate the scheduling overhead of *InSS* and three state-of-the-art algorithms: *Gpulet* [6], *GSLICE* [15], and *iGniter* [55] in a cluster with 4 GPUs and 10 models. In this paper, scheduling overhead specifically refers to the time it takes for the scheduling mechanism to make decisions. As shown in Fig. 3, the scheduling overhead of the compared algorithms is higher than that of *InSS*, even exceeding the inference latency of many tasks. Moreover, most existing algorithms offer only fixed and limited configurations (*e.g.*, 20%, 40%, 50%, 60%, 80%, 100%) [6], [15], failing to fully leverage the flexibility of MPS, which allows for resource allocation down to the granularity of SM. For example, an NVIDIA A100 GPU has 108 SMs that can be partitioned into 54 portions. With advancements in GPU technology leading to an increased number of SMs on a single GPU, the dimensions of GPU partitioning have expanded, further amplifying the search space for decision-making. This, in turn, diminishes the efficiency of current algorithms. Simultaneously, the growth in cluster size exacerbates the inefficiency of these algorithms.

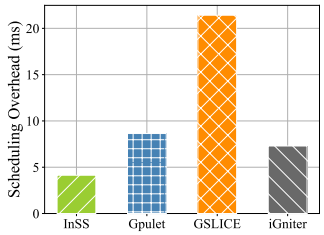

Fig. 3. Scheduling overhead of different algorithms.

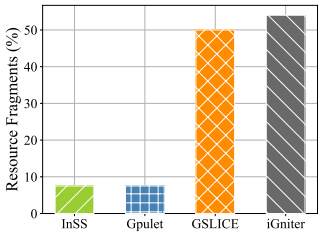

Fig. 4. Resource fragments of different algorithms.

2.4 Resource Fragments

Resource fragments are key challenges in GPU systems with spatial sharing, hindering optimal system performance by impeding efficient resource allocation and utilization. Specifically, resource fragments refer to the percentage of unused resources on GPUs with running models after scheduling. For instance, if 60% of a GPU's resources are allocated to run ResNet50 and the remaining 40% is left unused, the 40% of resource is considered as a fragment. In contrast, if an entire GPU remains idle without any model deployment, it incurs minimal operational costs and thus is not regarded as a fragment. A notable observation is the general absence of consideration for resource fragments in prior studies. We evaluate the resource fragments of *InSS* and three baselines under a certain workload, and the results are shown in Fig. 4. As evident from the results, *GSLICE* and *iGniter* are unable to mitigate the resource wastage. *Glet* opts for a fixed partition count of 2 per GPU to mitigate resource fragments, sacrificing flexibility in resource allocation and potential parallelism. The limited fragments are attributed to rounding down the allocation of actual SMs based on the percentage set by MPS.

Differing from these algorithms, we introduce *InSS*, an efficient scheduling orchestrator, that makes system decisions in a short time while avoiding resource fragments.

3 *InSS*: ARCHITECTURE & DESIGN

Driven by the observations in Sec. 2, we introduce the design of *InSS*, an intelligent scheduling orchestrator on GPU clusters that exploits *GPU partitioning* and *batching* to improve throughput while guaranteeing the SLO requirement of inference tasks.

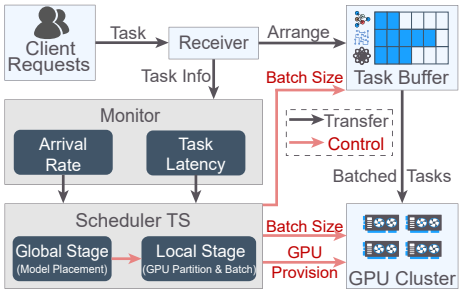

Fig. 5. Overall architecture of *InSS*.

3.1 Architecture and Workflow

Fig. 5 visually depicts the overall architecture of *InSS*, which mainly consists of four components. 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. Upon acceptance, the system places these tasks in the *task buffer*, where incoming tasks are queued until the completion of the preceding batch and the assembly of the current batch.

ii) *Monitor*. The *monitor* collects empirical data and performs predictions. It observes the task arrival process to forecast the workload arrival rate. A latency prediction model is devised to capture the interplay among system configuration, workload, and inference latency (employed for the scheduler training phase, unnecessary during online scheduling). Latency prediction includes two time metrics: **GPU execution latency** and **task latency**[3].

iii) *Scheduler*. The *Scheduler TS* utilizes a two-stage RL-based approach to determine the optimal configuration for the system (including model deployment, resource allocation, and batch size). 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*. Following the *scheduler*'s decisions, the *task buffer* organizes tasks into batches and transfers the data to the *GPU cluster* for processing once a batch is assembled. The *GPU cluster* deploys models, reallocates GPU resources based on the configuration determined by the *Scheduler*, and conducts inference computation upon receiving the data.

3.2 Formulation of Inference Services

System Overview. We consider a system for DNN inference services on a GPU cluster equipped with k GPUs. Each GPU can concurrently process the workloads of multiple DNN models by leveraging the MPS mechanism to spatially 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. 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. Therefore, denote $m_i(b_i)$ as the the memory footprint of the DNN model i with a batch size of b_i. Each model workload has its own SLO, and let SLO_i indicate 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.

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.

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

subject to:

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

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

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

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

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

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) 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 general assignment problem, which is known to be NP-hard [44]. 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.

3.3 Learning Optimal Decision with RL

Problem Transformation. To address the problem (1), we reformulate 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 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. In our scenarios, the *scheduler* is considered as an agent. The MDP of problem (1) can be designed as follows.

i) *State*. At each decision step t, the agent observes the state from the environment, which includes the information of DNN models $\mathcal{W}_t$ and GPUs $\mathcal{M}_t$. For DNN models, the state includes workload parameters like arriving rate of λ_i and queued workload G_i, *i.e.*, $\mathcal{W}_t = \{\lambda_i, G_i\}_{i \in \mathcal{I}}$. These

3. The definition is introduced in Sec. 1

variables reflect the current state of the DNN models' workload and provide the agent with insight into GPU resource demands. GPU state k comprises the previous action's used memory for DNN model placement $u_k = \sum_{\forall i \in \mathcal{I}} y_{i,k} m_i(b_i)$, *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, preventing model deployment failures.

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, resource allocations, and batch size, *i.e.*, $\mathcal{A}_t = \{y_{i,k}, \rho_{i,k}, b_i\}_{i \in \mathcal{I}, k \in \mathcal{K}}$.

iii) *Reward*. After taking the action a_t, the agent receives a reward r_t from the environment to evaluate the action's quality. Aligned with the objective of problem (1), the reward function is designed to encourage actions leading to higher throughput. Furthermore, considering the constraints (1b) and (1c), a reward function is developed to incentivize the agent to choose actions that satisfy the constraints. Let $v_{i,t}$ denote the percentage of inference tasks that violate SLO requirements at decision step t. For model comparison and ease of learning and optimization, the root mean square normalization method is used to standardize model throughput. Denote $\hat{h}_{i,k}$ as the normalized throughput of DNN model i at GPU k. We formulate our reward function as follows,

$$r_t = \sum_{i \in \mathcal{I}} r_{t,i}, \text{ where } r_{i,t} = \sum_{\forall k \in \mathcal{K}} y_{i,k} \hat{h}_{i,k} - \omega v_{i,t}, \tag{2}$$

where $r_{t,i}$ indicates the reward of DNN model i, and ω is the penalty factor. In this way, the actor is incentivized to select actions that result in higher throughput while ensuring the constraints are satisfied. Because violating constraint (1b) leads to model deployment failure, given the stochastic nature of RL, which cannot guarantee constraint satisfaction, it is necessary to check memory usage and make adjustments to the action if a violation occurs.

RL Based Solution. Accurately modeling the state of the environment in GPU clusters is challenging due to limited knowledge of transition probabilities. Moreover, dealing with high-dimensional continuous state spaces and high-dimensional action space poses a challenge in achieving tractable convergence performance [51]. Consequently, conventional dynamic programming solutions are inadequate for solving the MDP problem. To address these challenges, RL has emerged as a popular approach [68]. In this paper, we employ RL to solve this MDP problem.

3.4 Training and Scalability Challenge

Continuous-Discrete Hybrid Action Space. MPS resource allocation operates at the granularity of SMs, *e.g.*, an NVIDIA A100 GPU has 108 SMs that can be partitioned into 54 portions. Meanwhile, the batch size provides integer options ranging from 1 to 32. Treating all variables as discrete results in large action space, imposing exponential complexity and computational burden on traditional RL methods for optimal action search. Transforming the discrete action space into a continuous one allows for enhanced utilization of modern optimization algorithms, facilitating a more effective search for optimal strategies and thereby improving algorithm convergence speed and performance [68]. Consequently, an effective resolution is to consider resource allocation and batch size as continuous variables, converting the action space into a hybrid one. Although algorithms like P-DQN [54] and H-PPO [16] have been developed to tackle hybrid actions, they are constrained to a specific number of actions. This limitation poses challenges in scaling the action space, and these algorithms may demonstrate suboptimal performance, making them less suitable for this scenario.

Scalability Challenge. With the increase in the scale of GPU clusters, scalability has become a crucial concern. The state and action spaces exhibit exponential growth, requiring substantial training data to explore and determine the optimal system configuration. This exponential growth significantly prolongs the training time and may lead to the agent easily getting stuck at a local optimum [50], [51].

Given these challenges, we present a two-stage intelligent scheduler that includes a global-local partitioning of the action space. This innovative approach aims to improve training efficiency and enhance the scalability of the system.

4 LATENCY ANALYTICAL MODEL

Training RL networks involves frequent agent-environment interaction, which is a highly time-consuming and resource-intensive process. In practical systems, deploying a model and monitoring its inference service quality requires at least several tens of seconds and continuous occupancy of GPU resources. Since RL training typically involves tens of thousands of iterations to complete [35], [62], direct interaction of the scheduler with a real cluster is too slow to implement.

Similar to previous studies [36], [50], *InSS* designs a latency analytical model as a simulator to conduct learning through simulated experiments without model deployment and providing inference services in real clusters. The interference-aware latency analytical model predicts *GPU execution latency* and *task latency*, which accounts for heterogeneity in both DNN models and GPUs, as well as the potential interference caused by co-located DNN models.

4.1 GPU Execution latency prediction

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

Data Interaction Latency. The transmission of input data and computed results between CPU and GPU is conventionally facilitated through the PCIe bus. Therefore, the estimation of latency for data transfer can be predicated based on both the data volume and the accessible PCIe bandwidth. The data upload latency and the result feedback latency of a batch with DNN model i at GPU k can be obtained by $t^d_{i,k} = \alpha^d_k b_i + \beta^d_k$, $t^r_{i,k} = \alpha^r_k b_i + \beta^r_k$, where $\{\alpha^d_k, \beta^d_k, \alpha^r_k, \beta^r_k\}$ are coefficients associated with available PCIe bandwidth.

GPU Computing Latency. In this phase, we primarily focus on the effects of shared L2 Cache and DRAM in co-located DNN models under the MPS mechanism. System

statistics of L2 Cache and DRAM utilization are employed as indicators of the demand for L2 Cache and DRAM by model workloads. Higher utilization implies heightened competition for a fixed amount of resources on the GPU, thereby resulting in longer GPU running time. GPU runtime latency can be estimated by: $t^r_{i,k} = o^r_{i,k}(1 + \alpha^{cache}_{i,k} \sum_{\forall i \in \mathcal{I} \setminus i} l_{i,k} y_{i,k} + \alpha^{dram}_{i,k} \sum_{\forall i \in \mathcal{I} \setminus i} e_{i,k} y_{i,k})$, where $o^r_{i,k}$ denotes the GPU running time for a batch of DNN model i executing *alone* at GPU k, $l_{i,k}$ and $e_{i,k}$ indicate the L2 Cache and DRAM utilization of DNN model i executing alone at GPU k, $\alpha^{cache}_{i,k}$ and $\alpha^{dram}_{i,k}$ are the coefficients reflecting the interference impact on model i's GPU running time.

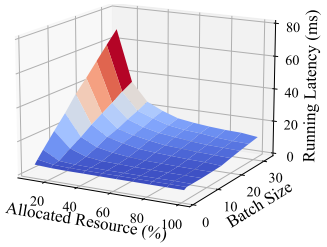

Fig. 6. GPU running latency under different configurations.

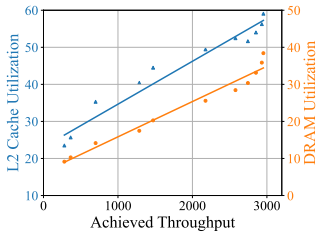

Fig. 7. L2 Cache and DRAM utilization under different throughput.

Obviously, the GPU execution time of models exhibits a direct relationship to both the batch size and the allocated resources. Fig. 6 depicts the GPU running time of an independently executed ResNet50 across diverse configurations. The data reveals an inverse correlation between GPU running latency and the quantity of allocated resources, coupled with a notable escalation in latency as the batch size increases. However, the intricate and nonlinear nature of the relationship between these variables precludes the derivation of a feasible analytical model. As a result, we utilized locally weighted regression (LWR) [10] to effectively capture the interdependence among allocated resources, batch size, and GPU running time. Then, the GPU running time can be obtained by $o^r_{i,k} = LWR(b_i, \rho_{i,k} \cdot C_k)$.

Heightened GPU computing capability tends to result in more pronounced resource competition. The L2 Cache and DRAM utilization under varying achieved throughputs are illustrated in Fig. 7. The dots in the figure indicate the measured values of utilization. According to the data, L2 Cache and DRAM utilization can be predicted as: $l_{i,k} = \alpha^c_{i,k}(b_i/o^r_{i,k}) + \beta^c_{i,k}$ and $e_{i,k} = \alpha^u_{i,k}(b_i/o^r_{i,k}) + \beta^u_{i,k}$, where $\{\alpha^c_{i,k}, \beta^c_{i,k}, \alpha^u_{i,k}, \beta^u_{i,k}\}$ are four coefficients for model i at GPU k. The curves fitted by the formulas are also depicted in Fig. 7, demonstrating a notably excellent fit.

Prediction Overhead. In the above prediction model, it is essential to measure parameters and coefficients before the system's operation. To derive the coefficients for the prediction model, it is essential to individually execute each model on every type of GPU with 16 distinct configurations (4 options of resource allocation and 4 options of batch size). This involves recording three steps of latency, as well as L2 Cache and DRAM utilization. Moreover, the coefficients $\{\alpha^{cache}_{i,k}, \alpha^{dram}_{i,k}\}$ require data collected by launching multiple DNN models concurrently. It is also essential to conduct three sets of multiple parallel runs and record latency data. Experimental data is obtained by utilizing Nsight Systems [43] and Nsight Compute [42]. Ultimately, the least squares method [38] is employed to fit coefficients $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, and independently executed GPU running time data is utilized for LWR. Typically, obtaining the coefficients for the prediction model is a process that requires only a few minutes.

4.2 Task Latency Prediction

Arriving Rate. Clients submit continuous inference tasks 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 notes that the task arrival rate for DNN inference services conforms to a Poisson distribution [67]. Consequently, the arrival rate of each model workload λ_i can be obtained by fitting a Poisson distribution. The *monitor* predicts the future arrival rate of the model workload based on the historical arrival data.

Task Latency. The latency associated with a task spans from its submission to the completion of result delivery. Since the data size of the result is relatively small, the transmission latency of the result delivery is ignored [32]. Then, the queuing latency within the task buffer emerges as a crucial consideration. Let $g_i(t)$ denote the queued workload of model i at t (before the task's arrival). The queued workload of DNN model i after task j is $w_{i,j} = g_i(a_{i,j}) + d_{i,j}$. Tasks are queued in the task buffer under two conditions: i) When the queued workload is sufficient (*i.e.*, $w_{i,j} \geq b_i$), the current task must await the processing of earlier arrived tasks. ii) When 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 subsequent tasks until the queued workload equals the batch size. The task latency of task j with DNN model i is then given by:

$$T_{i,j} = \begin{cases} \sum_{\forall k \in \mathcal{K}} y_{i,k} exec_{i,k} + \lceil w_{i,j}/b_i \rceil \sum_{\forall k \in \mathcal{K}} y_{i,k} t^{inf}_{i,k}, & w_{i,j} \geq b_i, \\ max(\sum_{\forall k \in \mathcal{K}} y_{i,k} exec_{i,k}, wait_{i,j}) + \sum_{\forall k \in \mathcal{K}} y_{i,k} t^{inf}_{i,k}, & w_{i,j} < b_i, \end{cases} \tag{3}$$

where $exec_{i,k}$ denotes the remaining execution time of the *currently* processed DNN model i workload on GPU k, and $wait_{i,j}$ indicates the queuing latency required for sufficient workloads to be completed for batch assembly and is calculated by: $wait_{i,j} = \frac{b_i - W_{i,j}}{\lambda_i}$.

5 TS: TWO-STAGE INTELLIGENT SCHEDULING

In this section, we introduce *TS*, an intelligent scheduler that enables efficient and scalable learning in extensive clusters. Considering the nature of the action space, we decompose the decision-making process into two stages (global stage and local stage) and partition the hybrid action space into discrete and continuous components, enhancing the efficiency of decision-making by the RL agents.

5.1 Global Stage: Model Placement

Redesign. Differing from Sec. 3.3, considering the scalability issue, we train an actor network for each model to make deployment decisions. Hence, the *state* $\mathcal{S}'_{i,t}$ is expanded to $\{\lambda_i, g_i, p_i, W_i, u_k, L_{i,k}, E_{i,k}\}_{k \in \mathcal{K}}$. W_i denotes the workload of models without deployment decisions, *i.e.*, $W_i = \sum_{i'=i}^{I} \hat{\lambda}_{i'} + \hat{g}_{i'}$ where $\hat{\lambda}_{i'}$ and $\hat{g}$ are the normalized values using the same

normalization method as the throughput. $L_{i,k}/E_{i,k}$ indicates the L2 Cache/DRAM utilization for models that have been decided to deploy on GPU k, *i.e.*, $L_{i,k} = \sum_{i'=0}^{i} y_{i,k} l_{i,k}$. These two metrics indicate the current resource contention on GPU k, enabling the agent to make more informed decisions. The *action* is $\mathcal{A}'_{i,t} = \{y_{i,k}\}_{k\in\mathcal{K}}$, which are *discrete*. The *reward* is $r'_{i,t} = \sum_{\forall k\in\mathcal{K}} y_{i,k}\hat{h}_{i,k} - \omega v_{i,t}$.

Solution. It is necessary to choose an appropriate RL algorithm for the *discrete* action space. *TS* utilizes a SACD based approach which comprises four key components that work together to improve performance [9]. *i)* An actor-critic architecture with an actor network $\pi_{\phi'}$, Q-networks $Q_{\theta'}$, and target Q-networks $Q_{\hat{\theta}'}$. The clipped double Q-networks technique is applied to both Q-networks and target Q-networks to avoid overestimation of Q-values [5]. *ii)* An off-policy with experience replay technique to expedite the efficiency of convergence. *iii)* Discrete action space suited for this situation. The actor network's output layer employs discretization, while Q-networks output Q-values for each possible action. *iv)* An entropy regularization term to ensure exploration and stability. Agent aims to find a policy π^* that maximizes both cumulative reward and entropy objective: $\pi^*_{\phi'} = \arg\max_\pi \sum_{t\in\mathcal{T}} \mathbb{E}_{(s'_t,a'_t)\sim\zeta_\pi}[\gamma^t(r'_t + \tau\mathcal{H}(\pi(.|s'_t)))]$, 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, to ensure that constraint (1b) is not violated, before executing the actor network to acquire actions, *TS* checks the GPUs' memory resource usage to obtain the "safe action space" (GPUs capable in deploying the present model).

Scalability Discussion. With the escalating number of GPUs within the cluster, this RL method still faces the challenge of exponential growth in state and action spaces. Nevertheless, *TS* can address this challenge adeptly through two methods. Firstly, by grouping GPUs, *TS* can extend the present RL method into hierarchical RL to solve the problem of model deployment [31], [50]. For instance, in a cluster with 64 GPUs, they can be organized into sets of 4, forming a three-layer RL structure where decisions cascade from the top layer to the bottom layer for specific deployments. Secondly, by grouping both GPUs and models, *TS* can systematically decompose the overarching large-scale problem into more manageable subproblems, facilitating the continued application of the present RL. For example, GPUs can be organized into sets of 4, with models evenly distributed, and each group makes independent decisions.

5.2 Local Stage: Resource and Batch Decision

Redesign. After determining the deployment strategy, the scheduling problems on each GPU become *independent*. Combining the deployment decisions with the original state, the *state* $s''_{k,t}$ at this stage is $s''_{k,t} = \{i, \lambda_i, g_i, u_k\}_{i\in\mathcal{I}'}$, where $\mathcal{I}'$ denotes the set of models deployed on GPU k, *i.e.*, $y_{i,k} = 1, \forall i \in \mathcal{I}'$. The action is $a''_{k,t} = \{\rho_{i,k}, b_i\}_{i\in\mathcal{I}'}$, which are considered as *continuous* variables. The *reward* is $r''_{k,t} = \sum_{\forall i\in\mathcal{I}'} y_{i,k}\hat{h}_{i,k} - \omega v_{i,t}$.

Solution. *TS* utilizes the TD3 [17] which comprises three key components to efficiently explore and optimize the *continuous* action space. *i)* An actor-critic architecture with an actor network $\pi_{\phi''}$, a target actor network $\pi_{\hat{\phi}''}$, Q-networks $Q_{\theta''}$, and target Q-networks $Q_{\hat{\theta}''}$. The clipped double networks technique is also applied to Q-networks and target Q-networks. *ii)* An off-policy with experience replay technique to enhance the convergence efficiency. *iii)* A truncated normally distributed noise is added to the output of actor network to balance bias and variance, mitigating overfitting, *i.e.*, action $a''_{k,t} = \pi_{\phi''}(s''_t)+n_1, n_1 \sim \mathcal{N}(0,\sigma_1^2)$ and target action $\hat{a}''_{k,t+1} = \pi_{\hat{\phi}''}(s''_{t+1}) + n_2, n_2 \sim clip(\mathcal{N}(0,\sigma_2^2), -c, c)$. Moreover, considering resource capacity constraints and the challenge of resource fragments, *TS* applies a softmax function to the output of resource allocation, thereby ensuring that the cumulative resource allocations sum up to 100%.

Algorithm 1 Intelligent Scheduler (*TS*)

1: Initialize SACD network parameters ϕ', θ'_1, θ'_2 and $\hat{\theta}'_1 \leftarrow \theta'_1$, $\hat{\theta}'_2 \leftarrow \theta'_2$, an empty replay buffer $\mathcal{D}'$, and other hyperparameters γ', τ, κ' ℓ'_Q, ℓ'_π, ℓ'_τ, $\bar{\mathcal{H}}$;
2: Initialize TD3 network parameters ϕ'', θ''_1, θ''_2 and $\hat{\phi}'' \leftarrow \phi''$, $\hat{\theta}''_1 \leftarrow \theta''_1$, $\hat{\theta}''_2 \leftarrow \theta''_2$, an empty replay buffer $\mathcal{D}''$, and other hyperparameters γ'', κ'' ℓ''_Q, ℓ''_π, σ_1, σ_2, c;
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: Transmit s_t to $s'_{1,t}$ and check the action space;
7: **for** $i = 1$ to I **do**
8: Select action with SACD policy $a'_{i,t} \sim \pi_{\phi'}(s'_{i,t})$;
9: Calculate new state $s'_{i+1,t}$ and check action space;
10: **end for**
11: Integrate s_t and a'_t to $\{s''_{k,t}\}_{\forall k\in\mathcal{K}}$;
12: **for** $k = 1$ to K **do**
13: Select action with TD3 policy $a''_{k,t} = \pi_{\phi''}(s''_t)+n_1$;
14: **end for**
15: Apply the actions $\{a'_t, a''_t\}$ and receive the reward $\mathbf{r}_t$;
16: Observe the next state s_{t+1} and store $\{s'_t, a'_t, r'_t, s'_{t+1}\}$ and $\{s''_t, a''_t, r''_t, s''_{t+1}\}$ into replay buffer $\mathcal{D}'$ and $\mathcal{D}''$, respectively;
17: Sample a mini-batch of experience $\mathcal{F}'$ from $\mathcal{D}'$;
18: Update SACD Q-networks, actor network and temperature with learning rate ℓ'_Q, ℓ'_π and ℓ'_τ;
19: Update SACD target networks: $\hat{\theta}'_v \leftarrow \kappa'\theta'_v + (1-\kappa')\hat{\theta}'_v$.
20: Sample a mini-batch of experience $\mathcal{F}''$ from $\mathcal{D}''$;
21: Update TD3 Q-networks with learning rate ℓ''_Q.;
22: **if** t mode ν **then**
23: Update TD3 policy with learning rate ℓ''_π;
24: Update TD3 target networks: $\hat{\theta}''_v \leftarrow \kappa''\theta''_v + (1-\kappa'')\hat{\theta}''_v$, $\hat{\phi}'' \leftarrow \kappa''\phi'' + (1-\kappa'')\hat{\phi}''$.
25: **end if**
26: **end for**
27: **end for**

5.3 Algorithm Design

Our proposed intelligent scheduler, *TS*, is presented in Alg. 1. The workflow of *TS* is as follows:

Initialization Phase. Firstly, lines 1-2 initialize networks' parameters, an empty reply buffer, and related hyperparameters for SACD and TD3, respectively. *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 the initial state s_t, the agent sequentially makes deployment decisions for each model $a'_{i,t} = \{y_{i,k}\}_{k\in\mathcal{K}}$ with SACD actor policy $\pi_{\phi'}$, calculate the

state $s_{i+1,t}$ for next model (lines 6-10). Then Integrating the initial state with deployment decisions to s''_t, and the agent chooses action of resource allocation and batch size $a''_{k,t} = \{\rho_{i,k}, b_i\}_{i\in\mathcal{I}'}$ for each GPU (lines 11-14). After receiving the actions $\{a'_t, a''_t\}$ from *TS*, *task buffer* and *GPU cluster* apply them to provide DNN inference service. Following a decision step, the agent receives the reward r_t, observe the next state s_{t+1}; the transition tuples $\{s'_t, a'_t, r'_t, s'_{t+1}\}$ and $\{s''_t, a''_t, r''_t, s''_{t+1}\}$ into replay buffer $\mathcal{D}'$ and $\mathcal{D}''$ (lines 15-16).

Training Phase. During the training stage, *TS* executes the subsequent procedures to update networks. For SACD, line 17 samples a mini-batch of experiences $\mathcal{F}'$ randomly from the replay buffer $\mathcal{D}'$. Then, the parameters of Q-networks, policy weight and temperature are updated using stochastic gradient descent (SGD) method [58] with learning rates ℓ'_Q, ℓ'_π and ℓ'_τ (line 18). Line 19 updates target Q-networks by employing a soft-updating method with update factor κ'. Similarly, line 20 samples experiences $\mathcal{F}''$ randomly to update networks for TD3. The parameters of Q-networks are also updated by the SGD method with learning rate ℓ''_Q (line 21). The distinction is that the actor network and target networks of TD3 are updated less frequently than the Q-networks. For every ν step, the actor network is updated parameters by using the SGD method with learning rates ℓ''_π and parameters of target networks are updated using the soft-updating method with factor κ'' (lines 22-25).

5.4 Discussion

In this paper, we focus on the fine-grained allocation of GPU resources, where partial GPU resources can efficiently handle the workload of DNN models. Here, We discuss two special scenarios for potential expansion within *InSS*.

Large Language Model. Large language models (LLMs) that employ model parallelism and serialized execution, can be divided into multiple interdependent sub-models [37]. These sub-models are activated and incorporated into the scheduling model set only after the completion of preceding execution stages. Subsequent deployment and execution of these sub-models are carried out based on the scheduling decisions derived from our algorithm *TS*.

Heavy Model Workload. In scenarios where a single GPU resource cannot meet the throughput requirements of a model's workload, deploying model replication is a viable strategy to ensure efficient inference services. Specifically, a metric can be established for each model to evaluate its workload level, derived from the model's computational demands and the task arriving rate. Based on throughput data across different allocated resources, thresholds are set for each model to determine whether model replication is necessary. Subsequent decisions of model deployment and resource allocation can be directly guided by algorithm *TS*.

5.5 Implementation

Prototype. We implemented *InSS* based on a GPU inference service software prototype [26], 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 RL, resulting in over 1k lines of code. Furthermore, the DNN models 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. If needed, we are open to releasing the source code of the *InSS* prototype.

GPU Resource Reallocation. NVIDIA's MPS facilitates specific computational resource allocation to processes. However, reallocating resources involves creating a new inference service process, which includes starting a new process, loading kernels, loading the model, and warming up the model. To minimize disruption to ongoing inference tasks, the reallocation process utilizes a shadow mechanism. This mechanism prepares a new process, activates it, and finally kills the original process, Although running the shadow and original processes concurrently incurs memory overhead, this approach ensures that the existing inference tasks remain uninterrupted.

6 EVALUATION

6.1 Experiments Setup

GPU Cluster. In our GPU cluster configuration, the inference server is equipped with 4 NVIDIA Ampere A100 GPUs. All GPUs in the cluster support post-Volta MPS capabilities. Both the *client* and the *scheduler* components are executed on the CPU, which is responsible for generating and sending inference task requests, as well as running the scheduling program of the scheduler.

Workload. Eight widely-known DNN models are selected for inference services [65], *i.e.*, AlexNet (A) [28], ResNet50 (R) [53], VGG16 (V) [46], MnasNet (M) [48], MobileNet (M1) [22], EfficientNet (E) [49], ResNet18 (R1), and VGG19 (V1). Both the training and inference stages utilize the widely used ImageNet dataset [14], comprising more than 1 million images, each with a resolution of 224x224 pixels and 3 channels. The SLO requirements for the eight selected models are set based on the GPU execution latency as follows: [10, 35, 65, 10, 15, 30, 15, 80] ms. To model the workload arrival rate for each DNN model, a Poisson random distribution is employed to sample inter-arrival time, a methodology demonstrated to be effective in approximating real-world arrival rates [67].

TABLE 2
Parameter Setting of *TS*

Parameter	Value	Parameter	Value
Number of episodes E	1600	Number of steps T	100
Replay buffer size	100000	Mini-batch size	256
Learning rate ℓ'	0.0001	Learning rate ℓ''	0.0003
Temperature initial τ	1.0	Target entropy $\hat{H}$	-$\log(1+\iota)$
Soft factor κ'	0.01	Soft factor κ''	0.005
Discount factor γ	0.99	Optimizer	Adam
Explore noise σ_1	0.1	Policy noise σ_2	0.2
Noise clip c	0.5	Policy frequent ν	2
Hidden layer act.	ReLU	Actor output act.	Softmax

Algorithm Networks. Our RL 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 256 neurons. We set the reward penalty factor to $\omega = 40$. Other parameters are listed in Table 2.

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

- *Gpulet* [6]: It divides a GPU into two pieces. It allocates GPU resources by maximizing request throughput and places DNN models on the most suitable GPU.
- *GSLICE* [15]: This strategy adjusts the allocation of GPU resources and batch sizes by traversing the average latency and throughput of the DNN models.
- *iGniter* [55]: 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 in our orchestrator. Here, we evaluate the performance of our GPU execution prediction model.

TABLE 3
Observed and predicted GPU execution latency with different batch sizes

Model / Batch size		1	4	8	16
R	obs (ms)	4.997	6.809	11.059	19.533
	pre (ms)	4.923	7.033	10.864	19.387
	error (%)	**1.484**	**3.295**	**1.763**	**0.748**
A&R*	obs (ms)	0.940	1.773	2.482	3.976
	pre (ms)	0.955	1.785	2.461	3.968
	error(%)	**1.723**	**0.705**	**0.850**	**0.186**
V&A*&R*	obs (ms)	4.366	11.180	19.343	36.688
	pre (ms)	4.349	11.176	19.602	37.427
	error (%)	**0.379**	**0.035**	**1.338**	**2.016**

Impact of Batch Size. Table 3 presents the GPU execution latency of three models with different batch sizes and co-located DNN models. In Table 2, all DNN models are deployed with an equal allocation of 20% GPU resources. Here, we use X* to indicate that DNN model X is fixedly scheduled, *i.e.,* its batch size is fixed at 8. It can be observed that *InSS* can accurately predict GPU execution latency with a low prediction error ranging from 0.748% to 3.295% for ResNet50, 0.186% to 1.723% for AlexNet, and 0.035% to 2.016% for VGG16.

TABLE 4
Observed and predicted task inference latency with different allocated resources

Models / Resource(%)		10	30	50	60
M	obs (ms)	5.674	3.633	3.675	3.623
	pre (ms)	5.673	3.665	3.642	3.638
	error (%)	**0.025**	**0.866**	**0.908**	**0.422**
R1&M*	obs (ms)	7.884	3.113	2.155	2.004
	pre (ms)	7.927	3.185	2.201	2.104
	error (%)	**0.555**	**2.323**	**2.112**	**4.971**
V1&R1*&M*	obs (ms)	43.645	15.213	9.909	9.161
	pre (ms)	44.878	15.716	10.202	8.867
	error (%)	**2.824**	**3.307**	**2.953**	**3.214**
E*&M1	obs (ms)	8.616	8.733	8.757	8.779
	pre (ms)	8.747	8.877	8.879	8.879
	error (%)	**1.520**	**1.639**	**1.393**	**1.145**

Impact of Allocated Resources. The prediction results of four models are presented in Table 4. The batch size of all DNN models is fixed at 8. Let X* denote DNN model X with a fixed resource allocation of 20% GPU. The results show that with the change of allocated resources, the prediction error of InSS consistently remains below 5%. 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 4, which shows an error range of 1.372% to 3.818%. This confirms that an increase in allocated resources corresponds to a higher system throughput, resulting in increased interference with the co-located model. *InSS* exhibits the ability to predict such variations, as detailed in Section 4. It is worth noting that the prediction results of other models are similar, we do not present redundant data for all models.

6.3 Evaluation Results

Convergence of TS. The convergence performance of the scheduler, *TS*, is presented in Fig. 8. The plot shows that both the reward value of SACD and TD3 gradually increases with an increase in the number of training episodes until it reaches a relatively stable value. It validates that SACD and TD3 converge after parameter iterations for 400 and 800 episodes. To provide further analysis on the convergence effect of *TS*, we plot SLO violation and throughput for each iteration in Fig. 9. 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.

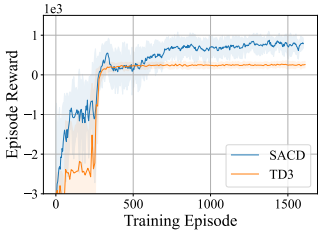

Fig. 8. Convergence performance of *TS*.

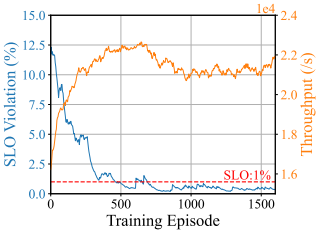

Fig. 9. Convergence performance details of *TS*.

System Throughput. After well offline training, we evaluate the performance of *InSS* on a system equipped with 4 GPUs providing inference services for 10 DNN models (two models are used twice). *Glet* divides one GPU into two partitions and provides service for only 8 models. As shown in Fig. 10, 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. This is because InSS employs RL to optimize system decisions rather than simple heuristic algorithms, thoroughly considering the intricate relationships among models, GPUs, and resource allocation. Specifically, in comparison with *Gpulet*, *GSLICE*, and *iGniter*, *InSS* attained improvements of 52.8%, 26.4%, and 86.1%, respectively. The difference in throughput performance between *InSS* and *InSS-ni* is not significant. However, the impact of interference prediction is primarily observed in SLO violations, which are described later. To further present the performance of *InSS*, we plot the system throughput overtime in Fig. 11, which shows that *InSS* adjusts decisions based on the dynamic tasks' arrival and maintains high throughput. Compared to baselines, *InSS* achieves a more stable throughput, exhibiting smaller fluctuations to environment changes. This stability contributes to the robustness and reliability of the system.

Inference Service Quality. To examine the quality of inference service provided by the system, we present the SLO

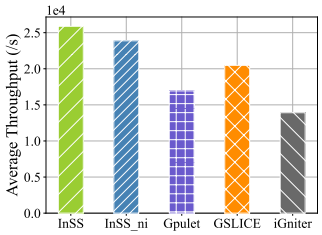

Fig. 10. Throughput of different algorithms.

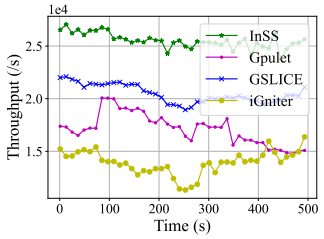

Fig. 11. Throughput over time of different algorithms.

violations of different algorithms in Fig. 12. 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. When *InSS* does not consider interference, the SLO violation increases by 0.80%. Although this impact may seem small, we argue that considering interference is necessary since a scheduler must ensure SLO at all times. *InSS-ni* cannot guarantee a service quality of 99% for inference tasks. This further emphasizes the importance of interference consideration. 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 RL which has higher fault tolerance. The SLO violation of the *Gpulet* and *GSLICE* is relatively high, because they do not consider interference, which may cause task latency beyond the SLO.

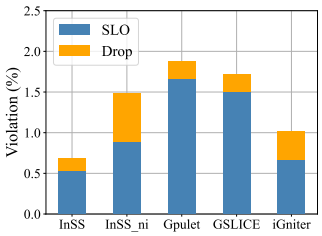

Fig. 12. SLO violation of different algorithms.

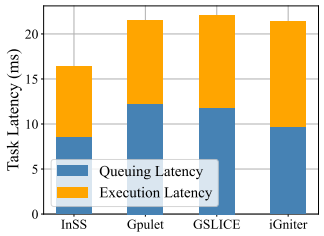

Fig. 13. Task latency of ResNet50.

Furthermore, Fig. 13 and Fig. 14 illustrate the average task latency of ResNet50 and VGG under different algorithms, respectively. It can be observed that *InSS* achieves the minimum task latency compared to the three baselines. Due to better utilization of GPU resources, *InSS* ensures both high system throughput and improved task response.

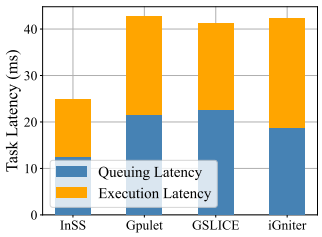

Fig. 14. Task latency of VGG16.

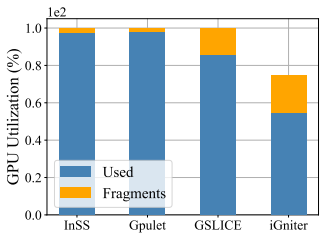

Fig. 15. GPU utilization of different algorithms.

GPU Utilization. Resource fragments are significant consideration for the system, and we present GPU utilization and Fragment in Fig. 15. It can be observed that *InSS*'s resource usage aligns with *Glet*. The limited fragments in *InSS* are attributed to rounding down the allocation of actual SMs based on the percentage set by MPS. Glet achieves this by dividing the GPU into only two partitions, limiting the allocation to only two models. In contrast, *InSS* introduces a softmax function at the output layer of the TD3 actor network to ensure optimal resource utilization while not imposing restrictions on the number of resource partitions for a single GPU. *iGniter* is designed with the primary objective of minimizing GPU usage, consequently employing only three GPUs. However, in its pursuit to minimize resource consumption, it allocates GPU resources in a fragmented manner. This leads to almost every GPU having small, unusable portions of resources that cannot satisfy the resource requirements of DNN models, exacerbating the issue of resource fragment. As illustrated in Fig. 15, results of *iGniter* exhibit the highest level of resource fragmentation compared to other algorithms.

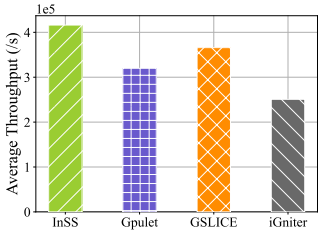

Fig. 16. Throughput of different algorithms.

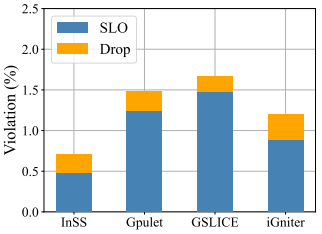

Fig. 17. SLO violations of different algorithms.

Scalability. We next evaluate the scalability of *InSS* in a large cluster with 64 GPUs. The system's average throughput and total SLO violation data are presented in Fig. 16 and Fig. 17, respectively. From the figures, we can observe that *InSS* outperforms the baselines in terms of system throughput, demonstrating an improvement of up to 66.0% compared to the baselines. Moreover, the system running with *InSS* exhibits superior service quality. The SLO violation of *InSS* is minimal and consistently below 1%. Fig. 18 presents the GPU usage of four algorithms. Similarly, *InSS* exhibits only a small amount of resource fragments, while the baselines experience up to 800% GPU fragments. These results demonstrate the effective scalability of *InSS* to large-scale clusters.

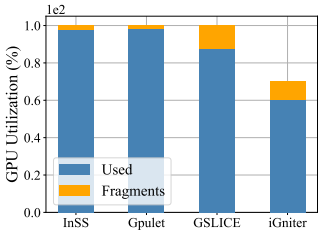

Fig. 18. GPU utilization of different algorithms.

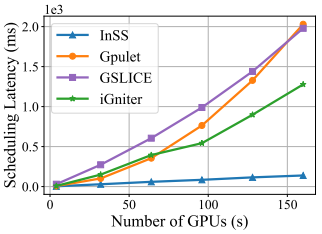

Fig. 19. Scheduling overhead of different algorithms.

Scheduling Overhead. We measure the running overhead of all algorithms across different numbers of GPUs. The results are depicted in Fig. 19. It can be observed that the overhead of *InSS* increases with the number of GPUs in the cluster but consistently remains at a relatively small value (less than 0.2s). In contrast, the running time of the baselines is significantly higher (up to 2.0s), which is unfavorable for online scheduling of inference services. This is because *InSS* utilizes an RL agent that has been trained to learn the relationship between different decision configurations and SLO violations under varying workload inputs. Consequently,

the RL agent can make decisions that inherently prevent SLO violations, thereby eliminating the need for repeated SLO violation checks and adjustments.

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], [7], [18], [24]. *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* [39] combines dynamic batching and DVFS techniques to control power consumption and improve throughput in DNN inference on GPUs. *DVABatch* [12] incorporates dynamic batching along with a multi-entry multi-exit scheme, achieving significant improvements in latency and throughput. Considering varied user demands for accuracy and latency, Liu *et al.* [33] explore optimal resource allocation strategies with batching and early exiting techniques to maximize throughput. However, due to the SLO requirements of tasks, the system is unable to wait for a large batch accumulation, thereby limiting the potential improvements in system throughput and efficiency in resource utilization.

Inference with GPU Temporal Sharing. Temporal scheduling is common in GPU inference [34], [52], [59]. *Nexus* [45] enhances GPU cluster performance and utilization with batching-aware scheduling and early drop mechanisms, using a fixed batch size for each epoch. *Clockwork* [20] uses predictive execution times to order user requests and proposes fine-grained request-level scheduling to meet their SLO. *PipeSwitch* [3] enables efficient GPU time-sharing through pipelined context switching and model-aware grouping, achieving near-maximal GPU utilization while meeting stringent SLOs. Temporal sharing enhances GPU utilization but the system still executes one job at a time. It also introduces overhead costs due to model switching and increased 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 [8], [13], [25], [27], [29]. *GSLICE* [15] focuses on resource allocation for GPUs and does not consider performance interference between co-located models. *Gpulet* [6] 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. *iGniter* [55] is designed to minimize GPU usage by calculating batch size and minimum resource allocation using two formulas and deploying models based on interference prediction. *REEF* [21] enables microsecond-scale kernel preemption and concurrent execution on GPUs, to improve throughput and reduce latency. *Clover* [30] aims to reduce the carbon footprint of inference services through mixed-quality models and GPU partitioning. However, task latency is significantly influenced by factors such as scheduling overhead, performance interference from co-located tasks, and queuing latency. These crucial aspects have not been comprehensively addressed in the prior works. 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 CONCLUSIONS

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 task latency based on interference and queuing latency. *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 maximizing the system throughput. Extensive prototype experiments verify the efficiency of *InSS*. *InSS* can increase the throughput by up to 86%, while satisfying the SLOs.

REFERENCES

[1] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[2] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *Computer*, 50(10):58–67, 2017.

[3] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514, 2020.

[4] Michael Braun, Anja Mainz, Ronee Chadowitz, Bastian Pfleging, and Florian Alt. At your service: Designing voice assistant personalities to improve automotive user interfaces. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–11, 2019.

[5] Shaotao Chen, Xihe Qiu, Xiaoyu Tan, Zhijun Fang, and Yaochu Jin. A model-based hybrid soft actor-critic deep reinforcement learning algorithm for optimal ventilator settings. *Information Sciences*, 611:47–64, 2022.

[6] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing. In *Proceedings of 2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, 2022.

[7] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. Lazy batching: An sla-aware batching system for cloud machine learning inference. In *Proceedings of 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 493–506, 2021.

[8] Marcus Chow, Ali Jahanshahi, and Daniel Wong. Krisp: Enabling kernel-wise right-sizing for spatial partitioned gpu inference servers. In *Proceedings of 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 624–637, 2023.

[9] Petros Christodoulou. Soft actor-critic for discrete action settings. 2019.

[10] William S. Cleveland and Susan J. Devlin. Locally weighted regression: An approach to regression analysis by local fitting. *Journal of the American Statistical Association*, 83(403):596–610, 1988.

[11] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Mingyi Guo. Ebird: Elastic batch for improving responsiveness and throughput of deep learning services. In *Proceedings of 2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 497–505, 2019.

[12] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. Dvabatch: Diversity-aware multi-entry multi-exit batching for efficient processing of dnn services on gpus. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 183–198, 2022.

[13] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, 2021.

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of 2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[15] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. Gslice: Controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 492–506, 2020.

[16] Zhou Fan, Rui Su, Weinan Zhang, and Yong Yu. Hybrid actor-critic reinforcement learning in parameterized action space. 2019.

[17] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1587–1596. PMLR, 10–15 Jul 2018.

[18] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[19] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. Fast-gshare: Enabling efficient spatio-temporal gpu sharing in serverless computing for deep learning inference. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 635–644, 2023.

[20] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[21] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent gpu-accelerated dnn inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.

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

[23] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. Scrooge: A cost-effective deep learning inference system. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 624–638, 2021.

[24] Yoshiaki Inoue. Queueing analysis of gpu-based inference servers with dynamic batching: A closed-form characterization. *Performance Evaluation*, 147:102183, 2021.

[25] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic space-time scheduling for gpu inference. 2018.

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

[27] Yunseong Kim, Yujeong Choi, and Minsoo Rhu. Paris and elsa: An elastic scheduling algorithm for reconfigurable multi-gpu inference servers. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 607–612, 2022.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Proceedings of Advances in Neural Information Processing Systems*, volume 25, 2012.

[29] Baolin Li, Viiay Gadepally, Siddharth Samsi, and Devesh Tiwari. Characterizing multi-instance gpu for machine learning workloads. In *Proceedings of 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 724–731, 2022.

[30] Baolin Li, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Clover: Toward sustainable ai with carbon-aware machine learning inference service. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, 2023.

[31] Suyi Li, Luping Wang, Wei Wang, Yinghao Yu, and Bo Li. George: Learning to place long-lived containers in large clusters with operation constraints. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 258–272, 2021.

[32] Ming Liu, Tao Li, Neo Jia, Andy Currid, and Vladimir Troy. Understanding the virtualization "tax" of scale-out pass-through gpus in gaas clouds: An empirical study. In *Proceedings of 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 259–270, 2015.

[33] Zhiyan Liu, Qiao Lan, and Kaibin Huang. Resource allocation for multiuser edge inference with batching and early exiting. *IEEE Journal on Selected Areas in Communications*, 41(4):1186–1200, 2023.

[34] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *Proceedings of 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[35] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 50–56, 2016.

[36] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270–288, 2019.

[37] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 932–949, 2024.

[38] Steven J Miller. The method of least squares. *Mathematics Department Brown University*, 8:1–7, 2006.

[39] Seyed Morteza Nabavinejad, Sherief Reda, and Masoumeh Ebrahimi. Coordinated batching and dvfs for dnn inference on gpu accelerators. *IEEE Transactions on Parallel and Distributed Systems*, 33(10):2496–2508, 2022.

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

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

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

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

[44] Xun Shao, Go Hasegawa, Mianxiong Dong, Zhi Liu, Hiroshi Masui, and Yusheng Ji. An online orchestration mechanism for general-purpose edge computing. *IEEE Transactions on Services Computing*, 16(2):927–940, 2022.

[45] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, 2019.

[46] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 2015.

[47] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. Serving dnn models with multi-instance gpus: A case of the reconfigurable machine scheduling problem. 2021.

[48] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[49] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114, 2019.

[50] Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. Metis: Learning to schedule long-running applications in shared container clusters at scale. In *Proceedings of SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17, 2020.

[51] Yuhuai Wu, Elman Mansimov, Roger B Grosse, Shun Liao, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vish-

wanathan, and R. Garnett, editors, *Proceedings of Advances in Neural Information Processing Systems*, volume 30, 2017.

[52] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, 2018.

[53] Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[54] Jiechao Xiong, Qing Wang, Zhuoran Yang, Peng Sun, Lei Han, Yang Zheng, Haobo Fu, Tong Zhang, Ji Liu, and Han Liu. Parametrized deep q-networks learning: Reinforcement learning with discrete-continuous hybrid action space. 2018.

[55] Fei Xu, Jianian Xu, Jiabin Chen, Li Chen, Ruitao Shang, Zhi Zhou, and Fangming Liu. igniter: Interference-aware gpu resource provisioning for predictable dnn inference in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):812–827, 2023.

[56] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: A native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 768–781, 2022.

[57] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Chenchen Liu, and Xiang Chen. A survey of multi-tenant deep learning inference on gpu. 2022.

[58] Hao Yu, Rong Jin, and Sen Yang. On the linear speedup analysis of communication efficient momentum SGD for distributed non-convex optimization. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7184–7193, 2019.

[59] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep learning applications. In *Proceedings of Machine Learning and Systems*, volume 2, pages 98–111, 2020.

[60] Deze Zeng, Andong Zhu, Lin Gu, Peng Li, Quan Chen, and Minyi Guo. Enabling efficient spatio-temporal gpu sharing for network function virtualization. *IEEE Transactions on Computers*, 72(10):2963–2977, 2023.

[61] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019.

[62] Chi Zhang, Yuan Meng, and Viktor Prasanna. A framework for mapping drl algorithms with prioritized replay buffer onto heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 34(6):1816–1829, 2023.

[63] Wei Zhang, Quan Chen, Ningxin Zheng, Weihao Cui, Kaihua Fu, and Minyi Guo. Toward qos-awareness and improved utilization of spatial multitasking gpus. *IEEE Transactions on Computers*, 2021.

[64] Wei Zhang, Quan Chen, Ningxin Zheng, Weihao Cui, Kaihua Fu, and Minyi Guo. Toward qos-awareness and improved utilization of spatial multitasking gpus. *IEEE Transactions on Computers*, 71(4):866–879, 2022.

[65] Xiaoyu Zhang, Degang Wang, Kaoru Ota, Mianxiong Dong, and Hongxing Li. Exponential stability of mixed time-delay neural networks based on switching approaches. *IEEE Transactions on Cybernetics*, 52(2):1125–1137, 2020.

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

[67] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. 44(3), 2016.

[68] Huan Zhou, Zhenning Wang, Hantong Zheng, Shibo He, and Mianxiong Dong. Cost minimization-oriented computation offloading and service caching in mobile cloud-edge computing: An a3c-based approach. *IEEE Transactions on Network Science and Engineering*, 10(3):1326–1338, 2023.

Ziyi Han received the BE degree from the School of Cyber Science and Engineering, Wuhan University, China, in 2021. She is currently working toward the master's degree from the School of Cyber Science and Engineering, Wuhan University, China. Her research interests include edge computing, online learning and network optimization.

Ruiting Zhou (Member, IEEE) is a Professor in the School of Computer Science Engineering at Southeast University. She received her Ph.D. degree in 2018 from the Department of Computer Science, University of Calgary, Canada. Her research interests include cloud computing, machine learning and mobile network optimization. She has published research papers in top-tier computer science conferences and journals, including IEEE INFOCOM, ACM MobiHoc, IEEE/ACM TON, IEEE JSAC, IEEE TMC. She serves as the TPC chair for INFOCOM workshop-ICCN 2019-2024. She also serves as a reviewer for international conferences and journals such as IEEE INFOCOM, IEEE ICDCS, IEEE/ACM IWQoS, IEEE SECON, IEEE JSAC, IEEE TON, IEEE TMC.

Chengzhong Xu (Fellow, IEEE) Chengzhong Xu received his Ph.D. degree from the University of Hong Kong in 1993. He is currently a Chair Professor of Computer Science, University of Macau, China. Prior to that, he was in the faculty of Wayne State University and Shenzhen Institutes of Advanced Technology of CAS. His recent research interests are in cloud and distributed computing, intelligent transportation and autonomous driving. He published two research monographs and more than 500 journal and conference papers and received more than 17000 citations. He was a best paper awardee or nominee of conferences, including HPCA'2013, HPDC'2013, ICPP'2015, and SoCC'2021. He was also a co-inventor of more than 150 patents and a co-founder of Shenzhen Institute of Baidou Applied Technology. He serves or served on a number of journal editorial boards, including IEEE TC, IEEE TCC, IEEE TPDS, JPDC, Science China, and ZTE Communication. Dr. Xu was the Chair of IEEE Technical Committee on Distributed Processing from 2015 to 2020. He was a recipient of the Faculty Research Award, Career Development Chair Award, and the President's Award for Excellence in Teaching of WSU. He was also a recipient of the "Outstanding Oversea Young Scholar" award of NSFC in 2010. Dr. Xu is an IEEE Fellow.

Yifan Zeng received the B.E. degree from the School of Cyber Science and Engineering, Wuhan University, China, in 2022. She is currently working toward the master's degree from the School of Cyber Science and Engineering, Wuhan University, China. Her research interests include edge computing, online learning and network optimization.

Renli Zhang received the B.E. degree in Information Security from Wuhan University, China, in 2020. He is currently pursuing the M.S. degree in the School of Cyber Science and Engineering at Wuhan University. His research interests include UAV-enabled wireless networks, network optimization, and online scheduling.