Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing

Abstract

As machine learning techniques are applied to a widening range of applications, high throughput machine learning (ML) inference servers have become critical for online service applications. Such ML inference servers pose two challenges: first, they must provide a bounded latency for each request to support consistent service-level objective (SLO), and second, they must be able to serve multiple heterogeneous ML models in a system as certain tasks involve invocation of multiple models and consolidating multiple models can improve system utilization. To address the two requirements of ML inference servers, this paper proposes a new ML inference scheduling framework for multi-model ML inference servers. The paper first shows that with SLO constraints, current GPUs are not fully utilized for ML inference tasks. To maximize the resource efficiency of inference servers, a key mechanism proposed in this paper is to exploit hardware support for spatial partitioning of GPU resources. With the partitioning mechanism, a new abstraction layer of GPU resources is created with configurable GPU resources. The scheduler assigns requests to virtual GPUs, called glets, with the most effective amount of resources. Unlike the prior work, the scheduler explores three-dimensional search space with different batch sizes, temporal sharing, and spatial sharing efficiently. The framework auto-scales the required number of GPUs for a given workloads, minimizing the cost for cloud-based inference servers. The paper also investigates a remedy for potential interference effects when two ML tasks are running concurrently in a GPU. Our prototype implementation proves that spatial partitioning enhances throughput by 61.7% on average while satisfying SLOs.

1 Introduction

The wide adoption of machine learning (ML) techniques poses a new challenge in server system designs. Traditional server systems have been optimized for CPU-based computation for many decades. However, the regular and ample parallelism in widely-used deep learning algorithms can exploit abundant parallel execution units in GPUs. Although powerful GPUs have been facilitating the training computation of deep learning models, the inference computation is also moving to the GPU-based servers due to the increasing computational requirements of evolving deep learning models with deeper layers [11, 35, 41].

However, the GPU-based inference servers must address different challenges from the batch-oriented processing in ML training servers. First, inference queries must be served within a bounded time to satisfy service-level objectives (SLOs). Therefore, not only the overall throughput of systems is important, but bounded response latencies for processing inference queries are also critical to maintain consistent service quality [15, 35, 41].

Second, to improve the utilization of server resources, multiple heterogeneous models need to be served even by a single node. As the amount of memory in GPUs increases, each GPU can maintain the parameters for multiple models in GPU DRAM, which enables fast switching of ML models without swapping models out of GPU. As even a single service can include multiple heterogeneous ML models [35], multiple models with different purposes co-exist in a system. The heterogeneity of ML models raises new scheduling challenges to map concurrent requests of heterogeneous models to multiple GPUs. Incoming queries for different ML models with their own computational requirements, must be properly routed to the GPUs to meet the SLO, while improving the overall throughput. In addition, the number of required GPU nodes must be dynamically adjusted to reduce the cost of serving inferences for cloud-based systems.

While the demands for GPU-based ML inferences have been growing, the computational capability of GPUs with many parallel execution units has been improving precipitously. Such ample parallel execution units combined with increasing GPU memory capacity allow multiple ML models to be served by a single GPU. In the prior study [35], more than one model can be mapped to a GPU, as long as the GPU can provide the execution throughput to satisfy the required

SLO. However, unlike CPUs which allow fine-grained time sharing with efficient preemption, GPUs perform only coarse-grained kernel-granularity context switches. In addition, to reduce the variance of execution latencies for a request, the GPU is allocated for a single batch of requests for a given model before accepting the next batch of the same or a different model [35]. Such coarse-grained time sharing incurs inefficient utilization of enormous computational capability of GPUs, as a single batch of an ML inference may not fill the entire GPU execution units.

However, the recent advancement of GPU architecture opens a new opportunity to use abundant execution resources of GPUs. Recent GPUs support an efficient spatial partitioning of GPUs resources (called MPS mechanism [12]). The MPS mechanism after the NVIDIA Volta architecture supports computational resources of a GPU to be partitioned to run different contexts simultaneously. Such a unique spatial partitioning mechanism can augment the limited coarse-grained time sharing mechanism, as the GPU resource can be spatially partitioned to serve multiple ML tasks concurrently. This unique spatial and coarse-grained temporal resource allocation in GPUs calls for a novel abstraction to represent partitioned GPUs and a new scheduling framework targeting high throughput ML servers under SLO constraints.

To address the emerging challenges of ML scheduling in partitionable GPUs, this paper proposes a new abstraction for GPUs called *glet*, which can create multiple virtual GPUs out of a single physical GPU with spatial partitioning. The new abstraction can avoid the inefficiency of the coarse-grained time sharing, by creating and assigning the most efficient GPU share for a given ML model. Each glet can be used for ML inferences independently from other tasks. Such a new abstraction of GPU resources allows latencies of ML execution to be predictable even when multiple models are concurrently running in a GPU, while achieving improved GPU utilization.

Based on the glet concept, we propose a ML inference framework prototyped on PyTorch interface. It can serve concurrent heterogeneous ML models in multi-GPU environments, with the auto-scaling support. The scheduling framework implements the glet abstraction, and its search space is multi-dimensional with spatial and temporal shares of GPU resources in addition to batch size adjustment. For each ML model, its computational characteristics are measured and registered to the framework. Based on the profiled information of each ML model, the scheduler routes requests to where the throughput would be maximized, while satisfying the SLO constraints.

One necessary mechanism for the spatial and temporal partitioning of GPU shares is to identify the potential performance overheads when two models are concurrently running on a GPU, where two virtual GPUs are mapped. Although a prior study [4] modeled such interference of concurrent kernels in pre-Volta MPS systems, this study proposes a newly tuned interference estimation model for concurrent kernel executions of ML inference workloads.

We evaluated the proposed ML inference framework on server systems with four and eight GPUs. The evaluation with four GPUs shows that the proposed scheduling technique with glets can improve the throughput with SLO constraints for five ML inference scenarios by 61.7%, compared to the one without partitioning GPU resources. The source codes will become publicly available after publication.

This study explores a new resource provisioning space of GPUs for machine learning inference serving. The contributions of this paper are as follows:

- It proposes a new GPU abstraction named *glet*, to support virtual GPUs with partitions of resources out of physical GPUs. It allows heterogeneous ML models to be mapped to multiple glets in the most cost-effective way.
- It proposes a scheduling framework for glets, which search the best schedule by multi-dimensional search considering batch sizes, temporal sharing, and spatial sharing. It adjusts the number of required GPUs for a given set of heterogeneous models, supporting auto-scaling.
- It proposes an interference model for glets for concurrent ML inference execution on partitions of a single GPU.

The rest of the paper is organized as follows. Section 2 describes the background of ML computation on GPUs and the prior scheduling technique for MLs on GPUs. Section 3 presents the motivational analysis of heterogeneous ML tasks on multiple GPUs. Section 4 proposes our design for glets to efficiently utilize GPU resources for heterogeneous ML tasks, and Section 5 presents the experimental results. Section 6 presents the related work, and Section 7 concludes the paper.

2 Background

2.1 Batch-Aware ML Inference Serving

Since rapid ML inference servers are required, an increasing number of service vendors are adopting GPUs [9, 13, 15, 22, 23, 30, 35, 38, 39, 41, 42] or even hardware accelerators such as TPUs [3, 7, 18, 29]. While GPU-based systems offer low latency for ML inference, obtaining high utilization is a challenging task, unlike ML training. The key difference between training and inference in terms of the GPU utilization is the suitability for *batching*. For training, since the input data is ready, the system can batch any number of input data, which allows GPUs to effectively leverage the massive parallelism. On contrary, the ML inference server underutilizes GPUs as it is an on-demand system where once the inference requests arrive, then and only then, the inference tasks can be scheduled to the compute engines.

One scheduling option is to wait until the desired number of inference requests to be accumulated and then initiate the execution for the large batch. However, the problem is,

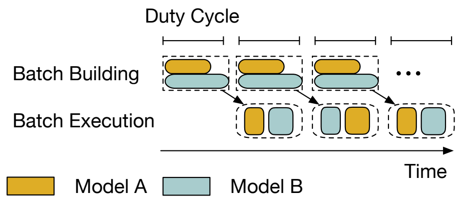


Figure 1: Round-based execution for two models consolidated on a GPU. The interval for a round is called *duty cycle*.

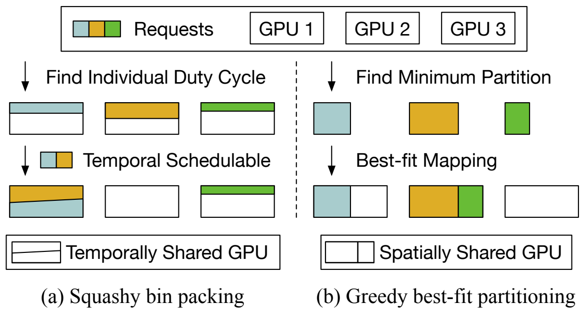
the applications cannot wait for the batch collection indefinitely, due to the service-level objective (SLO) requirements. Prior works [15, 34, 35, 41] have adopted *adaptive batching*, where a batch size is decided adaptively with estimated time to build and execute a batch size. By using profiled latency and observed incoming rate, the effective time required to build/execute a batch is estimated. Adaptive batching heuristically chooses maximum batch size that does not exceed SLO.

2.2 Temporal Scheduling for Multiple Heterogeneous Models

Temporal scheduling refers to the serialization of executions on GPUs where each inference takes up the entire GPU resource. Inference servers are multi-tenant systems where different models have different SLOs, which makes the SLO guarantees even more challenging when temporally scheduling multiple models. ML inference scheduling problem on GPU-based multi-tenant serving systems resembles the traditional bin packing problem. The capacity constraints of bins are the available resource on the GPUs, and the item weights are the necessary GPU resource to handle the given inference requests.

An inspiring prior work, Nexus [35], has tackled this problem and proposed a novel variant of bin packing algorithm, namely *squishy bin packing* (SBP). The term, *squishy*, is originated from the property that the required resource for processing a request (i.e., item) and its latency vary as the batch size changes. The SBP algorithm takes a set of models as input, each of which comes with its request rate.

Figure 1 illustrates an example of how the SBP algorithm is applied. In this scenario, the server handles two models, A and B, by building and executing the per-model batches simultaneously. The SLO violation occurs when the summation of batch building time and batch execution time exceeds either of the SLOs. Therefore, the SBP algorithm heuristically finds a maximum possible duration for batch building, called *duty cycle*, and the corresponding batch sizes in such a way that all the consolidated models would not violate the SLOs. The SBP algorithm repeats the duty cycles in a pipelined fashion until there is a change in the request rates, which would require a rescheduling.



(a) Squashy bin packing (b) Greedy best-fit partitioning

Figure 2: Multiple GPU Mapping

2.3 Spatial Sharing on GPU

Spatial sharing is a resource partitioning technique that splits a GPU resource into multiple pieces. It enables simultaneous inference serving in each partition. While temporal scheduling may potentially cause a GPU underutilization problem when the batch size is not sufficiently large to leverage the whole GPU resources, spatial sharing improves GPU utilization causing high throughput without SLO violation.

Modern server-scale GPUs offer spatial sharing features to users. NVIDIA Volta and Ampere GPUs, have added the computation resource provisioning and memory bandwidth isolation features on top of their prior generations, respectively. With these resource partitioning features, the users can split the given resource of a GPU into a set of *virtual* GPUs, each of which is assigned to a fraction of GPU resource, which we call *glet* in the rest of this paper[1].Also, we will call the GPU resource splitting as *GPU partitioning*.

A prior work GSLICE [17] leverages GPU sharing and related technical advances to increase throughput and utilization of GPUs. GSLICE employs a self-tuning algorithm for adjusting the amount of GPU resource based on performance feedback. After adjusting the amount of resource, the batch size is heuristically decided by the SLO for the given task. However, the solution provided in GSLICE is limited to a single GPU making it inapplicable to scaling the number of GPUs in inference servers.

2.4 Scaling Number of GPUs

Scaling the number of GPUs when scheduling multiple models for inference serving is a challenging problem. The number of GPUs must be minimal by maximizing utilization of GPU and the SLO of every model must be guaranteed. Figure 2 depicts two baseline GPU scaling methods inspired by prior work. One is the SBP algorithm and the other is greedy best-fit partitioning. The SBP algorithm computes each model's duty cycle and corresponding batch size as-

[1]In this paper, we only use the computation resource provisioning technique since we have at our disposal 2080 Ti GPUs, the microarchitecture of which is Turing, an older generation than Ampere that offers the memory bandwidth isolation feature.

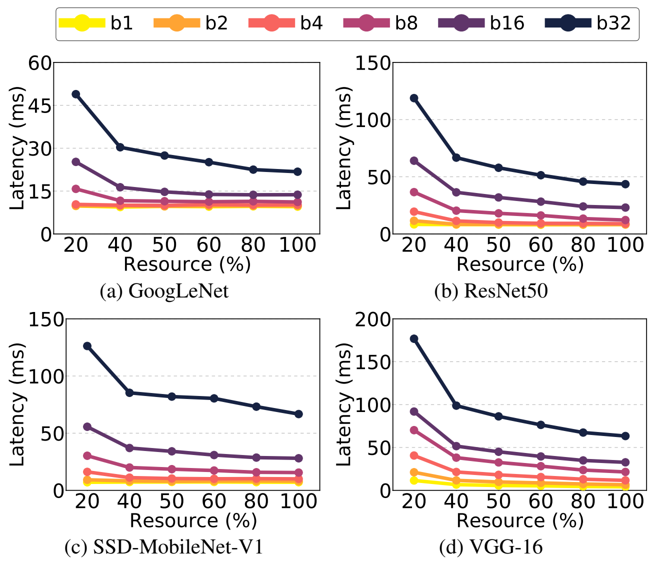


Figure 3: Batch inference latency as the fraction of compute resource assigned to the model inference changes from 20% to 100%, for the four ML models.

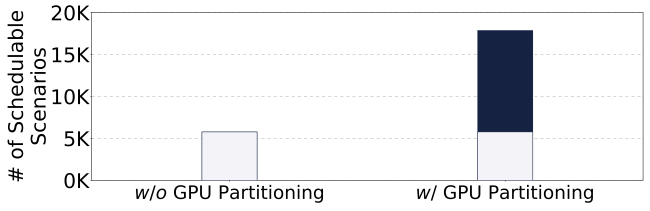


Figure 4: Number of schedulable scenarios when the SBP algorithm performs the scheduling *without* (right) and *with* (left) the GPU partitioning support.

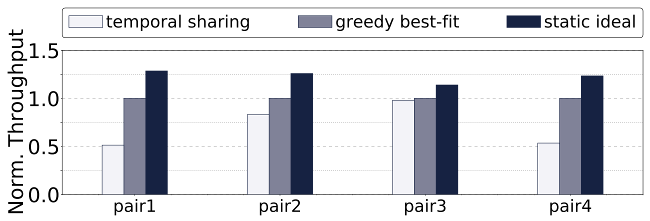


Figure 5: Comparison of SLO preserved throughput normalized to greedy best-fit partitioning scenario. Each pair of tasks are scheduled to two GPUs. Referring to the abbreviated notations in Table 4, each pair consists of (1) ssd, be, (2) res, vgg, (3) goo, mob, and (4) nas, den.

suming each model uses sufficient GPUs. It compares all eligible pairwise combinations. If a pair is eligible for temporal sharing, the pair will share one GPU and the batch size is adjusted to ensure SLO when both tasks are interleaved. The process continues until no more pairs can be temporally shared. Greedy best-fit partitioning chooses the minimum required partition size and allocates the partition to a GPU that has enough resources by best-fit searching.

Inspired by previous works, this paper aims to simultaneously employ both temporal and spatial scheduling to maximize utilization and minimize the number of required GPUs.

3 Motivation

3.1 Optimal Batch Size and Partition

To understand the performance implications of batching and GPU partitioning, we perform an experimental study, using four ML models: GoogLeNet, ResNet50, SSD-MobileNet-V1, and VGG-16. The detailed descriptions for the ML models and GPU server specifications are provided in Section 5.1.

Figure 3 shows the batch inference latency results as the batch size increases from 1 to 32. For each batch size, we sweep through the increasing fractions of GPU resource (i.e., glet size), ranging from 20% to 100% to observe how the batch size and computing resource utilization are correlated. When the batch size is large, the latency significantly drops as more resource is added. The large slope of the curves implies that the inference execution for the particular batch size can use the additional resource effectively to reduce the latency. On contrary, with the small batch, the latency is not largely affected by the amount of GPU resource, which implies resource underutilization. Hence, both batch size and amount of GPU resource must be considered as a joint factor when making cost-effective scheduling decisions.

3.2 Schedulability and GPU Partitioning

To evaluate the potentials of GPU partitioning on the schedulability improvement, we populate a significantly large number of possible multi-tenant inference serving scenarios and then measure the schedulability by counting the number of *schedulable* ones among the scenarios. For each scenario, models have one of the following request rates: 0, 100, and 200 requests per second (req/s). Excluding the scenario where all the models have a zero request ratio, we use the total of 19682 (= $3^9 - 1$) scenarios for the experiment.

Figure 4 reports the number of schedulable scenarios when we use the two different scheduling algorithms: 1) the default SBP algorithm without GPU partitioning support, and 2) the SBP algorithm that independently schedules two evenly split glets. With GPU partitioning, most scenarios identified as unschedulable by the default SBP algorithm shift to schedulable.

The results imply that GPU partitioning is capable of putting wasted GPU compute power to use, enabling higher resource utilization.

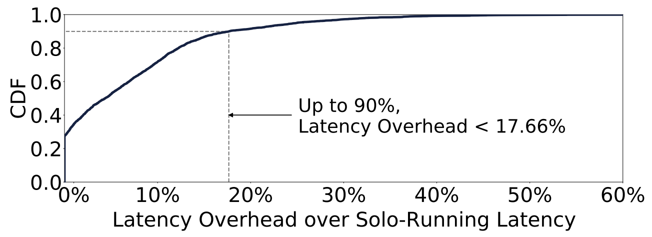


Figure 6: Cumulative distribution of latency overhead when the pairs of inference executions are consolidated on a GPU.

Features	Batch Tuning	Multi Model	GPU Scaling	Temporal Schedule	Spatial Schedule	Inter -ference
Clipper [15]	✓	✓	✓	✓	✗	✗
MArk [41]	✓	✗	✓	✗	✗	✗
INFaaS [34]	✓	✓	✓	✓	✗	✗
Nexus [35]	✓	✓	✓	✓	✗	✗
GSLICE [17]	✓	✓	✗	✗	✓	✗
Glet	✓	✓	✓	✓	✓	✓

Table 1: Comparison with prior works related to GPU ML inference serving

3.3 Performance of Effective Partitioning

To demonstrate how a cost-effective partitioning scheme affects performance, we compare the SLO preserved throughput of three partitioning scenarios. Figure 5 reports the normalized performance of each scenario. The first scenario, temporal scheduling, does not partition GPUs, but schedules tasks in a time-sharing manner. The second scenario partitions GPUs by our baseline algorithm, greedy best-fit introduced in Section 2.4. The algorithm greedily chose the minimum required partition size allocates the partition to a GPU that has enough resources in a best-fit searching manner. The last scenario, ideal static searches an ideal GPU partitioning ratio for performance and identically apply the ideal partitions for all pairs. Our baseline partitioning scheduler outperforms the non-partitioning scheduler by an average of 51%, proving the performance benefits of spatial sharing. Static ideal scenario shows an average of 23% better performance than greedy best-fit.

The results imply partitioning itself does not guarantee optimal utilization and cost-effective partitioning is necessary to provide better throughput.

3.4 Interference in Consolidated Executions

Cost-effective GPU partitioning allows enhancing the schedulability of SBP significantly. However, one important downside is the *performance interference* incurred from multiple inference executions concurrently running on a GPU.

To identify the interference effects, we perform an additional preliminary experimental study using a set of synthesized scenarios. We populate the pairs of two models drawn from five ML models (i.e., ${}_5C_2 = 10$) and associate the pairs with five different batch sizes (i.e., 2, 4, 8, 16, 32) to create 250 pairs in total. We also partition a GPU into two glets using various ratios: (2:8), (4:6), (5:5), (6:4), and (8:2). Then, we map the synthesized pairs to the different glet pairs to observe the interference effects in various settings.

Figure 6 exhibits the cumulative distribution function (CDF) of latency overhead due to the consolidated inference executions, in comparison with the case where the models are run independently. As noted in the figure, for 90% of the scenarios, the interference-induced overhead is lower than 18%, which is modest. However, the CDF reports the long tail, which suggests the interference effect could be severe in certain circumstances. Thus, the interference may cause the scheduling decisions to be significantly incorrect, when the interfered executions produce latencies that are largely off from the expected range. Motivated by the insight, we devise an interference model and leverage it to make the scheduling decisions more robust.

3.5 Comparison to the Prior Work

Table 1 provides a comparison of glet to related ML inference frameworks. All related work is capable of dynamically tuning batch size by either leveraging profiled latency or incoming request rates during runtime. Serving heterogeneous multiple models requires the scheduler to be aware of the type of the model and scaling the number of GPUs becomes even more challenging since the scheduler must provide a resource optimal adjusting mechanism. A few related works fail to address both issues.

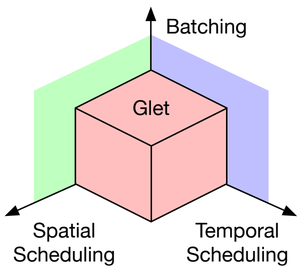
Regarding scheduling dimensions such as temporal and spatial sharing, a majority of work employs temporal sharing by leveraging profiled information of latency. Only GSLICE considered spatial sharing but it does not consider multi-GPU scheduling and temporal sharing. On the other hand, glet considers address all challenges, scheduling dimensions, and the potential interference among partitions in the same GPU making glet the only approach which considers all scheduling aspects.

4 Design

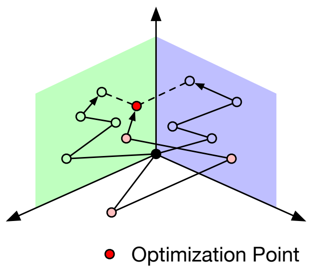
4.1 Overview

The goal of this paper is to devise a scheduling scheme for multi-tenant ML inference serving, which aims to assign incoming inference requests to the minimal number of GPUs by maximizing utilization. Figure 7 depicts an high-level illustration of our goal and methodology. Instead of exploring a subset of dimension, we propose a scheduler which fully explores all scheduling dimensions, batching, spatial, and temporal scheduling to find an optimal point of configuration.

For each ML model, a minimal performance profile is collected once for a trained model. Based on the profiles of current models, the scheduler distributes tasks to glets across



(a) Search space



(b) Configuration search

Figure 7: Graphical illustration of (a) search space dimensions and (b) process of searching optimal configuration.

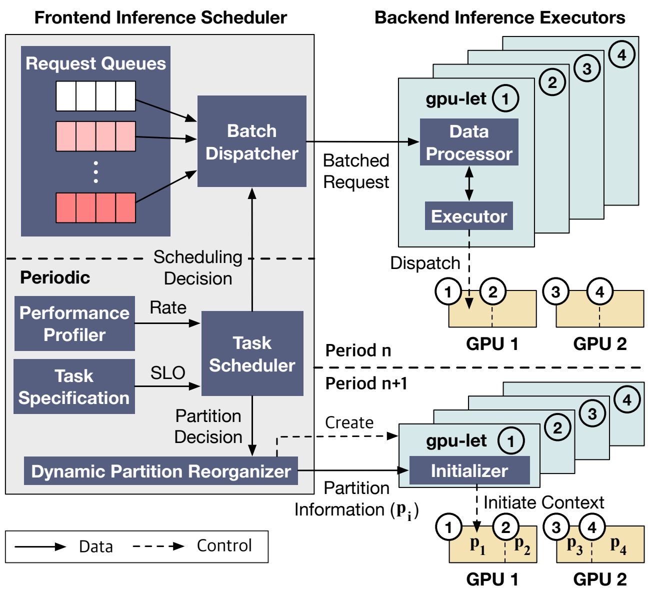


Figure 8: Overview of proposed scheduling scheme.

multiple physical GPUs. It minimizes the number of required physical GPUs, while satisfying the current request rates with the SLO mandates. The framework auto-scales the number of GPU servers adapting to the changes of request rates.

Figure 8 presents the overall organization of our proposed scheduler. The scheduler exploits the performance profiler, which not only monitors the incoming request rates, but also collects the performance statistics at offline, which are needed to make scheduling decisions (e.g., inference latency for a pair of batch size and glet size). The user-specified task specification provides the model-specific SLO target. At runtime, the task scheduler is periodically triggered to check if the rescheduling is required (e.g change of incoming rate). If required, the scheduler makes two types of decisions: 1) the size of glets, and 2) the model assignments to the glets. Based on the partitioning decisions, the dynamic partition reorganizer prepares the glets on the GPUs so that they can serve the inference request in the next period. The scheduling period is empirically determined based on the GPU partitioning latency so that the overhead for partitioning is hidden by the scheduling window.

Name	Description
$L(b,p)$	Latency function of batch size b and partition size p
$intf$	Interference overhead function
SLO_i	SLO (in latency) of model i
$glet.size$	Actual partition size of glet

Table 2: Definition of variables for Elastic Partitioning.

4.2 Challenge: Prohibitive Scheduling Search Space Size

The core challenge is that the scheduling decision is dependent on various variables that are dependent to each other. How a GPU is partitioned depends on the assigned model and batch size. Meanwhile, the batch size is dependent on the amount of allocated resource and how it is time shared with other models to ensure SLO. Therefore, the optimal scheduling decision would sit on the sweet spot in the search space built upon the three dimensions, 1) batching, 2) GPU partitioning, and 3) time sharing, which create a huge search space.

Let P be the number of GPU partitioning strategies on a GPU, N be the number of GPUs to schedule, and M be the number of models to serve. Each GPU can have P possible glets so there are total P^N possible strategies to partition N GPUs. Then, the M models can be placed on the glets, possibly having all the M models on a single glet. Since we need to check if the consolidation of multiple models violates the SLO, we must evaluate at most M^2 model placements per glet to assess schedulability. As we have NP glets on the system, the possible mappings of M models to the glets is NPM^2. In summary, the space complexity of scheduling problem is as follows:

$$\text{Search Space} = O(P^N NPM^2)$$

As the search space is prohibitively large, it is impractical to exhaustively search and pick an optimal solution. Therefore, we deploy a greedy approach, which effectively reduces the search space by allocating glets to GPU incrementally.

4.3 Scheduling Algorithm

Elastic partitioning: Algorithm 1 describes the overall procedure of scheduling models to glets. Table 2 lists the variables used in Elastic partitioning. The algorithm receives the following input for each model: 1) $L(b,p)$: profiled execution latency of batch b on partition size p, 2) $intf$: interference function, and 3) SLO: per-model SLO. For every scheduling period, the server checks incoming request rates of models that are running inference on the server. If rescheduling is

Algorithm 1: Dynamic glet Scheduling Algorithm

```
ElasticPartitioning($L(b,p)$, $intf$, $SLO$):
for each period do // If rescheduling is required
    Sort every model by $rate_m \times SLO_m$ in ascending order
    for each model $m$ do
        while IsRemainRate() and IsRemainGlets () do
            $rate \leftarrow$ Remaining rate of model $m$
            $p_{eff} \leftarrow$ MaxEfficientPartition()
            $p_{req} \leftarrow$ MinRequiredPartition($rate$)
            $p_{ideal} \leftarrow$ Min($p_{eff}$, $p_{req}$)
            $glet \leftarrow$ FindBestFit($p_{ideal}$, $SLO_m$, $intf$)
            Apply $glet$ to system
        end
    end
end
FindBestFit($p_{ideal}$, $SLO_m$, $intf$):
Sort every remaining glets by size in ascending order
for $glet$ in GetRemainGlets() do
    if $glet.size \geq p_{ideal}$ then
        if $glet$ is unpartitioned then
            Split and allocate $glet$ to $p_{ideal}$ size partition
        end
        $b \leftarrow \text{argmax}_{k \in \mathbb{N}}(L(k, glet.size) + intf \leq SLO)$
        if $b$ exists then
            TemporalScheduling($glet$)
            return $glet$
        end
    end
end
```

required, the scheduler performs elastic partitioning with the provided input (*line 1*).

Each model is sorted in ascending order by rate $\times$ SLO to ensure models with less room for time-sharing receive near optimal resource as possible (*line 2*). For each model m, the scheduler allocates one or more glets until the incoming rate can be satisfied or no more glet is left in the system (*line 3-4*).

Determining ideal glet size: Based on the observation from Section 3.1, elastic partitioning maximizes the system-wide throughput by allocating the most cost-effective sweet spot for glet size. Elastic partitioning chooses the smallest among 1) the most cost-effective glet size (p_{eff}), and 2) the minimum required glet size to serve $rate$ while avoiding the SLO violation (p_{req}). Figure 9 provides an graphical example of p_{eff} and p_{req}. To obtain the p_{eff}, the scheduler profiles at offline the affordable request rate for a given glet size. MaxEfficientPartition function calculates sweet spot of profiled glet size and uses the glet size at the knee as p_{eff} (*line 6*) . MinRequiredPartition examines the minimum size of glet, p_{req}, which is necessary to eschew the SLO violation under the given request rate (*line 7*). The scheduler always picks the minimum of p_{eff} and p_{req} to as ideal partition size p_{ideal} to ensure cost-effective glet size (*line 8*) .

Incremental allocation with best-fit: After finding p_{ideal}, FindBestFit performs a best-fit search. First, the scheduler sorts remaining glets by partition size in an ascending order (*line 14*). The algorithm searches through remaining glets until a $glet.size$ is greater or equal to p_{ideal} (*line 16*). Since the glets are sorted in ascending order, the sweeping naturally guarantees the best-fit. If the partition of glet can be split, which means the glet chosen has a size of 100%, the glet is split into two glets, each with a size of p_{ideal} and $100 - p_{ideal}$ (*line 17-19*). The maximum batch size b is decided and checked whether it can meet the SLO when there is additional interference-induced overhead (*line 20*). If a valid batch size exists, then $glet$ is chosen (*line 21*). After a $glet$ is chosen, elastic partitioning attempts temporal scheduling between $glet$ and previously allocated glets in the system (*line 22*). Temporal scheduling, first introduced in Section 2.2, is done by additionally considering $glet.size$ when calculating batch size and duty cycle in elastic partitioning. If successful, two glets will be merged to $glet$. Elastic partitioning updates the system's remaining and allocated glets with the result of FindBestFit (*line 10*).

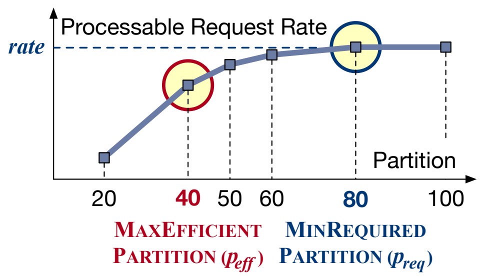


Figure 9: MaxEfficientPart and MinRequiredPart.

4.4 Modeling Interference

A key challenge in the interference handling is to predict latency increases when multiple inferences are executing in different glets of a same GPU. As shown in Figure 6, the interference effects are modest for the majority of consolidated executions, yet the overhead could be significant in infrequent cases.

In this study, we provide a simple interference-prediction model based on two key runtime behaviors of GPU executions. The interference effects by spatial partitioning is commonly caused by the bandwidth consumption in internal data paths including the L2 cache, and the external memory bandwidth. To find application behaviors correlated to the interference effects, we profiled the GPU with concurrent ML tasks with an NVIDIA tool (Nsight-compute). Among various execution statistics, we found the *L2 utilization* and *DRAM bandwidth utilization* are the most relevant statistics correlated to the interference.

Based on the observation, we build a linear model with the two parameters (L2 utilization and DRAM bandwidth utilization) as follows:

$$interference\_factor = c_1 \times L2_{m_1} + c_2 \times L2_{m_2} + c_3 \times mem_{m_1} + c_4 \times mem_{m_2} + c_5$$

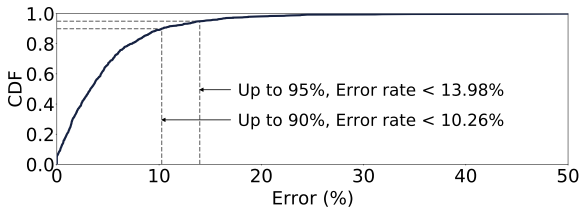


Figure 10: Cumulative distribution of relative error rate. Proposed analytical model can predict up-to 95% of cases with less than 13.98 % error rate.

Algorithm 2: glet Scaling Algorithm

```
SCALING(GPU_LIMIT):
for each period do
    N ← The number of used GPUs in previous period
    result ← ELASTICPARTITIONING with N GPUs
    while result is fail and $N < GPU\_LIMIT$ do
        $N \leftarrow N + 1$
        result ← ELASTICPARTITIONING with N GPUs
    end
    if result is fail then
        N ← The number of used GPUs in previous period
    end
end
```

$L2_{m_1}$ and $L2_{m_2}$ are L2 utilization of model 1 and 2, when they are running alone with a given percentage of GPU resource. mem_{m_1} and mem_{m_2} are memory bandwidth consumptions of model 1 and 2. Parameters (c_1, c_2, c_3, c_4, and c_5) are searched with linear regression.

We have profiled total 1,250 pairs (total 2,500 data) of inference interference and recorded how much interference each inference task has received. Among 2,500 data, we have randomly selected 1,750 data of execution as training data and 750 data for validation. Figure 10 presents the cumulative distribution of the prediction error with our interference model. The proposed model can predict up to 90% of cases within 10.26% error rate and up to 95% if 13.98 % of error is allowed.

4.5 Auto-Scaling GPUs

Algorithm 2 adapts a trial and error approach, which is viable due to the scalable nature of Elastic partitioning. Elastic partitioning is first attempted with the same number of GPUs of previous scheduling period (*line 2-3*). If the *result* is fail, due to the insufficient number of GPUs, same validating mechanism is repeated with one additional GPU. However, when the number of required GPUs exceeds the given limit, Algorithm 2 falls back to the result from previous period (*line 8-10*). As a result, SCALING find the minimum number of GPUs to satisfy ELASTICPARTITIONING.

System Overview	
CPU	20-core, Xeon E5-2630 v4
GPU	2 × RTX 2080 Ti
Memory Capacity	192 GB DRAM
Operating System	Ubuntu 18.04
CUDA	10.2
NVIDIA Driver	440.64
ML framework	PyTorch 1.10
GPU Specification	
CUDA cores	4,352
Memory Capacity	11 GB GDDR6
Memory Bandwidth	616 GB/sec

Table 3: The evaluated system specifications.

4.6 Implementation

SW Prototype for glet: As illustrated in Figure 8, our multi-GPU inference prototype serving system constitutes a frontend inference scheduler and backend inference executors. The SW modules were developed in C++ and the approximate lines of code is 21.8K. Glet is implemented as a separate process, which communicates with other servers through UNIX domain socket. Partition size of each glet is controlled by leveraging the MPS-supported resource provisioning capabilities.

Dynamic partition reorganizer: Our prototyped server monitors incoming rates and schedules glets for a period of 20 seconds. The period was decided by an observation where reorganizing a GPU partition takes approximately 10 to 15 seconds. The reorganization process includes spawning a new process with designated MPS resource, loading necessary kernels used by PyTorch, loading required models, and warming up. The whole process is executed in background and has minimal effect on violating SLO as evaluated in Section 5.

5 Evaluation

5.1 Methodology

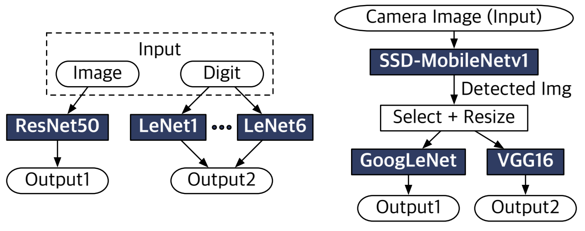
Inference serving system specifications: Table 3 provides a detailed description of the evaluated inference system and used GPU specifications. We use two identical multi-GPU inference servers each equipped with two NVIDIA RTX 2080 Ti GPUs, which offer post-Volta MPS capabilities. The table also provides the versions of used operating system, CUDA, NVIDIA drivers, and machine learning framework.

Each server operates as a backend server responsible for executing inference queries on two GPUs. One server additionally generates inference requests while the other server runs a frontend server to manage backend servers and make scheduling decisions. Both servers are network-connected, imitating inference serving system architecture, with 10 Gbps bandwidth.

Baseline scheduling algorithms: For our baseline, we have ported the Squishy bin-packing (`SBP`) algorithm (from

Model	Input Data (Dimension)	SLO (ms)
GoogLeNet (goo)	Imagenet (3x224x224))	66
LeNet (le)	MNIST (1x28x28)	5
ResNet50 (res)	Imagenet (3x224x224)	108
SSD-MobileNet (ssd)	Camera Data (3x300x300)	202
VGG-16 (vgg)	Imagenet (3x224x224)	142
MnasNet (nas)	Imagenet (3x224x224)	62
Mobilenet_v2 (mob)	Imagenet(3x224x224)	64
DenseNet (den)	Imagenet(3x224x224)	202
Base Bert (be)	Rand. Index Vector(1x14)	22

Table 4: List of ML models used in the evaluation.



(a) Scenario *game* (b) Scenario *traffic*

Figure 11: Two multi-model applications: *game* and *traffic*.

Nexus [35]) and greedy best-fit introduced in Section 2. Also, We have prepared two versions of our proposed algorithm, (glet +int) considers interference overhead while glet +int does not.

We do not provide a direct comparison to Nexus [35] due to the following reasons: 1) Nexus deploys optimizations which are orthogonal to our work, 2) several benchmarks that Nexus used in evaluation were not interoperable with our prototype server, such as models not supported by PyTorch. However, we deploy the same video processing models that Nexus have used to evaluate their system and show how spatially partitioning GPUs can further enhance performance.

DNN models: Table 4 shows the list of nine ML models used in our evaluation. The models have a wide spectrum of model sizes and topologies, which produce significant disparities among the inference latencies on the same GPU system. The last column, `SLO (ms)`, presents the per-model SLO latency constraint, which is set by doubling the solo execution latency of each model in our proposed system. We use the batch size of 32, since using a larger value than 32 engenders the SLO target latency unrealistically long.

Real-world multi-model applications: We use two real-world multi-model applications for evaluation, namely game and traffic. These two applications are first introduced in Nexus [35] but we were unable to use the open-source implementations due to their incompatibility issues with PyTorch. Thus, we develop the applications by ourselves, referring to application scenario descriptions provided by the paper.

Figure 11 delineates the detailed dataflow graph of the applications that contain ML models as well as the input/output data. The *game* application analyzes the digits and images

Name	Group Composition by Memory Footprint		
	<1GB	1GB - 2GB	>2GB
scen1	mob,be	nas,goo	-
scen2	-	den	vgg
scen3	mob	res	vgg
scen4	ssd	nas,den	-
scen5	le	ssd,nas	vgg

Table 5: Five request scenarios, each of which represents a particular composition of multiple models based on memory footprint. The amount of requests between each model within a group is equal.

from the streamed video games by using seven models in parallel. The *traffic* application is a traffic surveillance analysis with two phases, which are object detecting and image recognition. The SLO latency is set as 108 ms and 202 ms for game and traffic, respectively. Each SLO latency is calculated by doubling the longest model inference latency.

Deeper look into particular request scenarios: We particularly choose five model-level scenarios to take a deeper look. These five scenarios are characterized by the member of models and each respective memory footprint. Table 5 shows the details of each scenario.

Request arrival rate: In order to evaluate our proposed system, we sample inter-arrival time for each model from a Poisson random distribution, based on a previous literature [43] claiming that real-world request arrival intervals resemble a Poisson distribution.

Runtime evaluation of request scenarios and applications: For a given scenario or application, we evaluate the scheduling decisions by actually deploying scheduling results on a prototype server and measuring the SLO violation rates. To embrace the unpredictable performance variations, we iterate the experiment three times for each scenario and application, and pick the median SLO violation rate.

5.2 Experimental Results

Maximum achievable throughput comparison: We first evaluate the throughput implications of our schedulers. The *maximum achievable throughput* is defined to be the request rates that the schedulers would return `Schedulable` while the produced schedules do not violate the SLO target when experimented on our prototype server. We obtain the maximum achievable throughput of the schedulers by gradually increasing the request rate until SLO violation rate exceeds 1% of total requests.

Figure 12 reports the maximum achievable throughput for the two multi-model applications and five particularly chosen request scenarios when the four different scheduling algorithms are used. Our proposed glet +int scheduler offers higher throughput than both algorithms SBP and greedy best-fit by an average of 61.7% and 81.2%, respectively.

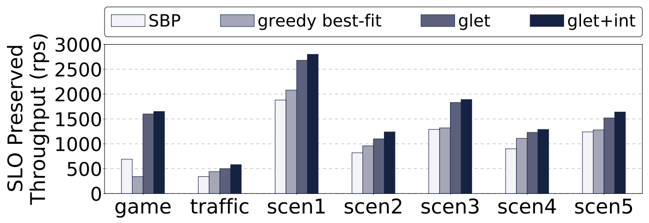


Figure 12: Maximum achievable throughput of the two multi-model applications (game and traffic) and five particularly chosen request scenarios.

Additionally, considering interference yields 7.5% better throughput on average. Although the benefit may seem marginal, we argue that such caution is necessary since a scheduler must be able to guarantee SLO at all times.

The reason why greedy best-fit shows such under-performance on game is because game consists of many short models (LeNet) and a single large model (ResNet50), making it more favorable for temporal sharing. ResNet50 received a 100% glet due to suboptimal partitioning. Hence, the advantages of partitioning a GPU were minimized and disadvantages of not being able to support temporal sharing became significant.

Note that the reported throughput improvement is achievable merely through the MPS features already available in the most server-class GPUs and scheduling optimization in software, using exactly the same GPU machine. Thus, by utilizing otherwise wasted GPU resources, the proposed scheduling scheme would be able to virtually offer cost savings for the ML inference service providers. For instance, glet +int achieves 1650 req/s throughput for game while SBP does 690 req/s, utilizing the identical physical system, which can be translated into 53.5% effective cost saving ($= \{1 - \frac{690}{1650}\} \times 100$).

Evaluation of meeting SLO: We take a deeper look into a certain level of maximum achievable request rate for the two multi-model application scenarios and five request scenarios to evaluate the efficacy of verifying interference. In order to evaluate and verify the system's capability to guarantee SLO we measure the percentage of requests that have violated the SLO, while counting dropped tasks also as SLO violating cases. We measured SLO violation by gradually increasing rates until both glet and glet +int consider the rate not schedulable, but only show the results of the maximum rate for brevity.

Figure 13 reports the SLO violation rate for each scenario and violation rates higher than 1% are highlighted with red rounds. Scheduler glet, which does not consider interference, shows violation rate higher than 1% even for rates that it considered to be schedulable for scen2, scen3, and scen5. However, glet +int successfully filters out such rate by either classifying such the rate as non-schedulable or successfully scheduling tasks to avoid SLO violation.

Evaluation of Scalability: To evaluate whether our prototype

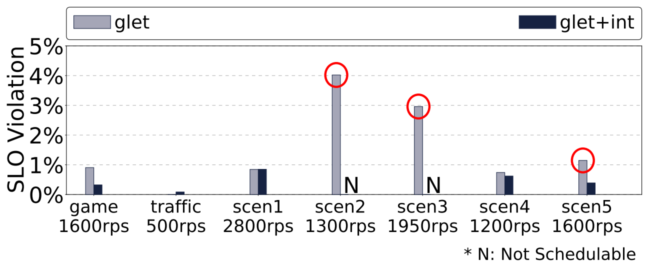


Figure 13: SLO violation rates of two multi-model applications and five request scenarios. Rates were gradually increased until both glet and glet +int concluded the rate to be *Not Schedulable*.

scheduler can successfully scale partitions for glets to accommodate fluctuating rates, we measure the performance of our scheduler while submitting inference requests with varying rates for all models in scen3. We have chosen scen3 because of its evenly distributed model size in order to reproduce a realistic workload.

In order to evaluate scalability beyond our testbed, we launched multiple servers with docker containers. By running one container per GPU and connecting 4 more identical GPUs, we conducted our experiment on total 8 servers. Additionally, the request generator and frontend server were specially tailored to send dimensional data, instead of actual data for inference request, to overcome network bottleneck between physical servers. The backend servers behave identically other than generating random data with dimensions provided from frontend server. Figure 14 reports how our scheduling framework performed for a 3,200 second window. The top graph shows a stacked graph of the accumulated throughput of each model. The second and third graph reports how many GPUs were scheduled and the sum of scheduled glet sizes, respectively. The last graph depicts the percentage of SLO violation (including dropped requests) for 20 second period. Between 0 and 1200 sec, the rate gradually increases and decreases to its initial rate. As the rate rises, our proposed scheduler successfully allocates more GPUs and glets to preserve SLO. When the rate decreases, the sum of utilized partitions also decreases by reorganizing partitions. The following wave, starting from 1400 sec, rises to a higher peak than the previous wave of requests. Nonetheless, our scheduler successfully adjusts the number of GPUs and preserve SLO, displaying SLO violation rate lower than 1%.

Comparison to ideal scheduler: We evaluate the scheduling capability of elastic partitioning by comparing the scheduling results produced from an ideal scheduler. To produce various model-level inference request scenarios, we use the same methodology described in Section 3.2, which populates a set of 19,682 possible scenarios. The ideal scheduler makes scheduling decisions by exhaustively trying all 4^4 glet combinations, where 4 GPUs can be partitioned into 4 cases. The search continues until all cases are searched or a case produces a viable scheduling result for a given request scenario.

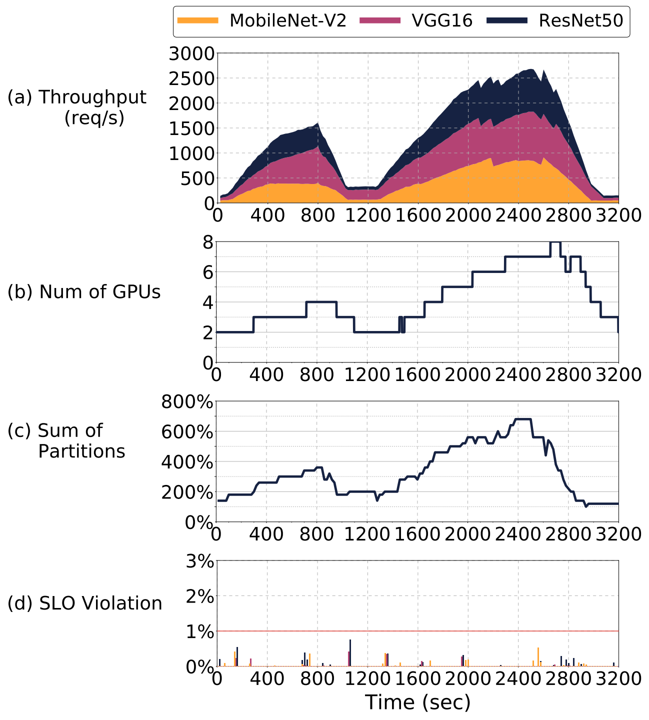


Figure 14: (a) Throughput(req/s), (b) number of GPUs, (c) sum of partition size (%) of glet, (d) SLO violation (%) of each model for a 3,200 second window.

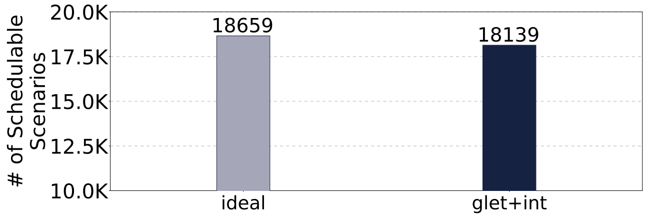


Figure 15: Comparison between number of schedulable scenarios for ideal scheduler and glet +int scheduler.

For a fair comparison, the ideal scheduler uses the same set of partitions as glet +int. Figure 15 compares the number of scenarios classified as schedulable by each scheduler. glet +int can schedule 520 fewer cases compared to ideal, which is 2.6% of the total 19,682 cases.

Figure 16 reports the maximum schedulable rate of each multi-model scenario. All rates are normalized to the maximum rate which ideal can provide. glet +int can achieve an average 92.6% of the maximum rate which the ideal scheduler can provide.

6 Related Work

Machine learning service platforms. A wide variety of computer systems and researches have been proposed to improve the quality of machine learning services [1, 2, 9, 13, 15, 17, 19, 20, 22, 24, 30, 33, 34, 35, 36, 37, 38, 41].

Although this paper did not cover training, past researches inspired this study with schedulers for optimizing GPU resource [16, 22, 31, 39, 42]. Another related research direction focus on how to ease the burden of deployment and optimization for machine learning across various platforms [6, 25, 26, 28, 30].

Interference estimation. Precise estimation of interference has been a key issue for high-performance computing. Bubble-up [27] and bubble-flux [40] models an application's sensitivity to cache and fits a sensitivity curve to predict performance. Han *et al.* [21] extends using sensitivity cure to distributed computing where interference can propagate among processes. Baymax [5] and Prophet [4] models concurrent task execution behavior for non-preemptive accelerators.

Multi-tenancy in Accelerator. GPU vendors have included HW/SW support for providing multi-tenancy to users such as NVIDIA Multi Process Service (MPS) [12], Multi Instance GPU (MIG) [14], and AMD MxGPU [10]. Academic researches also proposed multi-tenancy support in accelerators. Pratheek *et al.* devised page-walking stealing for multi-tenancy support in GPU [32] and Choi *et al.* [8] proposes fine-grain batching scheme. PREMA [7] proposed time-multiplexing solution with preemption and Planaria [18] supports multi-tenancy by partitioning processing elements.

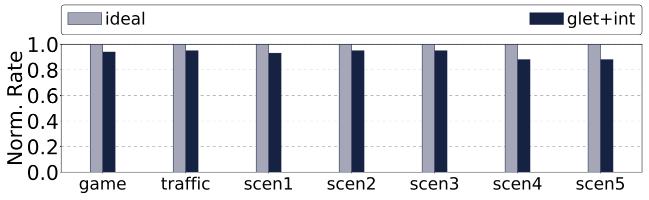


Figure 16: Comparison of normalized maximum schedulable rates of real-world multi-model benchmarks and five chosen request scenarios.

7 Conclusion

This study investigated an SLO-aware ML inference server design. It identified that common ML model executions cannot fully utilize huge GPU compute resources when their batch sizes are limited to meet the response time bound set by their SLOs. Using spatial partitioning features, glets significantly improved throughput of a multi-GPU configuration. The study shows ML inference servers call for a new abstraction of GPU resources (*glet*), instead of using conventional physical GPUs. The source code will become available.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, San-

jay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

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

[3] E. Baek, D. Kwon, and J. Kim. A multi-neural network acceleration architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 940–953, 2020.

[4] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[5] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[7] Y. Choi and M. Rhu. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 220–233, 2020.

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

[9] Amazon Corporation. *Amazon SageMaker Developer Guide*, 2020. `https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-dg.pdf`.

[10] AMD Corporation. *AMD MULTIUSER GPU:HARDWARE-ENABLED GPU VIRTUALIZATION FOR A TRUE WORKSTATION EXPERIENCE*, 2016. `https://www.amd.com/system/files/documents/amd-mxgpu-white-paper.pdf`.

[11] NVIDIA Corporation. *Deep Learning Inference Platform*, 2013. `https://www.nvidia.com/en-us/deep-learning-ai/solutions/inference-platform/`.

[12] NVIDIA Corporation. *Multi-Process Service*, 2019. `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`.

[13] NVIDIA Corporation. *TensorRT Developer's Guide*, 2020. `https://docs.nvidia.com/deeplearning/sdk/pdf/TensorRT-Developer-Guide.pdf`.

[14] NVIDIA Corporation. *NVIDIA Multi-Instance GPU User Guide*, 2021. `https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf`.

[15] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[16] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server. In *Proceedings of the 11th European Conference on Computer Systems (Eurosys)*, 2016.

[17] 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, New York, NY, USA, 2020. Association for Computing Machinery.

[18] S. Ghodrati, Byung Hoon Ahn, J. K. Kim, Sean Kinzer, Brahmendra Reddy Yatham, N. Alla, H. Sharma, Mohammad Alian, E. Ebrahimi, Nam Sung Kim, C. Young, and H. Esmaeilzadeh. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 681–697, 2020.

[19] 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 the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[20] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Mahmut Taylan Kandemir, and Chita R Das. Cocktail: Leveraging ensemble learning

for optimized model serving in public cloud. *arXiv preprint arXiv:2106.05345*, 2021.

[21] Jaeung Han, Seungheun Jeon, Young ri Choi, and Jaehyuk Huh. Interference Management for Distributed Parallel Applications in Consolidated Clusters. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[22] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[23] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018.

[24] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic Space-Time Scheduling for GPU Inference. In *Proceedings of the Conference and Workshop on Neural Information Processing Systems (NeurIPS)*, 2018.

[25] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel GPU task scheduling for deep learning. In *Proceedings of the Conference and Workshop on Neural Information Processing Systems (NeurIPS)*, 2020.

[26] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[27] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[28] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[29] N. Jouppi et. al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-Serving: Flexible, High-Performance ML Serving. In *Proceedings of the Conference and Workshop on Neural Information Processing Systems (NeurIPS)*, 2017.

[31] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the 13th European Conference on Computer Systems (Eurosys)*, 2018.

[32] B Pratheek, Neha Jawalkar, and Arkaprava Basu. Improving gpu multi-tenancy with page walk stealing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 626–639. IEEE, 2021.

[33] Kiran Ranganath, Joshua D Suetterlein, Joseph B Manzano, Shuaiwen Leon Song, and Daniel Wong. Mapa: Multi-accelerator pattern allocation policy for multi-tenant gpu servers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[34] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[35] 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)*, 2019.

[36] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. Alert: Accurate learning for energy and timeliness. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 353–369, 2020.

[37] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. Morphling: Fast, near-optimal auto-configuration for cloud-native model serving. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 639–653, 2021.

[38] Wei Wang, Sheng Wang, Jinyang Gao, Meihui Zhang, Gang Chen, Teck Ng, and Beng Ooi. Rafiki: Machine Learning as an Analytics Service System. In *Proceed-*

ings of the 44th International Conference on Very Large Data Bases (VLDB), 2018.

[39] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[40] Hailong Yang, Alex D. Breslow, Jason Mars, and Lingjia Tang. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[41] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[42] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[43] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.