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

Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon and Jaehyuk Huh
School of Computing, KAIST
{sbchoi, myshlee417, yjkim, jspark}@casys.kaist.ac.kr, {yjkwon, jhhuh}@kaist.ac.kr

Abstract

As machine learning (ML) techniques are applied to a widening range of applications, high throughput ML inference serving has become critical for online services. Such ML inference servers with multiple GPUs pose new challenges in the scheduler design. First, they must provide a bounded latency for each request to support a consistent service-level objective (SLO). Second, they must be able to serve multiple heterogeneous ML models in a system, as cloud-based consolidation improves system utilization. To address the two requirements of ML inference servers, this paper proposes a new inference scheduling framework for multi-model ML inference servers. The paper shows that with SLO constraints, GPUs with growing parallelism are not fully utilized for ML inference tasks. To maximize the resource efficiency of GPUs, a key mechanism proposed in this paper is to exploit hardware support for spatial partitioning of GPU resources. With spatio-temporal sharing, a new abstraction layer of GPU resources is created with configurable GPU resources. The scheduler assigns requests to virtual GPUs, called *gpulets*, with the most effective amount of resources. The scheduler explores the three-dimensional search space with different batch sizes, temporal sharing, and spatial sharing efficiently. To minimize the cost for cloud-based inference servers, the framework auto-scales the required number of GPUs for a given workload. To consider the potential interference overheads when two ML tasks are running concurrently by spatially sharing a GPU, the scheduling decision is made with an interference prediction model. Our prototype implementation proves that the proposed spatio-temporal scheduling enhances throughput by 61.7% on average compared to the prior temporal scheduler, 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. Powerful GPUs have been facilitating the training computation of deep learning models, and 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, 38, 45].

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 is important, but bounded response latencies for processing inference queries are also critical to maintain consistent service quality [15, 38, 45]. Second, to improve the utilization of server resources, many heterogeneous models are served by consolidated cloud-based systems. As even a single service can include multiple heterogeneous ML models [38], multiple models with different purposes coexist in a system. The heterogeneity of ML models raises 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 a prior study [38], 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. Such coarse-grained time

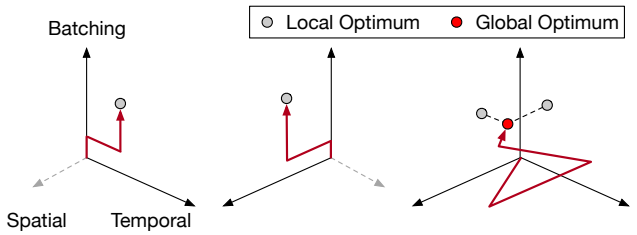
sharing incurs inefficient utilization of enormous computational capability of GPUs, as a single batch of an ML inference may not fill the entire GPU execution units.

However, the recent advancement of GPU architecture opens a new opportunity to better utilize the abundant execution resources of GPUs. Recent GPUs support an efficient spatial partitioning of GPUs resources (called MPS mechanism in NVIDIA GPUs [12]). The partitioning mechanism supports the computational resources of a GPU to be divided 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 *gpulet*, which can create multiple virtual GPUs out of a single physical GPU with spatial and temporal partitioning. The new abstraction can avoid the inefficiency of coarse-grained time sharing by creating and assigning the most efficient GPU share for a given ML model. 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 gpulet concept, we propose an ML inference framework prototyped with the PyTorch interface. It can serve concurrent heterogeneous ML models in multi-GPU environments with auto-scaling support. Figure 1 illustrates the extended search space of our scheduling mechanism. Our framework aims to find a global optimum by considering both temporal and spatial scheduling for enhanced performance. The search space for scheduling becomes three-dimensional with spatial and temporal shares of GPU resources in addition to batch size adjustment, unlike the prior work with two-dimensional searches [17, 38]. In the experimental results for SLO-preserved throughput presented by Figure 13, time scheduling and spatial scheduling yield on average 1,023 and 1,076 requests-per-second (RPS), respectively. The spatio-temporal scheduling significantly improves the SLO-preserved throughput to 1,584 RPS.

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 spatial and temporal partitioning of GPU shares is to identify the potential performance overheads when two models are concurrently running on a GPU. Our scheduling framework incorporates the interference estimation mechanism to consider the effect of concurrent execution.



(a) Temporal scheduling with batching (b) Spatial scheduling with batching (c) Temporal and spatial scheduling with batching

Figure 1: Multi-dimensional search space for providing globally optimal performance.

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 gpulets can improve the throughput with SLO constraints for seven ML inference scenarios by 61.7%, compared to the one without partitioning GPU resources.

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 *gpulet*, to support virtual GPUs with partitions of resources out of physical GPUs. It allows heterogeneous ML models to be mapped to multiple gpulets in the most cost-effective way.
- It proposes a scheduling framework for gpulets, which searches the most cost-effective 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.
- For accurate performance prediction, the scheduling framework considers the effect of interference among gpulets 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 prior scheduling techniques. Section 3 presents the motivational analysis of heterogeneous ML tasks on multiple GPUs. Section 4 proposes our design to efficiently utilize GPU resources for heterogeneous ML tasks, and Section 5 presents experimental results. Section 6 presents related work, and Section 7 concludes the paper.

2 Background

2.1 Batch-Aware ML Inference Serving

As high throughput ML inference serving has become widely required for online services, an increasing number of service vendors are adopting GPUs [9, 13, 15, 24, 25, 32, 38, 41, 42, 45, 46] or even hardware accelerators such as TPUs [3,

7, 18, 19]. 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 GPU utilization is the suitability for *batching*. For training, since the input data is ready, the system can batch a large number of input data, which allows GPUs to effectively leverage the massive parallelism. On the contrary, the ML inference server underutilizes GPUs as it is an on-demand system where inference tasks can be assigned to the compute engines once the requests arrive.

One scheduling option is to wait until the desired number of inference requests to be accumulated and then to initiate the execution for the large batch. However, the applications cannot indefinitely wait to collect a batch, due to the service-level objective (SLO) requirements. Prior work [15, 36, 38, 45] have adopted *adaptive batching*, where a batch size is decided adaptively with estimated times to build and execute a batch of the selected size. By using the profiled latencies and observed incoming rates, the effective time for a batch is estimated, and adaptive batching chooses the maximum batch size that does not violate the SLO.

2.2 Temporal Scheduling for ML on GPUs

Temporal scheduling allows sharing of a GPU where each inference takes up the entire GPU resource one at a time with time sharing. With multiple models with different execution characteristics and SLO requirements, guaranteeing SLO is challenging for the temporal scheduling of the heterogeneous models. The 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 an item weight is the necessary GPU resource to handle a given inference request.

An inspiring prior work, Nexus [38], 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 task (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 a given request rate. It assigns the inferences tasks across GPUs with a selected batch size, and may map multiple models to a single GPU with time sharing, if one task does not fully utilize a GPU.

Figure 2 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 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

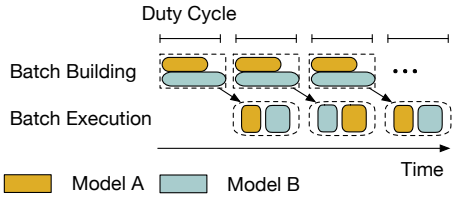
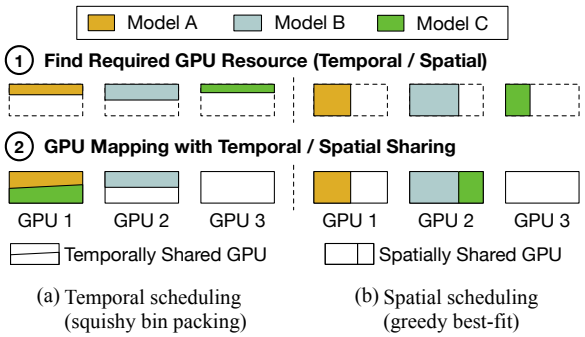


Figure 2: Round-based execution of SBP for two models consolidated on a GPU. *Duty cycle* is the interval for a round.



(a) Temporal scheduling (squishy bin packing)

(b) Spatial scheduling (greedy best-fit)

Figure 3: Temporal scheduling (SBP) vs. spatial scheduling (greedy best-fit partitioning).

until there is a significant change in the request rates, which would require rescheduling.

Baseline temporal scheduling algorithm: We use SBP as our baseline temporal sharing algorithm. Figure 3 (a) presents temporal sharing with batch adjustment by the SBP algorithm. The algorithm finds individual duty cycles and batch sizes for ML models with given request rates, and maps them to a minimum number of GPUs with temporal sharing. To map ML models to 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 further adjusted to ensure the SLO when both tasks are interleaved. The process continues until no more pairs can be temporally shared.

2.3 Spatial Sharing on GPU

Spatial sharing is a resource partitioning technique that splits a GPU resource into multiple pieces, as recent server-scale GPUs offer hardware-supported spatial sharing features to users. While temporal scheduling may potentially cause a GPU underutilization problem when the batch size is not sufficiently large to leverage all resources on a GPU, spatial sharing improves GPU utilization allowing high throughput without SLO violation.

With these resource partitioning features, the users can split the given resource of a GPU into a set of GPU *partitions*, each of which is assigned to a fraction of GPU resource[1]. Note

[1]In this paper, we only use the computation resource partitioning technique since we have at our disposal 2080 Ti GPUs, the microarchitecture of

that GPU *partitioning* is available on both NVIDIA MPS and Multi-Instance GPU (MIG), which has been featured since Ampere architecture GPUs. MIG provides physical partitioning with multi-GPU abstraction, while MPS provides logical execution resource partitioning by percentage. With physical partitioning, MIG allows partitions of memory capacity, memory bandwidth, and caches to be dedicated to each instance, in addition to execution cores.

A prior work, GSLICE [17], leverages GPU partitioning to increase throughput and utilization of GPUs. GSLICE employs a self-tuning algorithm for adjusting the amount of GPU resources per partition based on performance feedback. After adjusting the amount of resources, the batch size is heuristically decided by the SLO for the given task. However, the solution provided in GSLICE uses only spatial sharing and is limited to a single GPU.

Baseline spatial scheduling algorithm: As a baseline spatial sharing algorithm for our multi GPU framework, we use the *greedy best-fit* algorithm. Greedy best-fit algorithm chooses the minimum required partition size for each model to handle a given request rate with SLO constraint. It allocates the partitions of multiple ML models to GPUs through best-fit searching. Figure 3 (b) presents the spatial scheduling used by the greedy best-fit algorithm. Unlike the SBP or greedy-best fit algorithm, our scheduling scheme 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 4 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 resources, 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 the contrary, with a small batch, the latency is not largely affected by the amount of GPU resources, which implies that the GPU resource becomes underutilized when larger fractions are assigned. Hence, both batch size and amount of GPU resource must be considered as a joint factor when making cost-effective scheduling decisions.

which is Turing, an older generation than Ampere that offers the memory bandwidth isolation feature.

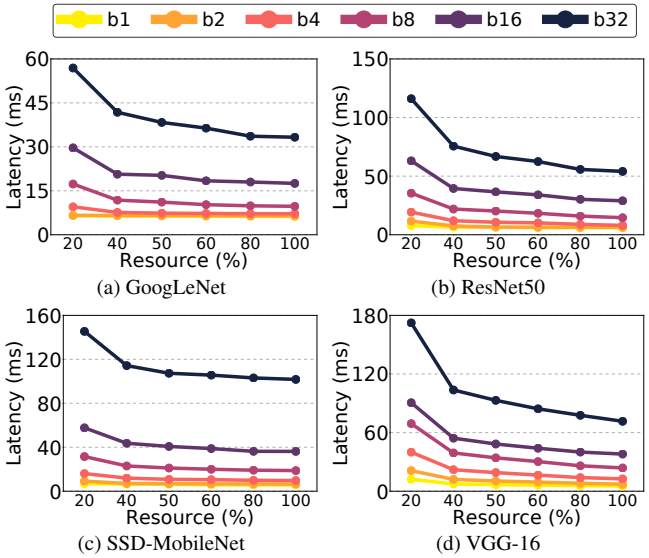


Figure 4: Batch inference latency as the fraction of computing resource assigned to the model inference changes from 20% to 100%, for the four ML models. Each curve represents a different batch size, and b*n* is a batch size of *n*.

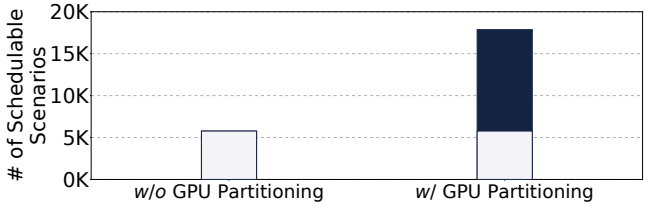


Figure 5: Number of schedulable scenarios when the SBP algorithm performs the scheduling *without* (left) and *with* (right) a fixed 1:1 GPU partitioning scheme.

3.2 Schedulability and GPU Partitioning

For a given set of SLO latencies for ML models, if incoming request rates are beyond the level that an inference server can cope with, the SLO will not be met as requests cannot be served on time. We define *schedulability* as the capability of a scheduling algorithm for serving the given request rates while not violating the SLO. A scheduler can improve schedulability by having better GPU utilization and in turn, having higher throughput with SLO satisfaction. To investigate the potential of GPU partitioning on the schedulability improvement, we evaluate a large number of possible multi-model inference serving scenarios. A *schedulable* scenario is the one in which the scheduler can successfully make a decision for the given rate while preserving SLO.

For the evaluation, we use nine models, and each corresponding SLO latency is listed in Table 3. For each scenario, models have one of the following request rates: 0, 100, and 200 requests per second (req/s). Since the zero req/s is in-

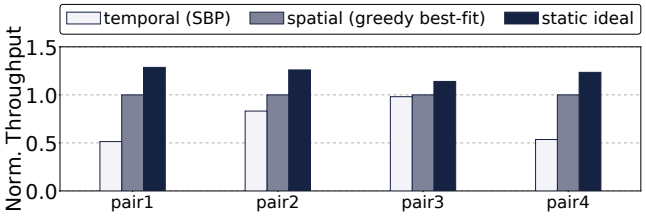


Figure 6: Comparison of SLO preserved throughput for temporal (SBP), spatial (greedy best-fit), and static ideal scheduling, normalized to spatial scheduling.

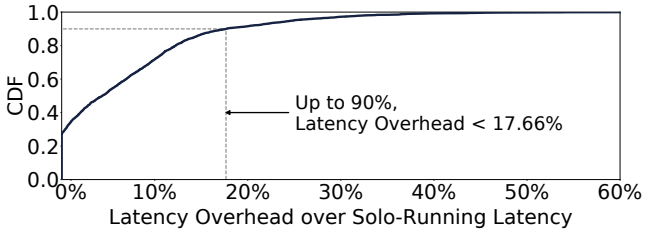


Figure 7: Cumulative distribution of latency overhead when pairs of inference executions are consolidated on a GPU.

cluded in the set of possible request rates, we assume that a subset of the nine models may not be served at all. Excluding the scenario where all the models have zero request rate, we obtained total 19,682 ($= 3^9 - 1$) scenarios for the experiment.

Figure 5 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 with GPU partitioning. In this motivational result, a GPU is split into two partitions with the same resource in each partition, although our scheduler later will use a wider range of partitioning of GPUs. With the fixed 1:1 GPU partitioning, schedulable scenarios increased significantly from 5,772 with SBP to 19,682 by SBP with two partitions. Even though the GPU is partitioned with a fixed 1:1 ratio, the results show that GPU partitioning is capable of putting wasted GPU compute power to use, enabling higher resource utilization.

3.3 Performance of Effective Partitioning

To demonstrate how a cost-effective partitioning scheme affects performance, we compare SLO preserved throughput of three scheduling schemes: temporal (SBP), spatial (greedy best-fit), and ideal scheduling. The SLO preserved throughput is the maximum throughput sustainable by a system while supporting SLOs for all models running in the system. Figure 6 presents the normalized SLO preserved throughput with the three scheduling schemes. We use two GPUs for this experiment, and a pair of ML models are scheduled. The models are selected from Table 3. The pair used for the experiment are (1) ssd/be, (2) res/vgg, (3) goo/mob, and (4) nas/den.

The first scheme, temporal scheduling, does not partition GPUs, but schedules tasks in a time-sharing manner with the SBP algorithm. The second schemes partitions GPUs by our baseline spatial scheduling algorithm (greedy best-fit) introduced in Section 2.3. The last scheme, static ideal exhaustively searches all possible GPU partitioning ratios among (2:8), (4:6), and (5:5) for two GPUs. For each pair of tasks, it uses a GPU partitioning option which yields the highest performance. For these selected sets of ML models, the spatial scheduling (greedy best-fit) outperforms the temporal scheduling (SBP) by 51% on average, proving the performance benefits of spatial sharing. The static ideal scheduling shows 23% improvements on average, compared to greedy best-fit. This experiment shows only limited example scenarios, but the results imply that partitioning can potentially improve the GPU utilization for certain scenarios, and better scheduling can further improve spatial scheduling.

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* caused by multiple inference executions concurrently running on a GPU. One common cause of such performance interference is the bandwidth sharing for the external memory, but other contentions on on-chip resources may engender performance degradation too. To identify the interference effects, we perform an additional preliminary experimental study using a set of scenarios with ML models running together on a GPU. The model pairs were chosen among five models from Table 3: GoogLeNet, LeNet, ResNet50, SSD-Mobilenet, and VGG-16 (i.e., $C(5,2)$ = 10). Each pair runs with five different batch sizes (i.e., 2, 4, 8, 16, 32) creating 250 unique pairs in total. We also partition a GPU into two partitions using five different ratios: (2:8), (4:6), (5:5), (6:4), and (8:2). Then, we map the ML model pairs to the different partition pairs to observe the interference effects in various settings.

Figure 7 presents the cumulative distribution function (CDF) of latency overheads 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 modest (lower than 18%). However, the CDF reports the long tail, suggesting that the interference effect could be severe in certain circumstances. In such cases, the interference may cause incorrect scheduling decisions, and the interfered task would experience 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, which reduces SLO violation rates.

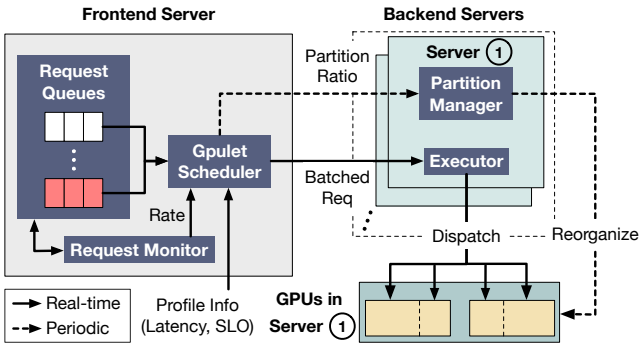


Figure 8: Overview of the scheduling framework with gpulets.

4 Design

4.1 Overview

The goal of this study is to devise a scheduling framework for multi-model ML inference serving, which aims to assign incoming inference requests to the minimal number of GPUs while maximizing their utilization. The scheduling of ML inference workloads with SLO requirements must consider three aspects: batching, temporal scheduling, and spatial scheduling. Unlike the prior approaches which consider a subset of the three dimensions [17, 38], we propose a scheduler that fully explores all three dimensions to find the most effective point for scheduling.

In this study, using the spatial sharing capability of GPUs, we introduce an abstraction of GPU partition, called *gpulets*. Multiple gpulets can utilize a GPU by both temporal and spatial sharing. For each trained ML model, a minimal performance profile is collected offline. Based on the profiles of models, the scheduler distributes tasks to gpulets across multiple physical GPUs. Our scheduling framework minimizes the number of required physical GPUs while satisfying the current request rates with the SLO requirements. Also, our framework auto-scales the number of GPU servers by adapting to the changes in request rates.

Figure 8 presents the overall architecture of our proposed scheduler. The scheduler is composed of a frontend server responsible for making scheduling decisions, and multiple backend servers for executing the decisions. The scheduler on the frontend decides and sends batched requests to the backend servers, and the executor in each backend server dispatches requests to GPUs The scheduling decision is made by utilizing profiled information of each model (e.g., SLO and inference latency for a pair of batch size and partition size) and incoming request rates. The request monitor tracks the number of newly arrived requests per second for each model. Based on the tracked request rates, the gpulet scheduler decides whether a new organization of partitions is required if the change of request rates is significant enough to update scheduling decisions. If a reorganization is necessary, the new partition ratios are sent to the backend server responsible for the GPU which needs reorganizing. The partition manager in the backend server prepares the partitions on the GPUs so that they can serve requests with the new partition ratios. The scheduling period is empirically determined based on the GPU partitioning latency to make the overhead of partitioning hidden by the scheduling window.

4.2 Search Space Challenge

The challenge of the three-dimensional scheduling space (batching, temporal, and spatial sharing) for gpulets is that the scheduling decision is affected by several variables dependent on each other. The best size of GPU partition depends on the computational requirement of the model and batch size. Also, the batch size is dependent on the amount of allocated resource and how it is temporally shared with other models to ensure SLO. Therefore, the most cost-effective configuration would sit on the sweet spot in the search space built upon the three dimensions, which creates a huge search space.

To represent the search space, let P be the number of possible GPU partitioning options on a GPU, N be the number of GPUs to schedule, and M be the number of models to serve. Therefore, there are total P^N possible options to partition N GPUs. The M models can be placed on a partition, possibly having all the M models on a single partition. Since we need to check if the consolidation of multiple models violates the SLO, we must evaluate at most M^2 model placements per partition to assess schedulability. As we have NP partitions on the system, the possible mappings of M models to the GPUs is NPM^2. The complexity of search space is as follows:

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

As the search space is prohibitively large, it is impractical to exhaustively search and pick a solution. To address the problem, we take a greedy approach, which effectively reduces the search space by allocating partitions to gpulets on GPU incrementally.

4.3 Elastic Partitioning Algorithm

This section discusses our scheduling algorithm called *elastic partitioning* which finds an efficient set of gpulets for given ML inference tasks.

Elastic partitioning: Algorithm 1 describes the overall procedure of scheduling ML models to gpulets. Table 1 lists the variables used in the algorithm. The algorithm receives the following inputs for each model: (1) L (b,p): profiled execution latency of batch size b on partition size p, (2) *intf*: interference function, (3) *SLO*: per-model SLO in latency, and (4) *gpulet.size*: size of partition allocated for *gpulet*. For every scheduling period, the server checks the request rates of each model. If rescheduling is required, the scheduler performs elastic partitioning with provided inputs (*line 1*).

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
$gpulet.size$	Actual partition size of gpulet

Table 1: Definitions of variables for Algorithm 1.

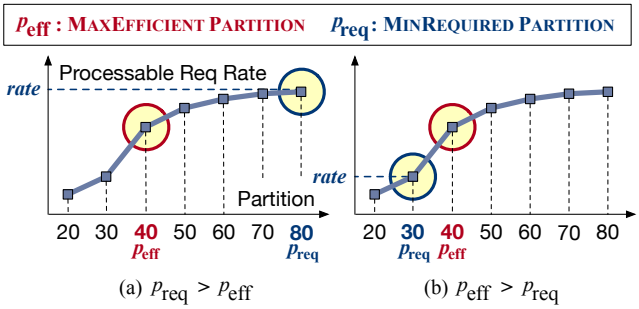
Algorithm 1: Gpulet Scheduling Algorithm

```
   ELASTICPARTITIONING(L(b,p), intf, SLO):
1  for each period do // If rescheduling is required
2      Sort every model by rate_m × SLO_m in ascending order
3      for each model m do
4          while ISREMAINRATE() and ISREMAINGPULET()
             do
5              rate ← Remaining rate of model m
6              p_eff ← MAXEFFICIENTPARTITION()
7              p_req ← MINREQUIREDPARTITION(rate)
8              p_ideal ← MIN(p_eff, p_req)
9              gpulet ← FINDBESTFIT(p_ideal, SLO_m, intf)
10             Apply gpulet to system
11         end
12     end
13 end
   FINDBESTFIT(p_ideal, SLO_m, intf):
14 Sort every remaining gpulets by size in ascending order
15 for gpulet in GETREMAINGPULETS() do
16     if gpulet.size ≥ p_ideal then
17         if gpulet is unpartitioned then
18             Split and allocate gpulet to p_ideal size partition
19         end
20         b ← argmax_{k∈ℕ}(L(k, gpulet.size) + intf ≤ SLO)
21         if b exists then
22             TEMPORALSCHEDULING(gpulet)
23             return gpulet
24         end
25     end
26 end
```

Each model is sorted in ascending order by $rate \times SLO$, which corresponds to the amount of work needed for the model (*line 2*). We allocate starting from the model with the least amount of work to the model which requires the most amount of work as a heuristic optimization for allocating resources. For each model m, the scheduler allocates one or more gpulets until the incoming rate can be satisfied or no more gpulet is left in the system (*line 3-4*).

Determining the most effective gpulet size: Based on the observation from Section 3.1, the scheduler maximizes the system-wide throughput by allocating the most cost-effective size for gpulet.

p_{eff} is the partition size that yields the highest performance per resource, which is the knee point in Figure 4. It is determined during profiling. p_{req} is the partition size satisfying SLO with the batch size that can handle the input rate. When request rates are low, p_{req} can be smaller than p_{eff}.



(a) $p_{\text{req}} > p_{\text{eff}}$ (b) $p_{\text{eff}} > p_{\text{req}}$

Figure 9: Max efficient partition (p_{eff}) and min required partition (p_{req}).

The scheduler chooses p_{eff} as the best partition size unless p_{req} is smaller than p_{eff}. If p_{req} is smaller than p_{eff}, p_{req} is chosen for the partition size, not to overprovision the GPU resource. Note that for a given model, a matching batch size is fixed for the p_{eff} of the model. Figure 9 presents two cases of p_{eff} and p_{req}. In the algorithm, MAXEFFICIENTPARTITION calculates a sweet spot of profiled gpulet size and uses the gpulet size at the knee as p_{eff} (*line 6*) . MINREQUIREDPARTITION examines the minimum size of gpulet, p_{req}, which is necessary to support the SLO on under the given request rate (*line 7*). The scheduler picks the minimum of p_{eff} and p_{req} as the ideal partition size p_{ideal} to ensure cost-effective gpulet size (*line 8*) .

Incremental allocation with best-fit: After finding p_{ideal}, FINDBESTFIT performs a best-fit search. First, the scheduler sorts the remaining gpulets by partition size in ascending order (*line 14*). The algorithm searches through remaining gpulets until a $gpulet.size$ is greater or equal to p_{ideal} (*line 16*). Since the gpulets are sorted in ascending order, the sweeping naturally guarantees the best-fit. If the partition of gpulet can be split, which means the chosen gpulet has a size of 100%, the gpulet is split into two gpulets, each with a size of p_{ideal} and $100-p_{\text{ideal}}$ (*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 the gpulet is chosen (*line 21*).

Temporal scheduling for gpulets: After a $gpulet$ is chosen, elastic partitioning attempts temporal scheduling between the returned $gpulet$ and previously allocated gpulets in the system (*line 22*). Temporal scheduling for gpulets follows the same rules which is introduced in Section 2.2: 1) adjust the duty cycle and batch size accordingly, and 2) check whether the SLO can be guaranteed for all models. We introduce an additional rule to consider $gpulet.size$ when calculating the batch size and duty cycle. For every pair of gpulets, the aforementioned rules are applied to see if temporal sharing is available. If a pair of gpulets has a different *size*, the larger *size* will be chosen to check if the SLO can be successfully guaranteed or not. If successful, two gpulets are merged to a single gpulet, thus reducing the total number of required gpulets. The scheduler updates the remaining and allocated

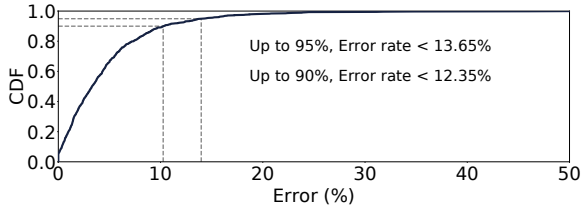


Figure 10: Cumulative distribution of relative error rate. Proposed analytical model can predict up-to 95% of cases with less than 13.98 % error rate.

gpulets with the result of FINDBESTFIT (*line 10*).

Reduced Search Space: Because the cost of iterating through every possible strategy is not required, the search space introduced in Section 4.2 is reduced as follows.

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

Instead of searching every case of P possible partitioning, for every N GPU in the system, each GPU is partitioned incrementally and the cost of checking temporal scheduling for M models still remains. Our search algorithm practically approximates the ideal one and removes P^N from the full search complexity. As a cost of the approximation, it may not always produce a theoretical optimal result. However, our evaluation shows that the algorithm performs closely to the ideal one as presented in Figure 16 (Section 5).

4.4 Modeling Interference

A key challenge in interference handling is to predict latency increases when multiple inferences are executed in different gpulets of the same GPU. As shown in Figure 7, the interference effects are modest for the majority of consolidated executions, yet the overhead could be significant in a few cases.

To confine the interference effect, we provide a simple yet effective interference-prediction model based on two key runtime behaviors of GPU executions. The interference effects of spatial partitioning are 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 profile the GPU with concurrent ML tasks with an NVIDIA tool (Nsight-compute). Among various execution statistics, *L2 utilization* and *DRAM bandwidth utilization* are the most relevant factors correlated to the interference.

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

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

$L2_{m_1}$ and $L2_{m_2}$ are L2 utilization of model m_1 and m_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 m_1 and m_2. Parameters (c_1, c_2, c_3, c_4, and c_5) are identified by running the 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.

Linear regression is chosen for its relatively high accuracy and low model construction complexity, so it satisfies our purpose in scheduling. Several prior studies have also used such linear models for predicting interference overheads [5, 43, 47].

Algorithm 2: GPU Scaling Algorithm

```
   SCALING(GPU_LIMIT):
1  for each period do
2      N ← The number of used GPUs in previous period
3      result ← ELASTICPARTITIONING with N GPUs
4      while result is fail and N < GPU_LIMIT do
5          N ← N + 1
6          result ← ELASTICPARTITIONING with N GPUs
7      end
8      if result is fail then
9          Report an unschedulable event
10     end
11 end
```

4.5 Scaling GPUs for Request Rate Changes

During a scheduling period, the monitor tracks the request rates of all ML tasks. If the rates change, it triggers the rescheduling procedure. The rescheduling procedure checks whether the changed rates can be sustained by the current number of GPUs. If not, it tries to increase the number of GPUs to support the SLOs for the new rates. Algorithm 2 presents the rescheduling procedure. It first attempts to use the same number of GPUs of the previous scheduling period (*line 2-3*). If the *result* fails due to the insufficient number of GPUs, the elastic partitioning is repeated with one additional GPU. However, when the number of required GPUs exceeds the given limit, it reports that an unschedulable event occurs.

4.6 Implementation

SW prototype: The SW prototype of our scheduler was developed in C++ and the approximate lines of code is 20.7K. We have chosen PyTorch for implementing ML inference due to its wide adoption in ML communities, in addition to the readiness to use C++ interfaces.

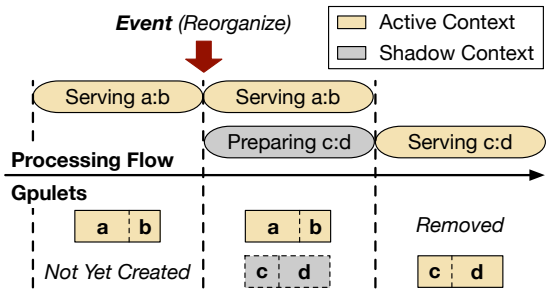


Figure 11: Illustration of dynamic partition reorganization.

Dynamic partition reorganization: NVIDIA provides the MPS control daemon, which allows users to control the proportion of computing resource reserved for processes spawned by the user. The amount of reserved resources is fixed when a process is created. Therefore, To change the proportion of reserved resource. a new process must be created with a new designated amount. This limitation affects our rescheduling procedure with a high cost of adjusting gpulets.

Our scheduling prototype controls gpulet partition size, by spawning a new proxy process to allocate a different amount of partition to gpulet. Preparing a new partition includes spawning a new process, loading kernels used by PyTorch, loading required models, and warming up. As illustrated in Figure 11, to hide the overhead of preparing new partitions when reorganizing is necessary, we overlap the procedure of preparing new partitions with serving the current partitions. The scheduling period for reorganization is 20 seconds which is a conservative estimate of time required for preparing a new partition.

5 Evaluation

5.1 Methodology

Inference serving system specifications: Table 2 provides a detailed description of the evaluated inference system and the used GPU specification. We use two identical multi-GPU inference servers, each of which is equipped with two NVIDIA RTX 2080 Ti GPUs supporting post-Volta MPS capabilities. The table also provides the versions of the operating system, CUDA, NVIDIA drivers, and machine learning framework.

Each GPU server operates as a backend server responsible for executing inference queries on two GPUs. One server additionally generates inference requests while the other server assumes the role of 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 Nexus [38]) and greedy best-fit introduced in Section 2. We evaluate two versions of our proposed algorithm, gpulet +int considers interference overhead while gpulet does not.

We do not provide a direct comparison to Nexus [38] due to the following reasons: 1) Nexus deploys optimizations that are orthogonal to our work, and 2) several benchmarks that Nexus used in evaluation were not interoperable with our prototype server, as the models are not supported by PyTorch. However, we deploy the same type of video processing models that Nexus used to evaluate the system and show how spatially partitioning GPUs can further enhance performance.

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 2: The evaluated system specifications.

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

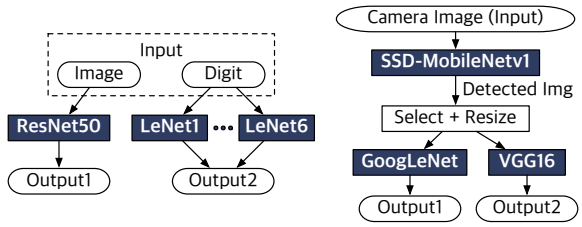
Table 3: List of ML models used in the evaluation.

ML models: Figure 12 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 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 detection 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 choose five model-level scenarios to take a deeper look into the multi-model inference serving. These five scenarios are characterized by the member of models and each respective memory footprint. Table 4 shows the details of each scenario.

Request arrival rate: We sample inter-arrival time for each model from a Poisson random distribution, based on previous literature [48] claiming real-world request arrival rates resemble a Poisson distribution.

Evaluation of request scenarios and applications: For a given scenario or application, we evaluate the scheduling decisions by deploying scheduling results on our prototype



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

Figure 12: Two multi-model applications: *game* and *traffic*.

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 4: Five request scenarios, each of which represents a particular composition of multiple models based on memory footprint. The amount of requests per model in a group is equal across the models in the group.

servers and measuring the throughput under SLOs and SLO violation rates. To consider the unpredictable performance variations, we iterate the experiment three times for each scenario and application, and pick the median result.

5.2 Experimental Results

SLO preserved max throughput: We first evaluate the throughput implications of our schedulers. The *SLO preserved max throughput* is defined to be the maximum achievable throughput while 99% of requests are processed within the SLO latency. We measure the SLO preserved max throughput of the schedulers by gradually increasing the request rate until SLO violation rate exceeds 1% of total requests.

Figure 13 reports the SLO preserved max throughput for the two multi-model applications and five scenarios for four different scheduling algorithms. Our proposed gpulet +int scheduler offers higher throughput than both algorithms SBP and greedy best-fit by an average of 61.7% and 81.2%, respectively. 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 low performance of greedy best-fit is caused by the lack of effective temporal sharing. In Figure 4, models show diminishing returns (over increasing GPU partitions) beyond a knee point. Note that different batch sizes can have different knee points. The greedy best-fit chooses the maximal batch size which satisfies SLOs and sets the partition for the batch size. The spatio-temporal scheduling can select the batch size and partition to better utilize GPU by considering smaller

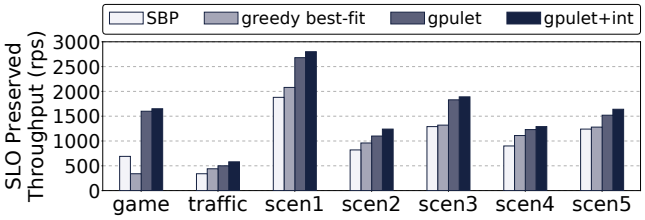


Figure 13: SLO preserved max throughput of the two multi-model applications (game and traffic) and five scenarios.

batches with temporal sharing across models. The limitation of greedy best-fit is clearly shown in game. Its flow has many small parallel tasks (LeNets), which cannot fill even a small GPU partition efficiently. Note that batch sizes cannot be increased arbitrarily as request rates and SLO limits it.

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 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, gpulet +int achieves 1,650 req/s throughput for game while SBP does 690 req/s, utilizing the identical physical system, which can be translated into 58.2% effective cost saving ($= \{1 - \frac{690}{1650}\} \times 100$).

The effect of interference model: This analysis shows how the interference model can avoid SLO violations by correctly incorporating the interference effect into the scheduling decision. In this result, we measured SLO violation rates by gradually increasing request rates until both gpulet and gpulet +int consider the current rate not schedulable. Figure 14 presents the SLO violation rates when the system is receiving the maximum request rate before both of them reach the not schedulable decision. In the figure, if the violation rate is higher than 1%, the case is highlighted with a red round. The scheduler gpulet, which does not consider interference, shows violation rates higher than 1% even for the rates that it considered to be schedulable for scen2, scen3, and scen5. However, gpulet +int successfully filters out such rate by either classifying as *not schedulable (N)* as shown in scen2 and scen3 or successfully scheduling tasks without violating the SLO such as scen5.

Evaluation of scalability: To evaluate whether our prototype scheduler can successfully scale gpulets 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 to reproduce a realistic workload.

To evaluate scalability beyond our testbed, we launched multiple servers by running each server with docker container. By running one container per GPU with four more identical GPUs, we conducted our experiment on total eight servers. Additionally, the request generator and frontend server were specially tailored to send dimensional data, instead of actual

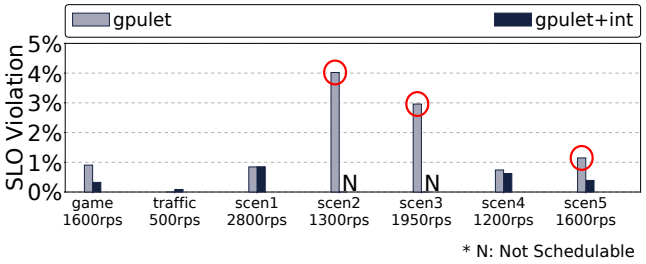


Figure 14: SLO violation rates of two multi-model applications and five scenarios. Request rates are increased until both gpulet and gpulet +int concluded the rate to be *Not Schedulable*.

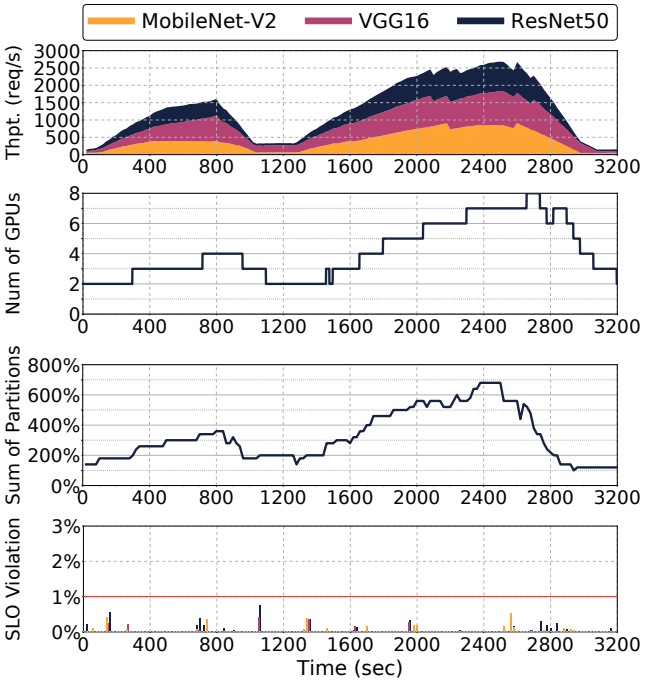


Figure 15: (a) Throughput(req/s), (b) number of GPUs, (c) sum of partition size (%) of gpulet, (d) SLO violation (%) of each model for 3,200 seconds.

data for inference request, for overcoming network bottleneck between physical servers. The backend servers behave identically other than generating random data with dimensions provided from frontend server. Figure 15 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 partition sizes of gpulets, respectively. The last graph depicts the percentage of SLO violation (including dropped requests) for 20 second period. Between 0 and 1,200 seconds, the rate gradually increases and decreases to its initial rate. As the rate rises, our proposed scheduler successfully allocates more GPUs to preserve SLO. When the rate decreases, the sum of utilized partitions also decreases by reorganizing partitions. The following wave, starting from 1,400 seconds, rises to a higher peak than the previous wave of requests. Nonetheless, our scheduler successfully adjusts gpulets and preserve SLO, guaranteeing SLO violation rate lower than 1%.

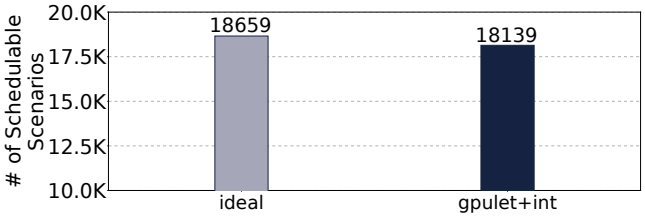


Figure 16: Comparison of the numbers of schedulable scenarios between the ideal scheduler and gpulet +int scheduler.

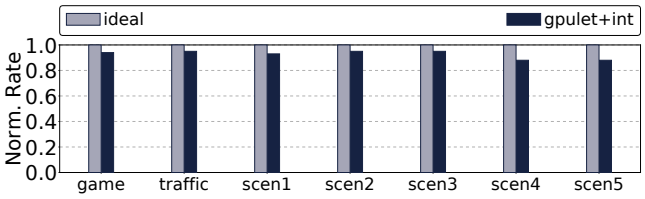


Figure 17: Comparison of the normalized maximum schedulable rates with real-world multi-model benchmarks and five scenarios.

Comparison to the 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 partition combinations, where 4 GPUs can be partitioned into either (2:8), (4:6), (5:5), or (10:0). The search continues until all cases are searched or a case produces a viable scheduling result for a given request scenario. For a fair comparison, the ideal scheduler uses the same set of partitions as gpulet +int. Figure 16 compares the number of scenarios classified as schedulable by each scheduler. gpulet +int can schedule 520 fewer cases compared to ideal, which is 2.6% of the total 19,682 cases.

Figure 17 reports the maximum schedulable rate of each multi-model scenario. All rates are normalized to the maximum rate which ideal can provide. gpulet +int can achieve an average 92.6% of the maximum rate which the ideal scheduler can provide.

6 Related Work

6.1 Prior ML Systems Studies

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,

Features	Batch Tuning	Multi Model	GPU Scaling	Temporal Schedule	Spatial Schedule	Interference Prediction
Clipper [15]	✓	✓	✓	✓	✗	✗
MArk [45]	✓	✗	✓	✗	✗	✗
INFaaS [36]	✓	✓	✓	✓	✗	✗
Nexus [38]	✓	✓	✓	✓	✗	✗
GSLICE [17]	✓	✓	✗	✗	✓	✗
Gpulet	✓	✓	✓	✓	✓	✓

Table 5: Comparison with prior work.

17, 21, 22, 24, 26, 32, 35, 36, 37, 38, 39, 40, 41, 45]. INFaaS is a platform for serving inference that guarantees SLO and minimizes the cost by choosing an adequate variation of a model [36]. INFaaS adopts a reactive approach when dealing with interference caused by co-locating model variants on the same hardware resource. Clockwork focuses on providing an accurately predictable system by leveraging the fact that the latency of inference is relatively consistent [21]. Clockwork preferred predictability over utilization gains from co-locating models and thus does not consider spatial sharing.

Although this paper did not cover training, past research inspired this study with schedulers for optimizing GPU resource [16, 24, 33, 42, 46]. Another related research direction focus on how to ease the burden of deployment and optimization for machine learning across various platforms [6, 27, 28, 31, 32]. Prior studies related to cluster scheduling have also influenced this paper [20, 30, 43].

Interference estimation: Precise estimation of interference has been a key issue for high-performance computing. Bubble-up [29] and bubble-flux [44] models an application's sensitivity to cache and fits a sensitivity curve to predict performance. Han *et al.* extend using sensitivity cure to distributed computing where interference can propagate among processes [23]. Prophet models concurrent task execution behavior for non-preemptive accelerators [4].

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 [34]. Choi *et al.* proposes fine-grain batching scheme [8]. PREMA proposed time-multiplexing solution with preemption [7]. Planaria supports multi-tenancy by partitioning processing elements [19].

6.2 Comparison to Prior Work

Table 5 provides a summarized comparison of our work to related ML inference frameworks. All the prior studies are capable of dynamically tuning batch size by either leveraging profiled latencies or incoming request rates during runtime. As more ML workloads are consolidated in cloud-based GPU servers, scheduling of multiple heterogeneous ML models in a system and scaling GPU servers under fluctuating request rates become more important. However, some prior studies do not consider such multi-model supports or GPU scaling.

Regarding scheduling dimensions such as temporal and spatial sharing, a majority of the prior work employ 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, our study addresses all challenges, scheduling dimensions, and predicting potential interference among partitions in the same GPU.

7 Conclusion

This study investigated an SLO-aware ML inference server design. It identified that common ML model executions cannot fully utilize GPU compute resources when their batch sizes are limited to meet the response time-bound set by their SLOs. By leveraging spatial partitioning features, our framework significantly improved throughput of multi-GPU configurations while supporting SLOs. Based on the new spatio-temporal scheduling technique, this study showed that a new abstraction of GPU resources (*gpulet*) can improve ML inference serving under SLOs. The source code is available at `https://github.com/casys-kaist/glet`.

8 Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF-2022R1A2B5B01002133) and Institute of Information & communications Technology Planning & Evaluation (IITP-2017-0-00466). Both grants are funded by the Ministry of Science and ICT, Korea.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay 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*. 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), 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)*, 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)*, 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)*, 2020.

[18] Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[19] 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)*, 2020.

[20] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM conference on Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[21] 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.

[22] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Cocktail: A multidimensional optimization for model serving in cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[23] 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.

[24] 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.

[25] 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)*, 2018.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Shanka Subhra Mondal, Nikhil Sheoran, and Subrata Mitra. Scheduling of time-varying workloads using reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

[31] 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.

[32] 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.

[33] 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.

[34] 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)*, 2021.

[35] 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 (SC)*, 2021.

[36] 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)*, July 2021.

[37] Wonik Seo, Sanghoon Cha, Yeonjae Kim, Jaehyuk Huh, and Jongse Park. Slo-aware inference scheduler for heterogeneous processors in edge platforms. *ACM Trans. Archit. Code Optim.*, 2021.

[38] 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.

[39] 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)*, 2020.

[40] 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 (SOCC)*, 2021.

[41] Wei Wang, Sheng Wang, Jinyang Gao, Meihui Zhang, Gang Chen, Teck Ng, and Beng Ooi. Rafiki: Machine Learning as an Analytics Service System. In *Proceedings of the 44th International Conference on*

Very Large Data Bases (VLDB), 2018.

[42] 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.

[43] Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads. In *Proceedings of the 19th International Middleware Conference*, 2018.

[44] 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.

[45] 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.

[46] 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.

[47] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[48] 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.

A Artifact Appendix

A.1 Abstract

To maximize the resource efficiency of inference servers, we proposed a key mechanism 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 gpulets, with the most effective amount of resources. The prototype framework auto-scales the required number of GPUs for a given workloads, minimizing the cost for cloud-based inference servers. The prototype framework also deploys a remedy for potential interference effects when two ML tasks are running concurrently in a GPU.

A.2 Hosting

The artifact is hosted on the following platforms:

- **Zenodo:** We have published the artifact on Zenodo: https://doi.org/10.5281/zenodo.6544909
- **GitHub:** Although the artifact provided in Zenodo contains all necessary and functional code, it is still in its early stage of development and needs improvement in terms of UI and code readability. Further improvement of code will be provided in the following GitHub repository: https://github.com/casys-kaist/glet.

A.3 Scope

The artifact is capable of:

- Serving machine learning inference on multiple multi-GPU servers.
- Adding new models defined and saved by TorchScript (provided as .pt files).
- Scheduling multiple models and guarantee SLO.
- Providing stand-alone ML inference executor for profiling performance and GPU resource usage.

The artifact does *not* provide the following:

- Utilize CPU for ML inference.
- Schedule heterogeneous GPUs.
- Guarantee availability (e.g. heartbeat, failure recovery).
- Add new models or deleting old models to serve while the frontend server is running.

A.4 Contents

A.4.1 SW Components

- **Frontend Server:** Receives requests from *M* clients and schedule requests to *N* backends.
- **Backend Server:** Receives batched requests from frontend server and conveys request to 0 servers running on the same machine.
- **Proxy:** Receives inputs from backend server and executes ML inference on a gpulet
- **Standalone Inference:** Executes inference on a GPU. Useful for debugging and profiling GPU resource utilization.
- **Standalone Scheduler:** Provides scheduling decision for given set of models and input rate of each model as stand-alone SW. Useful for inspecting scheduling decisions.

Please refer to *binaries.md* for further information of how to run and setup each components.

A.4.2 Models

The artifact includes 0 CNN models of VGG16 and ResNet50. Both models are stored as .pt file. All models are also available on Torchvision.

A.4.3 Dataset

A subset of ImageNet data and camera surveillance footage are each compressed as imagenet_data.tar and camera_data.tar respectively.

A.4.4 Docker Images

The following prerequisites must be installed in order to use the Docker images for this artifact:

- **Docker Ver.** >= 20
- **Nivdia-docker** (for utilizing GPUs)

Two Docker images are made public for experimenting with the provided artifact. One is the server Docker image available on sbchoi:glet-server and the other is the base Docker image used for building the backend Docker image available on sbchoi/glet-base.
We highly recommend using Docker images for experimenting since it contains all required code and scripts. For further instructions, please refer to the README file on https://github.com/casys-kaist/glet.

A.5 Requirements

A.5.1 Hardware

The artifact was evaluated on multi-GPU servers. Each GPU server had the following hardware specifications:

- **GPU:** NVIDIA RTX 2080ti (11GB global memory)
- **CPU:** Intel Xeon E5-2630 v4
- **Network:** Servers connected with 10 GHz Ethernet

A.5.2 OS and Kernel

The artifact was evaluated on Ubuntu 18.04 with a Linux kernel version 4.15.

A.5.3 Software

The artifact was built with the following drivers and libraries:

- LibTorch(PyTorch library for C++) = 1.10
- CUDA >= 10.2
- cuDNN >= 7.6
- Boost library>= 1.6
- OpenCV >= 4.0
- CMake >= 3.19

A.6 Experiment Setup

Experiments can be run by using the scripts provided in the artifact. We have also provided example files required for configuring experiments. Below are a few steps to configure multiple GPU servers using Docker images we have provided:

1. Run MPS daemon
2. Create and run an overlay network for Dockers
3. Setup and execute backend servers
4. Setup and execute frontend server, connecting all backend servers for serving inference.
5. Run clients
6. Analyze the content of glet/scripts/log.txt for how each request has been handled.

Please refer to the *README* file and *binaries.md* in https://github.com/casys-kaist/glet for detailed instructions of how to configure