USHER: Holistic Interference Avoidance for Resource Optimized ML Inference

Sudipta Saha Shubha and Haiying Shen, *University of Virginia;*
Anand Iyer, *Georgia Institute of Technology*

https://www.usenix.org/conference/osdi24/presentation/shubha

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

USHER: Holistic Interference Avoidance for Resource Optimized ML Inference

Sudipta Saha Shubha
University of Virginia

Haiying Shen
University of Virginia

Anand Iyer
Georgia Institute of Technology

Abstract

Minimizing monetary cost and maximizing the goodput of inference serving systems are increasingly important with the ever-increasing popularity of deep learning models. While it is desirable to spatially multiplex GPU resources to improve utilization, existing techniques suffer from inter-model interference, which prevents them from achieving both high computation and memory utilizations. We present USHER, a system that maximizes resource utilization in a holistic fashion while being interference-aware. USHER consists of three key components: 1) a cost-efficient and fast GPU kernel-based model resource requirement estimator, 2) a lightweight heuristic-based interference-aware resource utilization-maximizing scheduler that decides the batch size, model replication degree, and model placement to minimize monetary cost while satisfying latency SLOs or maximize the goodput, and 3) a novel operator graph merger to merge multiple models to minimize interference in GPU cache. Large-scale experiments using production workloads show that USHER achieves up to 2.6× higher goodput and 3.5× better cost-efficiency compared to existing methods, while scaling to thousands of GPUs.

1 INTRODUCTION

Driven by the breakthroughs achieved by Deep Learning (DL) models in a wide variety of domains [1–3], machine learning (ML) inference has emerged as the dominant workload that underpins many real-world applications. Our quest for improving the capability and accuracy of DL models has led to models growing in size rapidly [4]. While the success of DL models has been celebrated, it has come with a significant monetary cost: the increase in model sizes and popularity of ML model-based applications demand the use of expensive and power-hungry GPUs, leading to ML inference accounting for more than 90% of production costs [5]. Forecasts paint a gloomy picture: annual data center infrastructure and operating costs are projected to increase to over $76 billion by 2028 due to the rapid increase of the number of GPUs in the data centers, which is more than twice the estimated annual operating cost of Amazon AWS [6].

The exorbitant operating cost requirement has led to several systems innovations; state-of-the-art ML inference systems incorporate several optimizations that increase the *utilization* of GPUs. The fundamental technique to improve the utilization of a GPU is to use *batching*, where multiple inputs are combined and passed together through the model. Batching inputs together results in an increase in the compute requirements and thus improves the utilization of the GPU, albeit at the expense of increased latency. Unfortunately, batching is insufficient to optimally utilize a GPU because it is a *single knob* that influences two GPU resources: memory and compute, and thus is unable to saturate both resources at the same time. Moreover, since real-world batch sizes are not continuous in nature, there is no fine-grained control over the resources—while one batch size may severely underutilize the GPU in terms of memory or compute, the next possible batch size may not fit in the given resources or violate the strict latency Service-Level-Objective (SLO) (§2.1). When combined with the fact that request rates vary over time [3], real-world deployments have reported low GPU utilization averaging between 25% to below 50%, which has become a thorny pain point in reducing the total operational cost of large GPU clusters [7–10].

A natural solution to this problem is to place and simultaneously execute multiple models in a GPU. Unfortunately, previous research works have shown that this could result in *interference* between models due to resource contention which could introduce significant increase in inference latency, thus leading to SLO violation [11, 12]. An alternative is to leverage virtualization technologies that can divide the GPU resources; sadly GPU virtualization technologies available today are rudimentary and inflexible at best. NVIDIA Multi-Process Service (MPS) [13] facilitates simultaneous execution of multiple spatially multiplexed models by logically partitioning the computation space of the GPU among the models. Several scheduling systems [4, 14] have been proposed that leverage MPS and decide how much GPU computation space to allocate to each model to satisfy the latency SLO of the model requests based on offline profiling and place the models to the GPUs in such a manner that maximizes the utilization of the computation space of the GPUs. However, these works solely focus on compute utilization

and largely overlook the maximization of memory space utilization. NVIDIA multi-instance GPU (MIG) addresses the problem of inter-model interference by physically partitioning the computation and memory spaces of the GPU [15]. However, MIG only provides rigid partitioning, leading to GPU overprovisioning, and hence underutilization, and is only available in the latest generation of GPUs.

In this paper, we propose USHER, an end-to-end inference serving system that optimizes both GPU computation and memory utilizations by spatially multiplexing its resources in an interference-aware fashion. We design USHER from first-principles, based on a systematic analysis of performance and interference characteristics of the state-of-the-art solutions on real-world data traces (§2). Our analysis reveals several key observations. First, we may need to divide the workload of a model into multiple GPUs even when one GPU is enough to complete the workload within the latency SLO. Also, we need to perform this workload division holistically for all models. Second, not only the model parameter size, but also the intricate relationship between batch size, batch size-dependent resource requirements, and SLO contributes to making a model computation-heavy or memory-heavy. USHER leverages such observations in designing its three key components.

To accurately estimate the computation and memory requirements of a new model without incurring the prohibitive cost of offline profiling, USHER proposes a novel low-level GPU kernel analysis-based approach that first estimates which GPU kernels of the model will be executed concurrently and then sums up the resource requirements of those kernels. Finally, it takes the maximum sum across all sets of concurrent kernels as the highest resource requirement of the model (§3.2).

Based on the estimation, USHER needs to decide on the placement of each model that maximizes resource utilization without interference in both computation and memory spaces. Towards this, USHER incorporates a novel variant of a multi-dimensional bin packing scheduler [16–20]. To address the exponential complexity of holistic workload division, the scheduler first creates moderate-sized groups of models to maximize the probability of spatially multiplexing computation-heavy models with memory-heavy models within a group. Then, the scheduler decides the optimal workload division, batch size, and GPU placement decisions holistically for all models within a group by a heuristic algorithm (§3.3).

Finally, to minimize the cache interference among multiple spatially multiplexed models, USHER proposes a novel method that merges the computation graphs of multiple DL models to maximize the usage of GPU cache contents. Existing works [21–23] on computation graph merging reduce memory requirements by sharing weights across multiple models, which cannot maximize GPU cache usage. To this end, USHER merges the graphs in a manner that when the weight submatrix that is similar across different models is present in the GPU cache, the matrix multiplications of the different models associated with the submatrix are performed at the same time (§3.4).

We implemented USHER on Tensorflow (§4) and evaluated it on a wide variety of models and workloads using both real testbed and simulations. Our evaluation shows that USHER achieves up to 2.6× higher goodput and 3.5× better cost-efficiency against Shepherd [3], GPUlet [14], and AlpaServe [4], three representative state-of-the-art baselines (§5).

Overall, we make the following contributions in this paper:

1. We systematically analyze the underutilization of resources and inter-model interference in the state-of-the-art inference serving systems.
2. We propose USHER, a system that spatially multiplexes the inference serving of multiple DL models in an interference-aware and resource utilization-maximizing manner.
3. We evaluate USHER against the state-of-the-art baselines and show that it significantly outperforms them.

Table 1: DL models used in experiments.

Task & Domain	Model Name	Number of Parameters	Latency SLO	Dataset
Object detection (CNN-based vision models)	YOLO-v3 [24]	8.8M	197ms	COCO [25]
	R-CNN [26]	42M	284ms	
	MobileNetSSD-v2 [27]	15M	93ms	
Object recognition (CNN-based vision models)	ResNet-50 [28]	24M	108ms	ImageNet [29]
	ResNet-101 [28]	44M	198ms	
	ResNeXt-50 [30]	25M	116ms	
	ResNeXt-101 [30]	89M	407ms	
	SqueezeNet [31]	0.42M	14ms	
	ShuffleNet-v2 [32]	2M	40ms	
	MobileNet-v2 [33]	3.4M	64ms	
	DenseNet-121 [34]	7.6M	202ms	
	DenseNet-201 [34]	14.1M	405ms	
	Inception-ResNet-v2 [35]	56M	439ms	
	Inception-v3 [36]	25M	116ms	
	Inception-v4 [35]	43M	204ms	
	EfficientNet-B7 [37]	66M	217ms	
Language translation (Transformer-based language model)	GNMT [38]	278M	66ms	WMT 2019 [39]
Text classification (Transformer-based language model)	BERT [40]	110M	35ms	IMDB Movie Review [41]
Text generation (Transformer-based language models)	GPT-2 [42]	1.5B	140ms	WikiText [43]
	Llama-2 [44] (Large model)	13B	834ms	

2 EXPERIMENTAL ANALYSIS

We use *Cuti* and *Muti* to denote GPU computation and memory utilization, respectively, and use *Creq* and *Mreq* to denote their requirement from a model. Creq (or Mreq) of a model is the highest percentage of the total computation (or memory) space of a GPU consumed by the model at any point during its execution. We further use *Rreq* to denote the sum of *Mreq* and *Creq*. We use *C-heavy* and *M-heavy* to denote computation-heavy and memory-heavy, respectively. We use

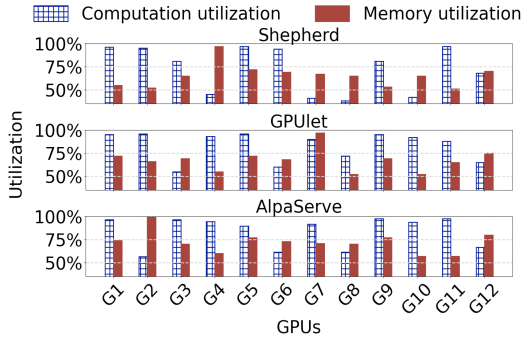


Figure 1: Performance of existing systems.

GPU# to denote the total number of GPUs required to satisfy the SLOs, and use *model#* to denote the number of models in a GPU. We use *C-space* and *M-space* to denote the available capacity of a GPU in computation and memory, respectively.

As [3, 14], we conducted analytical experiments using a mix of convolutional neural network (CNN) and Transformer models, which are typically the most widely used DL models in production deployments [3]. The 20 models used in our experiments are shown in Table 1. We got the trained models from HuggingFace repository [45]. The inference requests are also taken from the datasets. As [4, 14], the SLO of a model is taken as double the average inference latency of a single request in a Nvidia V100 GPU. As [3], the experiments were conducted on a GPU cluster formed by 12 Amazon EC2 servers of type p3.2xlarge. Each server has one Nvidia V100 GPU with 5760 computation cores, 16 GB GPU memory, one 2.3 GHz processor with 8 CPU cores, and 61 GB host memory. For the large model Llama-2, we utilized the DeepSpeed library [46] to partition the model into multiple partitions, allowing each partition to be loaded onto a GPU. Throughout the remainder of the paper, when referring to a model, it denotes the entire model for small models and a model partition for large models. Also, multiplexing refers to spatial multiplexing. As [3, 4], we used the Microsoft Azure Function trace 2019 [47] for the inference request rates and assigned the 46,000 function streams from the trace to the 20 models in a round-robin manner.

2.1 Performance of Existing Systems

We used Shepherd [3] to represent systems that do not allow spatial multiplexing and use GPUlet [14] and AlpaServe [4] to represent systems that allow spatial multiplexing. They aim to maximize GPU computation utilization and goodput. Goodput is defined as the number of inference requests completed within their latency SLOs per unit time.

Fig. 1 shows the average Cuti and Muti of each GPU. Shepherd achieves 41%-97% and 51%-97% Cuti and Muti, respectively. Though Shepherd uses batching to increase utilization (Ruti), it also increases the inference latency and memory requirement, which may become a bottleneck and limit the Cuti. GPUlet and AlpaServe increase the Cuti and Muti to 55%-97% and 54%-99%, respectively, due to their spatial multiplexing. However, there is still room for improvement. Also, Shepherd, GPUlet, and AlpaServe produce 14.2%-52%, 7%-40.1%, and 5.3%-44.9% $|Cuti - Muti|$ values, respectively. This is because the Creq and Mreq of a model are not necessarily correlated and hence, maximizing Cuti does not necessarily maximize Muti. Next, to study interference among models, we measured the goodput of each model with and without multiplexing in GPUlet and AlpaServe, and calculated their ratios. Fig. 2 shows the CDF of models versus the ratio. We see that 87% and 100% of the models have a ratio $\leq$ 0.55 in GPUlet and AlpaServe, respectively, meaning their goodputs are decreased by almost half due to the interference. GPUlet tries not to place the models in one GPU if their interference estimated by a regressor is high. However, it does not capture the interference among three or more models, and also does not address the cache interference problem.

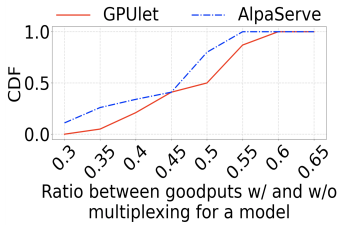


Figure 2: Impact of inter-model interference on goodput.

Observation 1. *The existing inference serving systems cannot maximize Cuti or Muti, and their model multiplexing significantly reduces the goodput due to model interference. In addition, maximizing Cuti does not necessarily maximize Muti.*

2.2 Opportunity of Workload Division

By equally dividing the workload (i.e., number of incoming requests per unit time) of a model into multiple GPUs, we essentially replicate the whole model in those GPUs. We use replication degree (RD) to denote the number of replicas of a model. In the example in Fig. 3a, models M1, M2, and M3 have 70%, 65%, and 40% Mreq (with 10%, 5%, and 10% for parameters, and the rest for intermediate data), respectively, to meet their SLOs with BS (batch size) = 8. Hence, the average Muti equals $\frac{70\%+65\%+40\%}{3} = 58.33\%$. Though the three GPUs have 30%, 35%, and 60% of the GPU memory unused, they are not enough to host any other model. Now, due to the strict latency SLO requirement, it is not possible to increase the BS of any model any further to increase the memory utilization since increasing the BS also increases the per-batch inference latency. Reducing the BS of M1 by half essentially conducts a workload division and lowers the intermediate data amount by half in each GPU where M1 is hosted, resulting in 40% Mreq in such a GPU. Fig. 3b shows a multiplexing schedule. The total number of GPUs (GPU#) is still 3 with 61.66% average Muti. Fig. 3c shows another multiplexing schedule, which performs workload division also for M2 and M3, and results in 2 GPU# and 100% Muti. This example shows that to increase Ruti in multiplexing, we may need to divide the workload to more GPUs than the minimum required and we need to decide the optimal workload division schedule *holistically* (instead of independently) for all models.

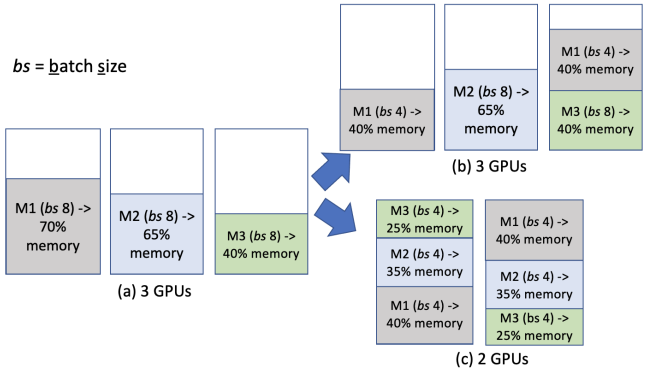
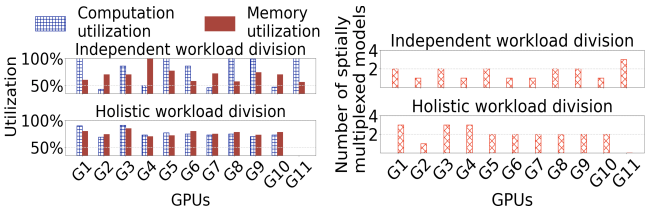


Figure 3: Performing workload division holistically for all models increases resource utilization.



(a) Resource utilization of each GPU by workload divison.

(b) Number of spatially multiplexed models in each GPU using workload division.

Figure 4: Effectiveness of holistic workload division.

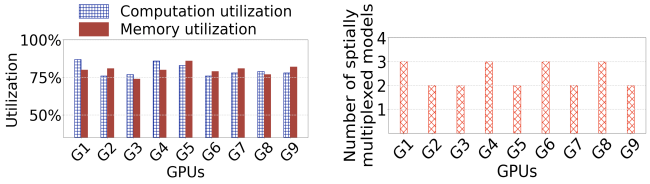
We then experimented to test the impact of workload division. We first used Shepherd to decide the BS (b_i^s) and the minimum RD (r_i^d) of each model i to complete its workload within the SLO. Then, for each model i independently, we created $2r_i^d$ replicas of each model with BS=$\frac{b_i^s}{2}$ and placed them to randomly selected GPUs with enough available C-spaces and M-spaces to host the model replicas. In our next experiment, to perform holistic workload division, we created all possible configurations, i.e., {(BS, RD) for each model i within range $[b_i^s, \frac{b_i^s}{2}]$ and $[r_i^d, 2r_i^d]$}. Then, for each configuration, we placed the model replicas to randomly selected GPUs having enough resources. Finally, we chose the best configuration that resulted in the lowest GPU#. Fig. 4 shows the average Cuti and Muti, and model# in each GPU for both experiments. Independent workload division increases the average Cuti and Muti by 3.5% and 3.8%, respectively, compared to Shepherd shown in Fig. 1 and the GPU# is decreased from 12 to 11. The holistic workload division further increases the average Cuti and Muti by 4.7% and 5.1%, respectively, leading to another decrease of GPU# by 1, even with a simple strategy of random GPU placement.

Observation 2. *In spatial multiplexing-based inference serving, unlike existing systems, we may need to divide the workload of a model even when one GPU is enough to complete the workload within SLO in order to increase the overall resource utilization.*

Observation 3. *Optimal workload division should not be decided independently for each model. Instead, a holistic approach that considers all models simultaneously is essential.*

2.3 Study on C-heavy and M-heavy Models

In this experiment, we did the same holistic workload division experiment described above, except that, for each configuration, we first ordered the models in descending order of Creq+Mreq and followed this order of models to place the model replicas to randomly selected GPUs with enough spare resources. Fig. 5 shows the results. The average Cuti and Muti increase by 5.3% and 4.9%, respectively, compared to the holistic approach in Fig. 4a, leading to a further decrease of GPU# by 1. Also, the average model# in a GPU increases by 0.24 compared to the holistic approach in Fig. 4b. Due to the ordering of the models, less spare resources remain in the GPUs after the model placement, leading to less resource fragmentation and better utilization.



(a) Resource utilization of each GPU by sorting the models according to total resource requirements.

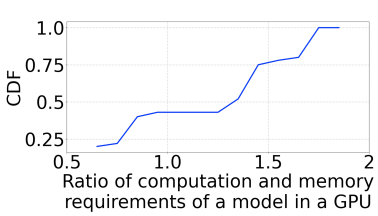
(b) Number of spatially multiplexed models in each GPU using the sorting approach.

Figure 5: Considering a model's total resource requirement in model replica placement.

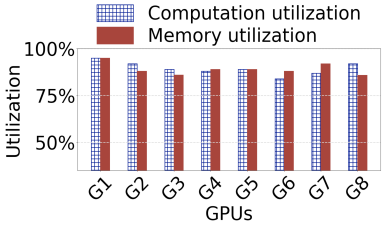
The general notion assumes large models have high Creq and Mreq, while small models have low Creq and Mreq. However, this distinction overlooks the impact of BS in inference serving. Increasing BS may boost resource use but risks latency violations and memory overflow. This highlights the delicate balance of Cuti and Muti in workload scheduling.

For example, when we executed LlaMA-2 (with 13 billion parameters) with BS=4 in a Nvidia H100 GPU, it takes up almost all GPU memory, but has 45% Cuti unused. Increasing BS any further overflows the memory. Hence, it is M-heavy instead of C-heavy, despite being a large model. On the other hand, MobileNetV2 (only 3.4 million parameters) with BS=128 reaches up to 93% Cuti but has 30% Muti. Increasing BS any further violates its 64ms SLO. Hence, it is a C-heavy model instead of M-heavy model.

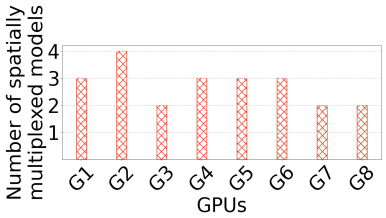
Fig. 6a shows the CDF of models versus the ratio $Creq/Mreq$ of a model. We see that 22% of the models have ratios $\leq$0.75, indicating they are M-heavy. Also, 28% of the models have ratios in (1.35,1.65], indicating they are C-heavy. Llama-2 is an M-heavy model (i.e., Mreq/Creq$\geq$1.2). Among the other small models, 39% are M-heavy, 2% have comparable Creq and Mreq, and the rest are C-heavy (i.e., Creq/Mreq$\geq$1.2).



(a) CDF of models vs. Creq/Mreq.



(b) Resource utilization by multiplexing C-heavy and M-heavy models.



(c) Model# by multiplexing C-heavy and M-heavy models.

Figure 6: Computation-heavy and memory-heavy models.

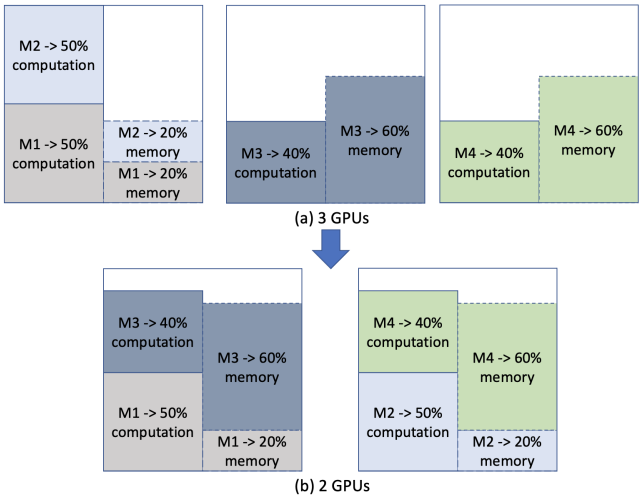
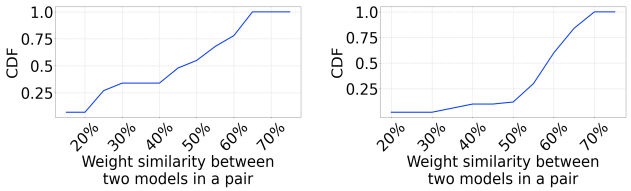


Figure 7: Spatially multiplexing a computation-heavy model with a memory-heavy model increases resource utilization.

Observation 4. *Unlike common belief, model parameter size alone cannot dictate whether a model is C-heavy or M-heavy. Even a small model can surpass a larger one in Creq, driven by the intricate relationship between BS, BS-dependent resource needs, and SLO.*

We next investigate whether multiplexing a C-heavy model with an M-heavy model improves Ruti. Let us consider 4 models M1, M2, M3, and M4 in Fig. 7 with BS=32. Their Creq are 50%, 20%, 40%, and 40%, and their Mreq are 20%, 20%, 60%, and 60% respectively. Hence, M1 and M2 are C-heavy and M3 and M4 are M-heavy. First, we multiplex M1 and M2 in the same GPU as shown in Fig. 7a, which results in 100% Cuti but 40% Muti. M3 and M4 cannot be multiplexed due to lack of memory, resulting in 3 GPU#, 60% average Cuti and 53.33% average Muti. Alternatively, if we multiplex M1 with M3 and M2 with M4 as shown in Fig. 7b, it results in 2 GPU#, 90% average Cuti and 80% average Muti. Hence, multiplexing a C-heavy model with a M-heavy model maximizes resource utilization.

To experimentally validate this, we did the same experiment described in §2.3, except that we interleave C-heavy models with M-heavy models in the ordered list. Figs. 6b and 6c show the results. The average Cuti and Muti increase by 12.1% and 11.8%, respectively, compared to the holistic approach in Fig. 4a, Also, the average model# increases by 0.6 compared to the holistic approach in Fig. 4b, and hence, GPU# is decreased by 2.



(a) Weight similarity across CNN models. (b) Weight similarity across Transformer models.

Figure 8: Weight similarity across models.

Observation 5. *Multiplexing C-heavy model with an M-heavy model increases both Cuti and Muti of a GPU.*

2.4 Models' Weight Overlapping

Previous studies [48, 49] have indicated that CNN models have significant weight overlapping (i.e., similar parameter values) in earlier layers because the first few convolutional layers function as feature-extractors [48] and are task-agnostic. Also, to reduce training cost, the task-specific transformer models are typically trained not from scratch, but from pretrained task-agnostic foundation models using transfer learning. Additionally, for both models, people generally customize a task-specific pretrained model for their own datasets by fine-tuning only the last few layers. These indicate the potential for weight similarity across models. Motivated by these, we evaluated the weight similarity across models. Specifically, for each pair of CNN models and Transformer models, we measured the weight similarity between the two models in the pair.

Given CNN models A and B, we first find the weight similarity between every possible pair of convolutional layers across the two models. Then, we take the average weight similarity of all pairs of layers as the weight similarity between the two models. To find the weight similarity between layer $i \in L_A$ and layer $j \in L_B$, where L_x denotes the set of all convolutional layers in model x, we calculated $\frac{2\times|\max(W_A^i \cap W_B^j)|}{|W_A^i|+|W_B^j|}$, where $\max(W_A^i \cap W_B^j)$ denotes the longest common submatrix between weight matrices W_A^i and W_B^j. We consider two weight values as the same if their absolute difference is very low (i.e.,

$\leq 10^{-7}$). We did the same experiment for the attention layers of the Transformer models.

Fig. 8 shows the results for randomly chosen 1k CNN and 1k Transformer models from HuggingFace trained model repository [45]. For the CNN models, 52% model pairs have weight similarity within $(45\%, 65\%]$. For the Transformer models, 70% model pairs have weight similarity within $(55\%, 70\%]$ in between themselves.

Observation 6. *There are significant weight overlappings between different CNN models, and also between different Transformer models.*

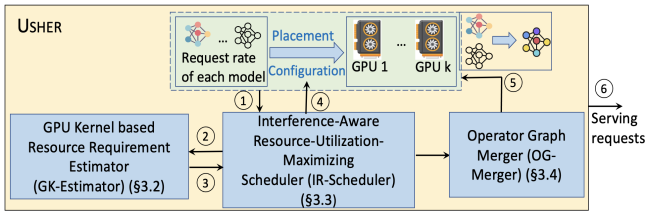


Figure 9: System overview of USHER.

3 SYSTEM DESIGN OF USHER

3.1 Overview

Observation(O)1 motivates us to propose a new system to maximize both computation and memory utilizations of GPUs in an interference-aware manner to minimize the inference serving cost. We design USHER based on O2-O6.

Given models with request rates and a cluster of GPUs (that may be heterogeneous), USHER decides the *schedule* that includes the configuration (BS,RD), GPU allocation, and placement of the model replicas of each model. We consider two scenarios with different goals in this paper: (i) non-fixed cluster, where USHER aims to minimize the monetary cost while satisfying the SLOs of all models [12, 14, 50], and (ii) fixed-cluster, where USHER aims to maximize goodput [3, 4].

USHER has following major methods as shown in Fig. 9.

(1) **GPU Kernel based Resource Requirement Estimator (GK-Estimator)**(§3.2). The estimator quickly and correctly calculates the Creq and Mreq of a model in a GPU type based on a given configuration.

(2) **Interference-Aware Resource-Utilization-Maximizing Scheduler (IR-Scheduler)**(§3.3). Instead of solving an optimization problem, which has high complexity, USHER provides a lightweight heuristic to quickly derive the schedule. It first groups the models in a manner that maximizes the opportunity of multiplexing C-heavy and M-heavy models within a group. Then, within each group, it chooses the configuration that results in the placement with the best performance regarding the specific goal (by leveraging O2-5). The IR-Scheduler ensures each model replica gets its Creq and Mreq in the GPU where it is placed, thus ensuring there is no inter-model interference in C-space and M-space.

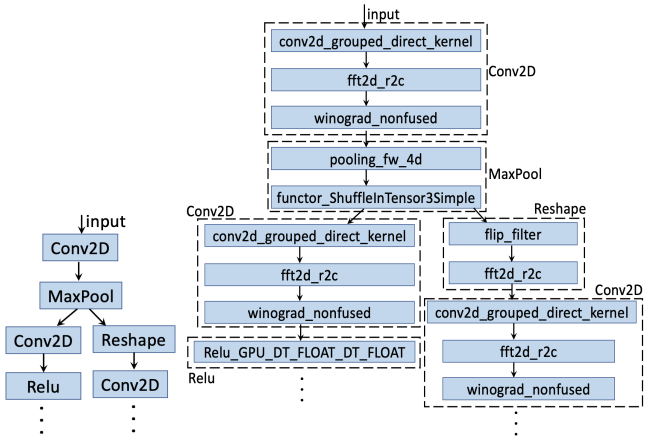
(3) **Operator Graph Merger (OG-Merger)**(§3.4). After deciding the placement, OG-Merger merges as many operator graphs of the models assigned to a GPU as possible to minimize inter-model interference in the GPU cache. After the merging, the merged graph is allocated the sum of the resources allocated to the models (decided by the IR-Scheduler) whose graphs have been merged. Note that the IR-Scheduler does not satisfy the cache requirements of the models, which we found to be almost 100% in the setup in §2. Hence, satisfying the cache requirement results in underutilization of C-space and M-space. That is why USHER addresses cache interference separately in OG-Merger.

In USHER, the input to the IR-Scheduler includes the request rate (i.e., workload) of each model and the types of GPUs in the system (①). During decision making, IR-Scheduler uses GK-Estimator (②) to estimate Creq and Mreq given a configuration (③), and finally outputs the optimal schedule (④). Then, OG-Merger merges the models placed to the same GPU (⑤). Then, the system starts serving the inference requests. As [14, 50], when a model's workload changes significantly, i.e., by 0.5k requests/second, (which may happen after 45s-300s as shown in §5.4), USHER is used again to adapt to the new workload pattern.

3.2 Kernel-based Resource Requirement Estimator

Offline profiling is a common approach for estimating the Creq and Mreq of each new model [3, 4, 12, 14, 50]. However, it is costly and time-consuming. To address this challenge, we propose the GK-Estimator that estimates the resource requirement of each model independently by analyzing its low-level GPU kernels without actually running the model in a GPU. We use Mreq as an example to explain how GK-Estimator works. Every model can be treated as a computational graph, in which a node is an operator and an edge is a tensor (i.e, multi-dimensional matrix) denoting model input or intermediate data generated by a model layer. Internally, each operator execution involves sequentially calling one or more GPU kernel APIs defined in the GPU programming framework (e.g., CUDA for Nvidia GPUs, ROCm for AMD GPUs). For each operator defined in ONNX [51], we found which GPU kernels are called by an ML framework during the execution of the operator by using Nvidia Profiling tool [52]. We noticed that 1, 2, 3, and 4 GPU kernels are called for 2%, 8%, 56%, and 34% of the operators, respectively.

The operators of a new model are usually from a pre-known set of operators [53, 54]. Therefore, GK-Estimator uses a regression model that quickly computes the memory required by the intermediate data generated by an operator based on the sizes of the input tensors and mathematical operations of the operator. Then, for a sequential DL model, its Mreq is the sum of the model parameter size and the highest memory required by an operator. The memory requirement for a model with parallel branches (e.g., Inception model, illustrated in Fig. 10a), the Mreq of the intermediate data is the sum of



(a) Operator-level computation graph **(b) Kernel-level computation graph**

Figure 10: Conversion of an operator-level computation graph for a CNN to its kernel-level computation graph.

Mreqs of the intermediate data from kernels that are executed concurrently. Then, as shown in Fig. 10, first, GK-Estimator converts the operator-level computation graph to a kernel-level computation graph by replacing each operator with the sequence of GPU kernels it calls. This sequence is found offline for each operator in ONNX. Then, it finds each set of kernels that will be executed concurrently. Next, for each set, it estimates the Mreq of the intermediate data generated by each kernel of the set by a regression model (called Mreq-Regressor) and then sums up the Mreqs of all the kernels in the set. Finally, it takes the sum of the model parameter size and the maximum memory requirement from all sets as the Mreq of the model.

The start time of the first kernel of the model (i.e., the kernel that directly gets the model input) is 0^{th}ms. To identify which kernels will be executed concurrently, GK-Estimator first finds the start time of each kernel. It uses another regression model to estimate the execution time duration of each kernel (called Time-Regressor). The kernels with a start time difference of no more than τms (e.g., 0.001ms) are considered to be potentially running concurrently.

To build Mreq-Regressor and Time-Regressor, we use a stacked model (a combination of lasso regression, kernel ridge regression, gradient boost regression, and XGBoost regression models) for higher accuracy [55]. The inputs to both the regressors include the following features that impact the resource requirement and execution time duration of a kernel: batch size, sizes of the kernel's parameter weight tensor and its input intermediate data tensor, the number of floating-point operations of the kernel, and GPU type. Note that none of the features depend on the kernels of the other models that the IR-Scheduler may potentially place to the same GPU. We trained the regression models offline using all the GPU kernels for all the operators defined in ONNX. The training takes 1.3hr in a V100 GPU. We conducted an experiment and found that this kernel analysis approach achieves 99.98% accuracy in estimating the Creq and Mreq of the models in Table 1 with BS 2. We also measured the accuracy of each of the constituting regression models in the stacked model and found that they provide 23%-40% less accuracy compared to the stacked model.

3.3 Interference-aware and Resource Utilization-maximizing Scheduler

USHER first groups the models such that the models inside a group are highly probable to be multiplexed in an interference-aware and resource utilization-maximizing manner. Then, inside each group, based on O3, USHER decides the GPU allocation and placement decisions *holistically* for all models of the group. Below, we first describe the grouping process (§3.3.1) and then the scheduling process (§3.3.2).

3.3.1 Model Grouping

Based on O4 and O5, multiplexing a C-heavy model with an M-heavy model maximizes Ruti. Such multiplexing makes the Cuti and Muti of a GPU comparable. If a GPU's C-space is much lower than its M-space or vice versa, it may not be able to host an additional model. Based on this, we follow one principle during multiplexing models. That is, the sum of the Creqs of the models is nearly equal to the sum of the Mreqs of the models in the GPU (i.e., $\sum_i Creq_i \approx \sum_i Mreq_i$). Based on this, USHER groups the models such that $\sum_i Creq_i \approx \sum_i Mreq_i$ for the models in each group.

Before conducting the grouping, USHER first finds the Creq and Mreq of each model. USHER calculates the average Rreq across all possible BS and GPU type combinations using GK-Estiamtor (described in §3.2). For a GPU type, USHER stops at the BS for which the Creq or Mreq exceeds the maximum C-space or M-space of the type, respectively.

Next, USHER performs the grouping using a variant of k-means clustering [56]. The algorithm clusters a set of elements into nearly equal-sized groups so that the sum of the distances between elements within each group is minimized, while the sum of the distances between groups is maximized.

At the beginning, each group consists of a single model. USHER calculates the distance between every two models as $D = |\sum_i Creq_i - \sum_i Mreq_i|$. Then, it uses the k-means algorithm to group the models into several groups, where each group consists of two models such that D is minimized, i.e., $\sum_i Creq_i \approx \sum_i Mreq_i$ for each group. Next, considering each group as an element, USHER executes another pass of the algorithm. This process merges two groups created by the previous pass to one and increases the number of models in each group by two times. As a result, if we decide to have at most 2^p (i.e., 4) models in each group, we need to perform p passes of the algorithm. Finally, the models are grouped into several groups and the set of the groups is denoted by $\mathbf{G} = \{\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_n\}$.

3.3.2 Scheduling: Deciding Configuration and Placement

After grouping the models, for every model in each model group $\mathcal{G}_i$, IR-Scheduler decides the schedule. Below, we use

Algorithm 1 Interference-aware and resource utilization-maximizing scheduler for **G**.

1: **for** each $\mathcal{G}_i \in$ **G do**
2: Generate all possible `configurations={(BS, RD)` for each model $M \in \mathcal{G}_i$}.
3: **for** each `configuration` **do**:
4: `cost, total_goodput = PLACEMENT(configuration)`
5: Schedule as per the `configuration` for which all of the requests are completed within their latency SLOs, i.e., `total_goodput = total_workload` and the `cost` is minimum.

the non-fixed cluster as an example to present the method and then extend it for the fixed cluster. Based on O2, workload division (i.e., RD>1) may increase the utilization even when one model replica is enough to complete all of the inference requests within the latency SLO. During the scheduling, first, USHER takes all possible configurations, i.e., {(BS, RD) for each model in the group $\mathcal{G}_i$} (Algorithm 1). For each configuration, it finds the placement to minimize the cost (Algorithm 2). Finally, USHER chooses the configuration and placement that result in the minimum cost.

The placement algorithm tries to assign one group of models $\mathcal{G}_i$ to the same GPU set to maximize Cuti and Muti. Therefore, to assign the models in $\mathcal{G}_i$, the algorithm prioritizes the GPUs that are already assigned with the models in $\mathcal{G}_i$, and initializes a new GPU only when no used GPU can host a model. To avoid resource fragmentation and increase Ruti, it prioritizes the models $\mathcal{G}_i$ that have high Creq and Mreq and aims to place it to a GPU that leaves the lowest C-space and M-space after hosting it. Additionally, based on O5, USHER takes placement decisions alternatively between C-heavy and M-heavy models. The scheduling algorithm is shown in Algorithm 1. The algorithm finds the best placement decision for each possible configuration by calling the `PLACEMENT()` algorithm (Lines 1-4). The set of possible values of BS of a model is: {4, 8, 16, 32, 64, 128}, and the set of possible values of RD of a model is: $\{m \cdot c_M^l\}$, $m = 1, 2, ..., 6$, where c_M^l is the minimum possible value of RD of model M to satisfy its SLO. It is calculated as the minimum number of GPUs of the highest GPU type required to complete all of the model M's requests within the SLO considering the highest possible BS in each GPU. The highest possible BS is taken as the minimum value between 128 and the BS beyond which its Mreq exceeds the memory capacity ($\max M$) of the highest capacity GPU type. This way, as the maximum possible value of RD of a model can be as much as 6 times the minimum possible value, the scheduling algorithm conducts workload division based on O2.

Algorithm 2 Placement algorithm for model group $\mathcal{G}_i$.

1: **procedure** `PLACEMENT (configuration)`
2: $\mathcal{G}_{iGPU} \leftarrow$ GPU group for $\mathcal{G}_i$, initially empty
3: **for** each $M \in \mathcal{G}_i$ **do**
4: Calculate its $Creq$ and $Mreq$ in each type of GPU
5: **if** $Creq > \max C$ or $Mreq > \max M$ (highest-capacity GPU) **then**
6: **return** `Infeasible_configuration`
7: Group the models into C-heavy and M-heavy models
8: Sort two groups in descending order of $Creq + Mreq$: $\{M_1, M_2, \ldots, M_n\}$ and $\{M'_1, M'_2, \ldots, M'_m\}$
9: `final_model_list` $\leftarrow \{(M_1, M'_1), (M_2, M'_2), \ldots, (M_n, M'_m)\}$
10: **for** each $M \in$ `final_model_list` **do**
11: `MODEL_REPLICA_PLACEMENT_WITHIN_`$\mathcal{G}_{iGPU}$`()`
12: `MODEL_REPLICA_PLACEMENT_OUTSIDE_`$\mathcal{G}_{iGPU}$`()`
13: **for** each model replica of M **do**
14: `NEW_LOWEST_COST_GPU_INITIALIZATION()`
15: Assign the new GPU to $\mathcal{G}_{iGPU}$.
16: **for** each $M \in \mathcal{G}_i$ **do**
17: `goodput`$_M$ = `min (achieved_goodput`$_M$, `workload`$_M$`)`
18: `total_goodput` = $\sum_M$`goodput`$_M$
19: **return** additional `costs` for initializing new GPUs and `total_goodput` for the taken placement decision.

The placement algorithm is shown in Algorithm 2. USHER first calculates the Creq and Mreq for each model in each GPU type according to the configuration given as input using GK-Estimator (Lines 3-4). Then it further groups the models in the model group $\mathcal{G}_i$ into C-heavy and M-heavy models (Line 7). A model is C-heavy if its average C-req/M-req ≥ 1.2, and is M-heavy if M-req/C-req ≥ 1.2. Next, USHER sorts each of the two sub-groups in the descending order of the Creq+Mreq (Line 8). After that, USHER pairs up each two models from the two sub-groups to create `final_model_list` (Line 9). Finally, USHER inserts the models that are neither C-heavy nor M-heavy into the list while maintaining the descending order of Creq+Mreq.

Then, USHER picks up a pair or a model one by one from the list to be assigned to a GPU. Specifically, USHER calls the `MODEL_REPLICA_PLACEMENT_WITHIN_`$\mathcal{G}_{iGPU}$ function (Line 11). The function places as many model replicas of M as possible to the mode's GPU group (denoted by $\mathcal{G}_{iGPU}$). The GPU group of a model group is defined as the group of GPUs that hosts most of the model replicas of the model group. Basically, when USHER initializes a new GPU for any model replica of model group $\mathcal{G}_i$, the new GPU is added to the GPU group $\mathcal{G}_{iGPU}$. This way, USHER tries to place the model replicas in the same model group to the GPUs of the same GPU group. If multiple GPUs are available for a model or a pair, USHER chooses the one that leaves the lowest C-space+M-space after hosting it to avoid resource fragmentation.

After that, USHER calls `MODEL_REPLICA_PLACEMENT_OUTSIDE_`G_{iGPU} that places as many remaining model replicas of M as possible to the GPUs of the other GPU groups (Line 12). Finally, for each model replica that is not placed to any GPU yet, USHER calls `NEW_LOWEST_COST_GPU_INITIALIZATION` that initializes a new GPU of the GPU type that can host the model or pair with the minimum cost and assigns the GPU to G_{iGPU} (Lines 13-15). At last, the placement algorithm returns to the scheduling algorithm the additional costs for initializing the new GPUs and the total goodput across all the models (Line 19). As [14], goodput of a model is taken as the ratio between the batch size and the expected time (including in-queue wait time) to

complete a batch. The execution time of a batch is calculated using the Time-Regressor in §3.2.

For the fixed cluster setup, the modifications are as follows. First, the maximum value of model replication degree is capped by the total number of GPUs in the cluster. Second, in Algorithm 1, USHER schedules as per the configuration for which the `total_goodput` is maximum.

3.4 Operator Graph Merging to Minimize Cache Interference

To minimize interference in GPU cache, among the models assigned to one GPU, USHER merges as many operator graphs of the models as possible into a single graph. To perform maximal operator graph merging, USHER first groups the models to sub-groups based on their architectural similarity (i.e., the structure and the constituting operators) (§3.4.1). Then, for each sub-group, motivated by O6, USHER decides which operators across multiple graphs to merge based on their weight similarity (§3.4.2). Finally, during the merging process, USHER extracts away the largest common weight submatrix between the operators that need to be merged and ensures that the matrix multiplications associated with the submatrix for different inputs of different models are processed at the same time, while the submatrix is in the GPU cache (§3.4.3).

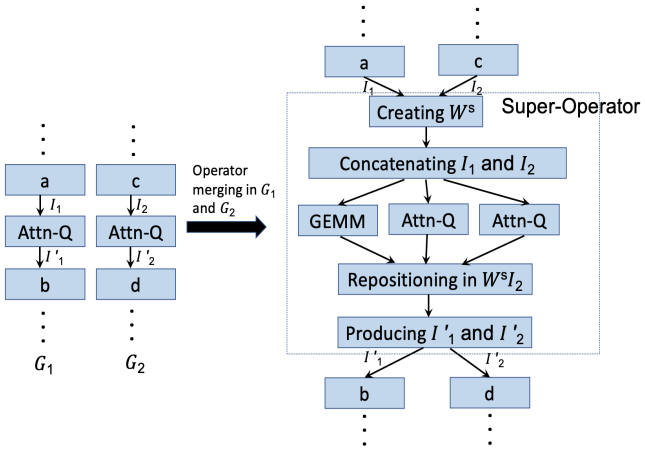
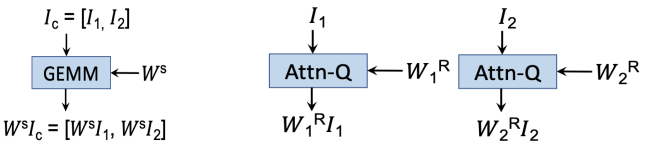


Figure 11: Operator merging in G_1 and G_2 to maximize GPU cache usage. Attn-Q refers to the Attention Query operator.



(a) Executing on W^s at the same time.

(b) Executing the remaining operations using separate operators.

Figure 12: Creating a new operator GEMM.

3.4.1 Grouping Architectural-Similar Operator Graphs

The models with architectural-similar operator graphs are more likely to have high similarity in their weights [54]. Based on this, USHER uses DBSCAN algorithm [57] to group the models assigned to a GPU based on architectural similarity. Given a set of elements, DBSCAN algorithm can cluster the elements with very little distance (i.e., 10^{-7}) between themselves within the same group, without requiring any predetermined number of groups or number of elements within a group. Inside DBSCAN algorithm, USHER uses graph edit distance [58] to calculate the distance to measure the architectural similarity between two operator graphs. Basically, the edit distance algorithm finds how many addition/deletion/replacement of operators need to be performed to make the two operator graphs identical. A shorter distance means higher architectural similarity.

3.4.2 Graph Matching in an Architectural-Similar Group

Among the models in an architectural-similar group. USHER first randomly takes two models. Then, it generates a bipartite graph $\mathcal{B}$ containing the operators of both models. An edge e exists between two vertices from the two graphs if and only if all of the following conditions are satisfied: (i) same type (i.e., either convolutional operator or attention operator), (ii) same starting time (explained in §3.2), (iii) the weight similarity between the operators (described in §2.4) is no less than ω (e.g., 40%). It is assigned as the edge weight.

After generating $\mathcal{B}$, USHER finds the maximal weighted matching using Hungarian algorithm [59] in $\mathcal{B}$, which chooses a set of independent edges (i.e., that do not share any common vertex) such that the sum of weights is maximized. The two endpoint operators of each of the chosen edges are matched and will be merged (explained in §3.4.2). Then, USHER randomly takes another model from the remaining models and generates a new bipartite graph $\mathcal{B}'$ using $\mathcal{B}$ and the other model by repeating the same procedure. This process repeats until no more models in the group can be merged.

3.4.3 Operator Merging Process

As an example of operator merging, we describe the process for two *Query* operators in the attention layers of two Transformer models. It is a matrix multiplication operation: $I'_1 = W_1 I_1$, where I_1 is the input and W_1 is the Query weight matrix. Now, we explain how USHER modifies this operation for operator merging. If I_1 and I_2, as well as W_1 and W_2, do not have the same size, we apply zero padding to make them the same size.

Fig. 11 shows the overall process of merging two Query operators of two graphs. USHER creates a Super-Operator to merge operators, which consists of several individual operators, each of which performs a specific task as described below. First, it extracts out the largest common submatrix between W_1 and W_2, denoted by A_{W_1,W_2} using template matching [60]. It creates a matrix W^s, which contains A_{W_1,W_2} at the same position as W_1 (or W_2), and zeros in other entries. It also creates another matrix W_1^R, which is the same as W_1, except that the entries associated with A_{W_1,W_2} are all zero, and creates matrix W_2^R from W_2 similarly. Second, as illustrated in Fig. 12a, USHER concatenates the inputs I_1 and I_2 into a matrix as

$I_c = [I_1, I_2]$ and creates a new general matrix multiplication (GEMM) operator that performs $W^s I_c = [W^s I_1, W^s I_2]$. This way, the new GEMM operator executes the matrix operations of the original two operators corresponding to A_{W_1,W_2} at the same time, while W^s is still loaded in the GPU cache, thus maximizing its usage.

Also, as illustrated in Fig. 12b, for each of the original operators, USHER creates a Query operator to perform the remaining matrix multiplication operations that are not associated with W^s: $W_1^R I_1$ and $W_2^R I_2$. Third, I'_1 can be calculated simply as $I'_1 = W^s I_1 + W_1^R I_1$. However, the entries in $W^s I_2$ need to be repositioned. This is because A_{W_1,W_2} is positioned in W^s according to its position in W_1. This repositioned $W^s I_2$ would be generated if A_{W_1,W_2} were positioned in W^s according to its position in W_2. Finally, after the repositioning, I'_2 is calculated as $I'_2 = W^s I_2 + W_2^R I_2$.

While merging a Super-Operator with another Query operator O_q, USHER merges each Query operator inside the Super-Operator with O_q.

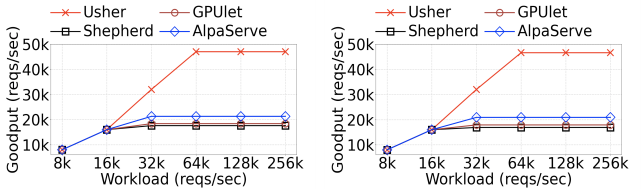
4 Implementation Details

We developed USHER using Python and the implementation is available at [61]. We used Tensorflow for the inference executions of the models, but note that our techniques are general and can be incorporated on other serving platforms. After the XLA operator graph optimization [62] is performed on the operator graph in TensorFlow, we converted it into the framework-independent ONNX format to ensure USHER works for other ML frameworks (e.g., PyTorch, MXNet) as well. We found the GPU kernels called by an ML framework during the execution of an operator by profiling the operator using Nvidia Nsight [52] with the `print-gpu-trace` option turned on. We used `achieved_occupancy` and `dram_utilization` options in Nsight to measure the Creq and Mreq of a kernel, respectively. In each GPU, we used Nvidia MPS [13] to divide the GPU computation resource among the models based on their requirements. After merging the operator graphs in ONNX format, we applied the TVM optimization [63] from Nvidia TensorRT [64] to further optimize the merged graph and executed the merged graph as a single `CUDA Context` in MPS, which was allocated the sum of `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`s assigned to the models whose operator graphs were merged.

5 Performance Evaluation

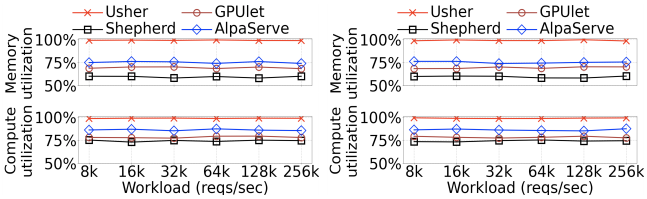
5.1 Experimental Setup

Unless otherwise specified, the experiment settings are the same as those in §2. In addition to the models described in Table 1, we also used two multi-model applications that include multiple DL models [2, 50]: video surveillance (SLO: 500ms) [2] and social media (SLO: 750ms) [65]. In addition to the Microsoft Azure Function trace 2019 (MAF1) with steady and dense request arrival rates, we also experimented with MAF trace 2021 (MAF2) with bursty arrival rates [66].



(a) MAF1 (b) MAF2

Figure 13: Goodput comparison of different methods in real testbed for a fixed cluster.



(a) MAF1 (b) MAF2

Figure 14: GPU computation and memory utilization comparison of different methods in real testbed for a fixed cluster.

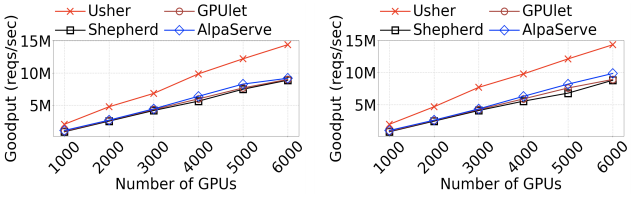
Testbed. We conducted both real testbed and simulation experiments. The real testbed is a cluster with 6 AWS EC2 p3.8xlarge servers, each of which consists of 4 V100 Nvidia GPUs and the GPUs are inter-connected via NVLinks. In the simulation, we increased the number of GPUs up to 6000 to simulate a large enterprise-grade GPU cluster [67]. As it is prohibitively expensive to actually execute the models using these many GPUs, we report the result directly from the scheduler decision, without actually running the models. In simulation, we experimented with tremendously large workloads reaching up to 15M requests/second to simulate the large workloads in enterprise-grade clusters. We tested for both fixed and non-fixed cluster setups. We compared USHER with Shepherd [3], GPUlet [14], and AlpaServe [4].

5.2 Comparison Results

Our key results include: USHER (i) achieves up to 2.6× higher goodput in a fixed cluster and (ii) requires up to 3.5× lower cost in a non-fixed cluster.

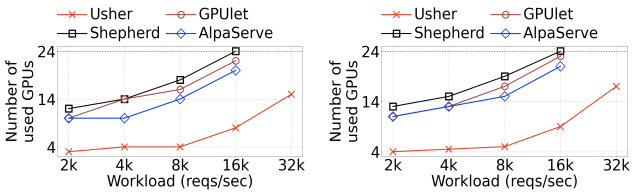
5.2.1 *Fixed Cluster*

Fig. 13 shows the average goodput in the real testbed with varying average workloads (averaged across the total duration of 2 weeks for a trace). Fig. 14 shows the average Cuti and Muti per GPU. USHER achieves 1×-2.6× higher goodput, 22%-24.2% higher Cuti and 38.9%-40.1% higher Muti compared to Shepherd. This is because USHER performs



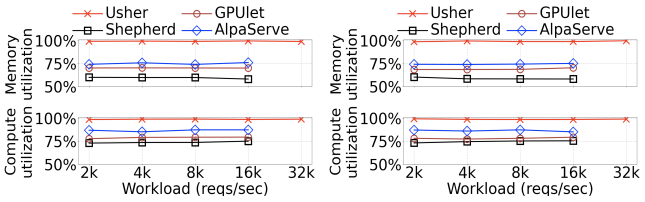
(a) MAF1 (b) MAF2

Figure 15: Goodput comparison of different methods in simulation for a fixed cluster.



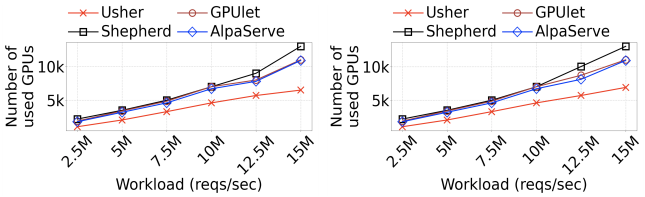
(a) MAF1 (b) MAF2

Figure 16: Number of used GPUs in real testbed for a homogeneous non-fixed cluster. The maximum number of GPUs was 24.



(a) MAF1 (b) MAF2

Figure 17: GPU computation and memory utilization of different methods in real testbed for a homogeneous non-fixed cluster.



(a) MAF1 (b) MAF2

Figure 18: Number of used GPUs of different methods in simulation for a homogeneous non-fixed cluster.

model multiplexing in each GPU in an interference-aware manner, whereas Shepherd allows only one model in each GPU. We observed that the models in each of the following sets were multiplexed in USHER for at least half the total duration of the experiment: {YOLO-v3, ResNeXt-101, Inception-v4, GNMT, BERT}, {ResNet-101, EfficientNet-B7, BERT, GPT-2}, {SqueezeNet, Llama-2}, and {R-CNN, ShuffleNet-v2, GNMT, GPT-2}.

USHER achieves 1×-2.2× higher goodput, 19%-23% higher Cuti and 25.1%-32.2% higher Muti compared to GPUlet and AlpaServe. GPUlet and AlpaServe only try to optimize the GPU computation use, while USHER addresses the interference between models not only in the computation space but also in the memory and cache spaces. Also, USHER has several strategies in its IR-scheduler such as multiplexing C-heavy models with M-heavy models and holistic workload division to maximize the utilizations of both computation and memory spaces. USHER performs consistently for both traces, indicating its resilience to different request arrival patterns. The minimal rescheduling overhead of USHER (i.e., 0.51s from Table 2) enables it to quickly adapt to the bursty workload of MAF2. Nonetheless, if the workload exhibits very frequent burstiness, then the benefits of USHER may be reduced. However, such extreme workloads are uncommon in real-world settings [3,4] (e.g., the production workload illustrated in §5.4, where burstiness occurs after every 45s-300s) and thus the rescheduling overhead of USHER is reasonable enough to not have any adverse impact on its performance.

Goodput of USHER becomes stable at around 47k reqs/s, whereas Shepherd, GPUlet, AlpaServe become stable at much less goodput values. At this point, the method has used up all the GPU resources in the fixed cluster. Fig. 14 establishes that merely increasing the workload cannot saturate the GPUs. This is because it is the batch size that mainly determines the resource consumption of a model and a system cannot choose a larger batch size that leads to a latency exceeding the SLO. To cope up with the increased workload, a system scales out, i.e., increases the number of used GPUs.

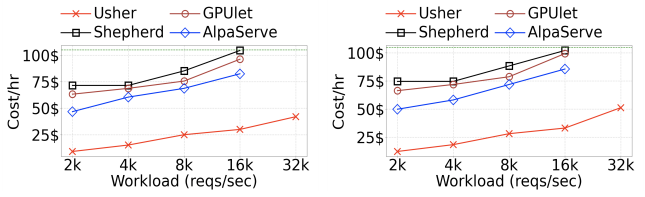
Fig. 15 shows the goodput in simulation with varying number of V100 GPUs. USHER achieves 1.5×-1.9× higher goodput compared to the comparison methods due to the same reasons described above. With the increase in the number of GPUs, the goodput of USHER increases 1.6×-2.1× faster than other methods, indicating the higher scalability of USHER.

5.2.2 *Non-fixed Cluster*

For the non-fixed cluster, we first considered homogeneous V100 GPUs. Then, the cost is proportional to the number of used GPUs. We report the average number of GPUs required by a method at a second. Next, we considered heterogeneous GPUs: K80, V100, A100, and H100. The GPU types have varying costs following AWS on-demand pricing [68].

Homogeneous GPUs. Fig. 16 shows the number of used GPUs in a homogeneous real testbed to complete all inference requests within their SLOs. Fig. 17 shows the average Cuti and Muti per GPU. USHER requires 2.5×-3× fewer GPUs, and achieves 19.3%-24.4% and 24.9%-40% higher Cuti and Muti, respectively, compared to Shepherd, GPUlet, and AlpaServe. As the comparison methods either do not employ multiplexing or suffer from increased latency due to inter-model interference, they require more GPUs.

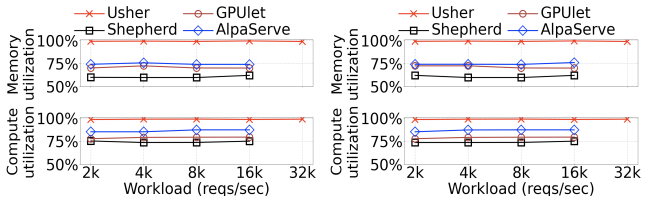
Fig. 18 shows the number of used GPUs in simulation. USHER uses 1.7×-2.1× fewer GPUs due to the same reasons explained above. In both real testbed and simulation, with the increase of the workloads, the increase rate in the number of GPUs of USHER is slower than other methods especially when the workload is very high. These results show the cost-effectiveness of USHER even for large workloads.



(a) MAF1 (b) MAF2

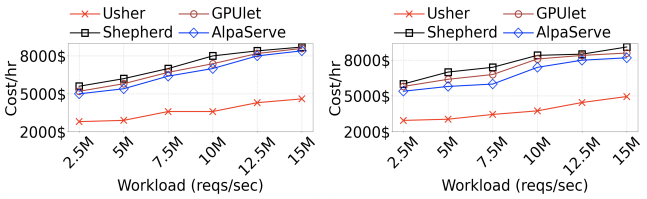
Figure 19: Cost comparison in real testbed for a heterogeneous non-fixed cluster. The maximum cost/hr was 105$.

Heterogeneous GPUs. Fig. 19 shows the cost per hour with varying workloads in a heterogeneous real testbed. Fig. 20



(a) MAF1 (b) MAF2

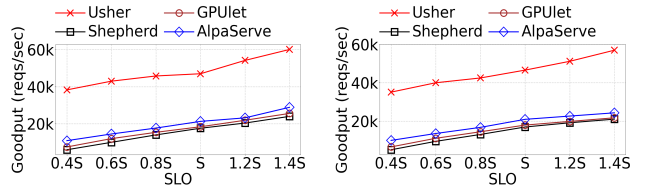
Figure 20: GPU computation and memory utilization comparison in real testbed for a heterogeneous non-fixed cluster.



(a) MAF1 (b) MAF2

Figure 21: Cost comparison of different methods in simulation for a heterogeneous non-fixed cluster.

shows the average Cuti and Muti per GPU. USHER requires 2.8×-3.5× lower cost, and achieves 19%-24.1% and 25.2%-40.3% higher Cuti and Muti, respectively, due to the same reasons explained above. USHER performs slightly better for heterogeneous GPUs compared to homogeneous GPUs as USHER can choose the optimal GPUs from varying GPU types depending on their cost-performance trade-offs.

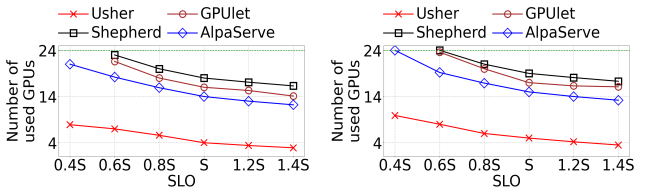


(a) MAF1 (b) MAF2

Figure 22: Goodput comparison with varying SLO values in real testbed for a fixed cluster. *S* denotes the default SLO (Table 1).

Fig. 21 shows the cost per hour in simulation. USHER incurs 1.9×-2.2× lower cost for the same reasons described above. With the increase in workload, its cost increases slower than other methods, especially when the workload is very high.

Overheads. Table 2 presents the average time overhead and average impact on accuracy of different methods. USHER's grouping of models takes only 0.1s and it needs to be updated only when there is addition or deletion of models in the system. The decision-making times of the scheduling in different methods are comparable. This is because all the methods employ time-efficient heuristics to accelerate the scheduling process in order to excel in autoscalability when workload changes (§5.4). USHER's operator graph merging takes slightly more time than its scheduling. Hence, after the scheduling, the requests are executed using individual model graphs. After the graphs are merged, the requests are then executed on the merged graph. For the models whose GPU



(a) MAF1 (b) MAF2

Figure 23: Number of used GPUs with varying SLO values in real testbed for a homogeneous non-fixed cluster. The maximum number of GPUs was 24. *S* denotes the default SLO (Table 1).

Table 2: Overheads of the methods.

Methods	Grouping of models	Scheduling algorithm		Operator graph merging based on weight similarity	Impact on model accuracy due to operator graph merging
		Decision making	Model loading		
USHER	0.1s	0.51s	0.63s	2.3s	-0.0003%
Shepherd	0	0.5s	0.62s	0	0
GPUlet	0	0.46s	0.64s	0	0
AlpaServe	0	0.63s	0.65s	0	0

placement has changed during scheduling, each method takes 0.69s-0.72s to load the models from host memory to GPU memory. Table 3 presents the average model loading time of USHER for each model shown in Table 1. Rescheduling does not happen so frequently [3,4], e.g., after every 45s-300s from Fig. 24. As a result, considering the scheduling time required in USHER, there is enough time left before another scheduling occurs to realize the benefits of USHER. The accuracy loss is only 0.0003% per request batch of a model due to the operator graph merging. This is because, while finding the longest common submatrix in the merging process, USHER takes two weight values as the same only when their absolute difference is very low (i.e., $\leq 10^{-7}$) (§2.4). Table 3 shows the average accuracy loss per request batch for each model in USHER.

5.3 Performance on Varying SLOs

5.3.1 *Fixed Cluster*

Fig. 22 shows the goodput with varying SLO values in the same real testbed setup of Fig. 13 with workload=256k reqs/sec. In the figure, *S* denotes the default SLO of each model as defined in §2, and $m_f S$ denotes that the SLO is multiplied by $m_f \in \{0.4, 0.6, 0.8, 1, 1.2, 1.4\}$. USHER achieves 2.2×-2.7× higher goodput than the comparison methods for the same reason described in §5.2.1. As SLO decreases, goodput also decreases for each method. This is because, in the fixed GPU resources of the cluster, more requests miss their SLO deadlines as the SLO becomes stricter. The result shows that USHER retains its higher goodput compared to the existing methods even when the SLO is ultra-low.

5.3.2 *Non-fixed Cluster*

Fig. 23 shows the number of used GPUs with varying SLO values in the same real testbed setup of Fig. 16 to complete all inference requests within their SLOs with workload=8k reqs/sec. The result shows that USHER achieves 3.2×-4.3× fewer GPUs than the comparison methods for the same reason

Table 3: Model loading and accuracy overheads in USHER.

Model name	Model loading	Impact on model accuracy due to operator graph merging
YOLO-v3	0.44s	-0.0002%
R-CNN	0.57s	-0.0003%
MobileNetSSD-v2	0.495s	-0.0003%
ResNet-50	0.51s	-0.0002%
ResNet-101	0.586s	-0.0004%
ResNeXt-50	0.531s	-0.0004%
ResNeXt-101	0.7s	-0.0003%
SqueezeNet	0.31s	-0.0001%
ShuffleNet-v2	0.39s	-0.0002%
MobileNet-v2	0.395s	-0.0002%
DenseNet-121	0.4s	-0.0002%
DenseNet-201	0.49s	-0.0002%
Inception-ResNet-v2	0.6s	-0.0002%
Inception-v3	0.53s	-0.0003%
Inception-v4	0.58s	-0.0003%
EfficientNet-B7	0.64s	-0.0004%
GNMT	0.89s	-0.0002%
BERT	0.78s	-0.0004%
GPT-2	1.18s	-0.0004%
Llama-2	1.59s	-0.0003%

described in §5.2.2. As SLO decreases, the number of used GPUs increases for each method. This is because each method needs to create more replicas of a model to execute more requests in parallel as the SLO becomes stricter. Otherwise, the requests would have to wait longer for the required GPU resource, leading to SLO violation. The result shows that USHER retains its superior cost-efficiency compared to the existing methods even when the SLO is very strict.

5.4 Microanalysis on Autoscalability

To better evaluate the system autoscalability during rescheduling, we measured the number of used GPUs and the in-queue wait time of a request at each second during a randomly chosen window of 1000 seconds. We used the MAF2 trace in the

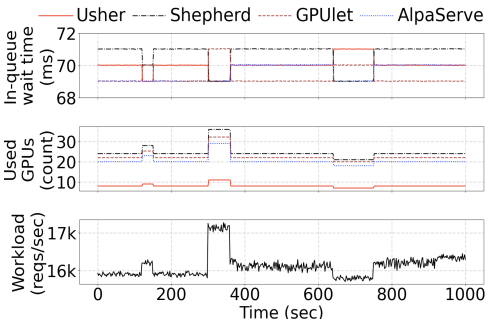
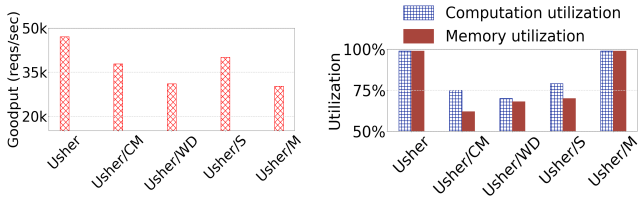


Figure 24: Microanalysis on autoscalability.

homogeneous non-fixed cluster real testbed. Fig. 24 shows the

Table 4: Performance of USHER's GK-Estimator.

Methods	Accuracy of computation requirement calculation	Accuracy of memory requirement calculation	Cost	Time
USHER's GK-Estimator	99.98%	99.98%	**0**	**31.6ms**
Profiling	**100%**	**100%**	42.7$	4.8hr



(a) Goodput **(b) GPU utilization**

Figure 25: Performance of different variants of USHER.

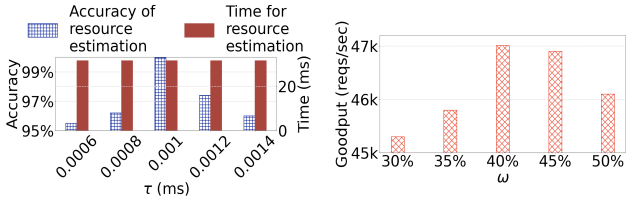
results. Each method increases the number of GPUs when the workload surges up and decreases it when the workload surges down. USHER requires 1.6×-3.9× fewer GPUs compared to the comparison methods, due to its interference-minimizing and resource utilization-maximizing design. The in-queue wait time is almost the same for all the methods and also varies by only 0.8ms-1.4ms as the workload surges up or down. This means that all the methods excel in autoscalability, but USHER requires much fewer GPUs.

5.5 Ablation Study

In this section, we evaluate the effectiveness of each proposed strategy of USHER. We first evaluated the cost- and time-efficiency of USHER's GK-Estimator compared to the existing profiling approaches in estimating the resource requirements of a new model. Table 4 shows the results averaged across the models in Table 1. The GK-Estimator takes 100% less cost and time with comparable accuracy. This is because the profiling approach needs to actually execute the model in the GPUs for different BSs and GPU types, whereas USHER's GK-Estimator only needs to analyze the kernel-level computation graph.

To measure the effectiveness of the other methods in USHER, we created several variants of USHER as follows. 1) USHER/CM skips the step to classify the models in a group to C-heavy and M-heavy. 2) USHER/WD does not conduct workload division between multiple GPUs if one GPU can support the workload to satisfy the SLO. 3) USHER/S does not sort the models based on their computation and memory requirements. 4) USHER/M does not have the OG-Merge.

Fig. 25 shows the goodput and GPU utilization performance of USHER and its different variants in the same setup as Fig. 13a with workload=256k reqs/s. The results show that USHER achieves 24.3%-51.6% higher goodput than USHER/CM, USHER/WD, and USHER/S. This is because, by skipping a method in each of these variants, the Cuti and Muti decrease by 24%-41.4% and 42%-59.6%, respectively, resulting in much lower goodput. USHER achieves 55.7% higher goodput than USHER/M because OG-Merge reduces the interference in GPU cache, resulting in lower latency.



(a) Impact of τ **(b) Impact of ω**

Figure 26: Sensitivity of USHER on its parameters.

5.6 Sensitivity on Parameters

We did the following experiments in the fixed cluster real testbed setup using MAF1 trace with 256k reqs/s workload.

Impact of τ (in §3.2). Fig. 26a shows the accuracy and time for resource estimation of a model with varying values of τ. All τ values lead to the same time for resource estimation. However, we chose $\tau = 0.001$ms as it leads to the highest accuracy of resource estimation since it can correctly capture which GPU kernels will be executed concurrently.

Impact of ω (in §3.4.2) Fig. 26b shows the goodput of USHER for different values of ω. We chose $\omega = 40\%$ as it provides the highest goodput. When $\omega > 40\%$, the number of operators that can be merged is reduced, leading to higher cache interference. When $\omega < 40\%$, more operators can be merged. However, the overhead of additional operations in Fig. 11 outweighs the benefit.

6 Limitations and Discussion

Precision Quantization. The current version of USHER implements FP16 quantization of weights. In the future, we will explore the impact of various precision quantizations (e.g., FP8, FP32) on various factors such as accuracy, interference, and resource utilization and extend USHER to adaptively select the most suitable precision quantization for each model based on the above factors.

Model Parallelism. USHER supports model parallelism out of the box for large models. We assume that model parallelism is enabled by the underlying framework (e.g., DeepSpeed decides how to do model parallelism on Llama-2 in our experiments (§2)), and USHER simply uses it. An interesting future work would be to jointly optimize the model parallelism and placement strategies of USHER to further enhance the resource utilization.

7 Related Work

Inference Serving Systems without Spatial Multiplexing. Many of the systems avoid spatial multiplexing of models within a GPU to prevent inter-model interference [3, 7, 11, 65, 69–81]. Shepherd [3] aggregates request streams into similar-sized groups for high computation utilization and schedules placement to maximize goodput within each group. Several methods [7, 65, 71–76] rely on profiling to find the optimal request batch size for each model, aiming for high GPU utilization and goodput. However, these methods suffer from low resource utilization due to lack of spatial multiplexing (§2.1). Additionally, offline profiling to calculate the resource requirements of a model is time-consuming and costly.

Inference Serving Systems with Spatial Multiplexing. A set of systems adopt spatial multiplexing to enhance GPU utilization while maximizing goodput [1, 2, 4, 12, 14, 21, 23, 50, 82–85]. GPUlet [14] proposes a heuristic for placing models in GPUs to maximize computation space utilization. AlpaServe [4] explores the best placement scheduling by leveraging model parallelism. However, due to inter-model interference in spatial multiplexing, these systems may suffer from longer inference latency (§2.1). Additionally, these systems fail to maximize both GPU computation and memory utilizations (§2.1). Orion [86] is a recent work that maximizes resource utilization by spatially multiplexing the best-effort jobs (e.g., training), while avoiding multiplexing the high-priority jobs (e.g., inference) so that they are not impacted by inter-model interference. However, in our scenario where all the jobs are high-priority inference, Orion will fail to maximize utilization due to the lack of multiplexing. A group of methods [21–23] propose merging layers and sharing parameter weights across multiple models to reduce the memory requirement. However, the merging processes cannot solve the interference in GPU cache as they do not maximize the usage of cache contents.

8 Conclusion

Spatial multiplexing has the potential to increase resource utilization of the GPUs to design a cost-efficient inference serving system. However, it requires careful system design to address the challenges of spatial multiplexing, i.e., maximizing the utilizations of both computation and memory spaces, while minimizing inter-model interference. To this end, we propose USHER. USHER has a lightweight interference-aware scheduler that schedules the models to jointly maximize GPU computation and memory utilizations. During the scheduling, USHER uses a novel lightweight GPU kernel-based estimator to compute the resource requirement of each model. Finally, USHER has a novel operator graph merging approach to minimize interference in GPU cache. Experimental results on both real testbed and large-scale simulations show that USHER achieves up to 2.6× higher goodput and 3.5× better cost-efficiency compared to existing systems.

Acknowledgments

We sincerely thank the anonymous reviewers of OSDI and our shepherd for their invaluable feedback. We are grateful to Kevin Skadron for the discussion on GPU architecture. This research was supported in part by U.S. NSF grants NSF-1827674, NSF-2206522, NSF-1822965, FHWA grant 693JJ31950016, Microsoft Research Faculty Fellowship 8300751, and Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber research, innovation, and workforce development. For more information about CCI, please visit cyberinitiative.org.

REFERENCES

[1] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *Proc. of NSDI*, 2022.

[2] Sudipta Saha Shubha and Haiying Shen. Adainf: Data drift adaptive scheduling for accurate and slo-guaranteed multiple-model inference serving at edge servers. In *Proc. of SIGCOMM*, 2023.

[3] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. {SHEPHERD}: Serving {DNNs} in the wild. In *Proc. of NSDI*, 2023.

[4] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *Proc. of OSDI*, 2023.

[5] Amazon AWS. Inference dominates ML infrastructure cost. https://aws.amazon.com/machine-learning/inferentia/, 2023.

[6] Forbes. Generative AI breaks the data center. https://www.forbes.com/sites/tiriasresearch/2023/05/12/generative-ai-breaks-the-data-center-data-center-infrastructure-and-operating-costs-projected-to-increase-to-over-76-billion-by-2028/, 2023.

[7] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *Proc. of NSDI*, 2022.

[8] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proc. of EuroSys*, 2023.

[9] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proc. of SC*, 2021.

[10] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *Proc. of OSDI*, 2020.

[11] 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 *Proc. of OSDI*, 2020.

[12] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In *Proc. of ATC*, 2021.

[13] Nvidia. Nvidia Multi Process Service (MPS). https://docs.nvidia.com/deploy/mps/index.html, 2021.

[14] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *Proc. of ATC*, 2022.

[15] Nvidia. Nvidia MIG. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html, 2023.

[16] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proc. of SIGCOMM*, 2014.

[17] Sangmin Lee, Rina Panigrahy, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Validating heuristics for virtual machines consolidation. *Microsoft Research, MSR-TR-2011-9*, pages 1–14, 2011.

[18] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. *research.microsoft.com*, 2011.

[19] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean:{VM} allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861, 2020.

[20] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *Proc. of SIGCOMM*, 2022.

[21] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model merging for memory-efficient, real-time video analytics at the edge. In *Proc. of NSDI*, 2023.

[22] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. Retiarii: A deep learning {Exploratory-Training} framework. In *Proc. of OSDI*, 2020.

[23] Angela H Jiang, Daniel L-K Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A Kozuch, Padmanabhan Pillai, David G Andersen, and Gregory R Ganger. Mainstream: Dynamic {Stem-Sharing} for {Multi-Tenant} video processing. In *Proc. of ATC*, 2018.

[24] Peiyuan Jiang, Daji Ergu, Fangyao Liu, Ying Cai, and Bo Ma. A review of yolo algorithm developments. *Procedia Computer Science*, 199, 2022.

[25] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Proc. of ECCV*, 2014.

[26] Ross Girshick. Fast r-cnn. In *Proc. of ICCV*, 2015.

[27] Yu-Chen Chiu, Chi-Yi Tsai, Mind-Da Ruan, Guan-Yu Shen, and Tsu-Tian Lee. Mobilenet-ssdv2: An improved object detection model for embedded systems. In *Proc. of ICSSE*, 2020.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. of CVPR*, 2016.

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

[30] Saifuddin Hitawala. Evaluating resnext model architecture for image classification. *arXiv preprint arXiv:1805.08700*, 2018.

[31] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[32] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proc. of CVPR*, 2018.

[33] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proc. of CVPR*, 2018.

[34] Ke Zhang, Yurong Guo, Xinsheng Wang, Jinsha Yuan, and Qiaolin Ding. Multiple feature reweight densenet for image classification. *IEEE Access*, 7, 2019.

[35] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proc. of AAAI*, 2017.

[36] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proc. of CVPR*, 2016.

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

[38] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[39] WMT. WMT dataset. https://huggingface.co/datasets/wmt19, 2019.

[40] Ce Zhou, Qian Li, Chen Li, Jun Yu, Yixin Liu, Guangjing Wang, Kai Zhang, Cheng Ji, Qiben Yan, Lifang He, et al. A comprehensive survey on pretrained foundation models: A history from bert to chatgpt. *arXiv preprint arXiv:2302.09419*, 2023.

[41] IMDB. IMDB dataset. https://huggingface.co/datasets/imdb, 2011.

[42] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[43] WikiPedia. WikiText dataset. https://huggingface.co/datasets/wikitext/blob/main/README.md, 2016.

[44] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[45] HuggingFace. HuggingFace Models. https://huggingface.co/models, 2023.

[46] DeepSpeed. DeepSpeed Library. https://github.com/microsoft/DeepSpeed, 2023.

[47] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. of ATC*, 2020.

[48] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Proc. of ECCV*, 2014.

[49] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Proc. of NeurIPS*, 2014.

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

[51] ONNX. ONNX Operators. https://github.com/onnx/onnx/blob/main/docs/Operators.md, 2023.

[52] Nvidia. Nvidia Nsight. https://developer.nvidia.com/nsight-systems, 2023.

[53] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proc. of PLDI*, 2021.

[54] Fan Lai, Yinwei Dai, Harsha V Madhyastha, and Mosharaf Chowdhury. {ModelKeeper}: Accelerating {DNN} training via automated training warmup. In *Proc. of NSDI*, 2023.

[55] Saso Džeroski and Bernard Ženko. Is combining classifiers with stacking better than selecting the best one? *Machine learning*, 54, 2004.

[56] Abiodun M Ikotun, Absalom E Ezugwu, Laith Abualigah, Belal Abuhaija, and Jia Heming. K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data. *Information Sciences*, 622, 2023.

[57] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3), 2017.

[58] David B Blumenthal, Nicolas Boria, Johann Gamper, Sébastien Bougleux, and Luc Brun. Comparing heuristics for graph edit distance computation. *The VLDB journal*, 29(1), 2020.

[59] MB Wright. Speeding up the hungarian algorithm. *Computers & Operations Research*, 17(1), 1990.

[60] Simon Korman, Daniel Reichman, Gilad Tsur, and Shai Avidan. Fast-match: Fast affine template matching. In *Proc. of CVPR*, 2013.

[61] USHER. Author code. https://github.com/ss7krd/Usher, 2023.

[62] Tensorflow. Tensorflow XLA Optimization. https://www.tensorflow.org/xla, 2023.

[63] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *Proc. of OSDI*, 2018.

[64] Nvidia. Nvidia TensorRT. https://docs.nvidia.com/tensorrt/index.html, 2023.

[65] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proc. of SoCC*, 2020.

[66] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proc. of SOSP*, 2021.

[67] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent. In *Proc. of ATC*, 2023.

[68] Amazon AWS. AWS on demand pricing. https://aws.amazon.com/ec2/pricing/on-demand/, 2023.

[69] 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 *Proc. of NSDI*, 2022.

[70] 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 *Proc. of SOSP*, 2019.

[71] Zhou Fang, Dezhi Hong, and Rajesh K Gupta. Serving deep neural networks at the cloud edge for vision applications on mobile platforms. In *Proc. of MMSys*, 2019.

[72] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proc. of EuroSys*, 2019.

[73] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *Proc. of NSDI*, 2017.

[74] Vinod Nigade, Ramon Winder, Henri Bal, and Lin Wang. Better never than late: Timely edge video analytics over the air. In *Proc. of SenSys*, 2021.

[75] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proc. of SIGCOMM*, 2018.

[76] Yunseong Kim, Yujeong Choi, and Minsoo Rhu. Paris and elsa: An elastic scheduling algorithm for reconfigurable multi-gpu inference servers. In *Proc. of DAC*, 2022.

[77] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *Proc. of OSDI*, 2022.

[78] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proc. of SOSP*, 2023.

[79] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[80] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

[81] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[82] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. {RECL}: Responsive {Resource-Efficient} continuous learning for video analytics. In *Proc. of NSDI*, 2023.

[83] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proc. of SOSP*, 2023.

[84] Yoonsung Kim, Changhun Oh, Jinwoo Hwang, Wonung Kim, Seongryong Oh, Yubin Lee, Hardik Sharma, Amir Yazdanbakhsh, and Jongse Park. Dacapo: Accelerating continuous learning in autonomous systems for video analytics. *arXiv preprint arXiv:2403.14353*, 2024.

[85] Cyan Subhra Mishra, Jack Sampson, Mahmut Taylan Kandemir, Vijaykrishnan Narayanan, and Chita R Das. Usas: A sustainable continuous-learning´ framework for edge servers. In *Proc. of HPCA*, 2024.

[86] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained GPU sharing for ML applications. In *Proc. of EuroSys*, 2024.