Power-aware Deep Learning Model Serving with *μ*-Serve

Haoran Qiu, Weichao Mao, Archit Patke, and Shengkun Cui, *University of Illinois Urbana-Champaign;* Saurabh Jha, Chen Wang, and Hubertus Franke, *IBM Research;* Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer, *University of Illinois Urbana-Champaign*

https://www.usenix.org/conference/atc24/presentation/qiu

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

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

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

只有 GPU 打不满的时候才有 power-latency 的 tradeoff

Power-aware Deep Learning Model Serving with μ-Serve

Haoran Qiu[1] Weichao Mao[1] Archit Patke[1] Shengkun Cui[1] Saurabh Jha[2] Chen Wang[2]
Hubertus Franke[2] Zbigniew T. Kalbarczyk[1] Tamer Başar[1] Ravishankar K. Iyer[1]

[1]*University of Illinois at Urbana-Champaign* [2]*IBM Research*

Abstract

With the increasing popularity of large deep learning model-serving workloads, there is a pressing need to reduce the energy consumption of a model-serving cluster while maintaining satisfied throughput or model-serving latency requirements. Model multiplexing approaches such as model parallelism, model placement, replication, and batching aim to optimize the model-serving performance. However, they fall short of leveraging the GPU frequency scaling opportunity for power saving. In this paper, we demonstrate (1) the benefits of GPU frequency scaling in power saving for model serving; and (2) the necessity for co-design and optimization of fine-grained model multiplexing and GPU frequency scaling. We explore the co-design space and present a novel power-aware model-serving system, μ-Serve. μ-Serve is a model-serving framework that optimizes the power consumption and model-serving latency/throughput of serving multiple ML models efficiently in a homogeneous GPU cluster. Evaluation results on production workloads show that μ-Serve achieves 1.2–2.6× power saving by dynamic GPU frequency scaling (up to 61% reduction) without SLO attainment violations.

1 Introduction

Over the past few years, increasingly capable large models have been developed for everything from recommendations to text or image generation. Pre-trained deep learning (DL) models make it easy for developers to develop and deploy new models with lightweight fine-tuning, few-shot learning, or even prompting [3]. As a result, model serving (i.e., inference) has become an essential workload in modern cloud systems.

However, serving deep learning models at scale puts a pressing need to improve ***power efficiency***, i.e., to reduce energy consumption while maintaining model-serving performance requirements. A recent study from Google attributed 60% of its ML energy use to inference [29]. AWS estimates inference to make up 90% of total ML cloud computing demand [24]. Unfortunately, existing model-serving systems focus on either performance (i.e., latency and throughput) [7,14,22,26,52,53], model prediction accuracy [8, 15, 35], or LLM optimization [12, 20, 47, 51], but are not power-aware. In this paper, we target ***power efficiency*** in DL model serving.

Challenges. We could potentially leverage GPU Dynamic Voltage and Frequency Scaling (DVFS) techniques that have been widely used during GPU training [5, 6, 42, 50] to save energy. However, dynamic GPU frequency scaling on modern deep learning model-serving workloads poses three main challenges due to their unique characteristics.

[C1] First, it is challenging to determine the optimal GPU frequency that meets performance requirements while maximizing power efficiency. Since low latency in inference is critical, model-serving workloads are typically associated with service-level objectives (SLOs) [22, 52, 53] on end-to-end latency. To meet stringent SLOs, contemporary serving systems either fix the GPU frequency at the default setting or rely on DVFS to scale down frequency based on utilization metrics [27]. However, DVFS is sub-optimal, imprecise, and has non-intuitive implementations in production [27] while overprovisioning GPU frequency leads to high power consumption. The optimal frequency is non-trivial to set for a GPU device that can serve different models or a mix of model partitions from different models [22] when considering stringent latency SLOs, throughput, and costs.

[C2] Second, serving non-deterministic generative models makes it hard to differentiate the negative impact of GPU frequency down-scaling from the inherent non-determinism in execution time. The non-determinism comes from the *autoregressive* nature of generative models. To process each request, one has to run multiple iterations; the output of each iteration is used as input in the following iteration. Worse, a first-come-first-serve (FCFS) request scheduling policy suffers from *head-of-line blocking* issues [30], leading to less power-saving headroom without SLO attainment violations.

既然说到 auto-regressive，为啥不能用 TPOT 来判断 GPU freq down-scaling 的影响?

[C3] Finally, unlike CPUs, GPUs do not support fine-grained (e.g., per-core) frequency scaling. Therefore, if the model or multiple model partitions (from different models) are placed onto a device, the frequency can only be reduced to the maximum frequency demand among all partitions. Power-unaware model partitioning and placement strategies [22] fail to unlock further power saving, even if the optimal frequency mentioned in [C1] can be figured out for each partition.

Our Work. This paper addresses the above challenges by presenting μ-Serve, the first power-aware deep learning model serving framework with fine-grained model provisioning and

GPU frequency scaling. The goal of μ-Serve is to maximize power efficiency while preserving the model-serving performance (i.e., SLO attainment). To achieve the goal, μ-Serve consists of the following main components:

- ***Power-aware fine-grained model provisioning.*** μ-Serve generates a model parallelism plan (i.e., model partitioning) for each model and then places model partitions onto devices in a *power-aware* manner. A model comprises a series of operators (e.g., multiply) over multidimensional tensors. Our key insight is that some operators are more *sensitive* [1] to frequency changes while others are not; those operators that are insensitive to frequency changes often contribute less to the end-to-end latency of serving a request. Based on this insight, we design a novel model provisioning algorithm (§3.1 and §3.2) by considering operator sensitivities to generate model partitions and ensure that each device contains partitions with comparable sensitivities. This algorithm addresses **[C3]** and maximizes power-saving opportunities.
- ***Speculative request serving.*** To address non-deterministic execution when serving generative models (i.e., **[C2]**), we introduce a lightweight *proxy model* that is fine-tuned to predict the execution times of each input query. This is based on our insight that a much smaller model can be quite good at understanding the output length of a large model. Based on the proxy model, we design a speculative shortest-job-first (SSJF) scheduler (§3.3) that improves the performance predictability of serving requests of autoregressive models. SSJF helps achieve higher SLO attainment and, consequently, more power-saving opportunities.
- ***SLO-preserving GPU frequency scaling.*** At runtime, μ-Serve uses a multiplicative-increase-additive-decrease (MIAD) algorithm (§3.4) to exploit the power-saving opportunities and dynamically scale GPU frequency while preserving performance SLOs, which addresses **[C1]**.

Offline Profiling and Online Orchestration. μ-Serve requires an offline phase before running online for power-aware model serving. In the offline phase, μ-Serve first profiles each *primitive operator* [40] to get its *sensitivity score*, indicating how operator execution latency changes with the change in GPU frequency. After obtaining a sensitivity score database, μ-Serve then generates power-aware model partitioning plans by utilizing model parallelism (§3.1) and deploys model partitions based on placement plans (§3.2) that create maximized power-saving opportunities. Such power-saving opportunities are then exploited at runtime by dynamic GPU frequency scaling based on workloads and SLO performance. In the online phase, μ-Serve serves model inference requests with a novel speculative shortest-job-first scheduler (§3.3) that addresses non-deterministic execution patterns of autoregressive models. To reduce power consumption while preserving model-serving performance SLOs, μ-Serve dynamically scales GPU frequency at each device with an MIAD algorithm (§3.4).

[1] We define sensitivity as the change in operator execution latency ($\Delta L > 0$) given the change in GPU frequency ($\Delta F < 0$), i.e., $-\Delta L/\Delta F$.

Results. We evaluate μ-Serve with a diverse set of deep learning models (including traditional non-Transformer models like CNNs, Transformers, and Transformer-based generative models) and production workloads on an 8-node 16-GPU cluster (§4). Evaluation results show that, compared to existing state-of-the-art serving systems, μ-Serve achieves a power reduction factor of 1.9–2.6× at varying rates, 1.5–2.3× at varying stringent SLOs, and 1.2–2× when scaling to higher numbers of devices, without SLO attainment violations.

Contributions. In summary, our main contributions are:

- A novel power-aware ML model serving framework that unlocks power-saving opportunities through fine-grained operator management and dynamic GPU frequency scaling.
- A speculative request scheduler with light proxy models that addresses non-determinism in generative models.
- An open-source implementation of μ-Serve and evaluation on real-world model datasets and production traces.

2 Background and Motivation

We begin with an overview of model-serving systems (§2.1), opportunities for power saving (§2.2), and non-deterministic execution of generative model workloads (§2.3).

2.1 Model Serving System Model

Model training is the process of building or learning a model from datasets, starting from scratch or pre-trained models [3]. The training process requires iterative *forward and backward passes*. After training, those ready-to-serve models are saved to a *model registry* (Fig. 1 left).

Model serving, or model inference, is to use the model to extract useful features from the inputs through a *forward pass* (Fig. 1 right). The structure of model-serving workloads follows a simple request-response paradigm where clients (either users or AI applications) submit requests for that model to a serving system, which schedules the requests, dispatches them to hardware devices (e.g., GPUs), and returns the results. Compared to model training, model serving does not involve complex backward computation and model weight updates but has more demanding performance requirements in inference latency (i.e., request completion time) and throughput. With the rise of pre-trained foundation models (e.g., in NLP and vision), it is expected that a model can be trained once and serve an extensive sequence of millions of inferences.

In this paper, we focus on the Deep Neural Network (DNN) model (including CNNs [16], Transformers [10], and Transformer-based generative models [46]) serving on homogeneous GPU clusters (Fig. 1).

Model Parallelism and Partitioning. To satisfy user demand, model serving often must adhere to stringent SLOs on latency (e.g., 99% of the requests for a model must be finished within 500 ms). However, there can be significant and unpredictable burstiness in the arrival process of user requests [22, 44, 49].

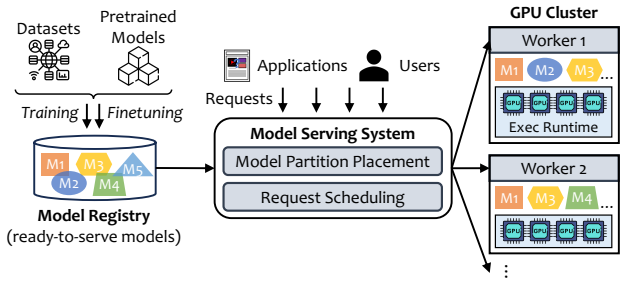


Figure 1: Model training and model serving.

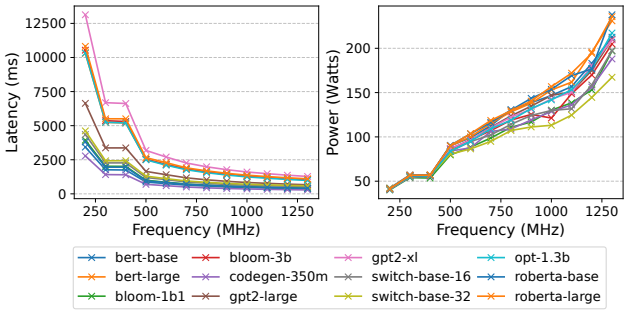


Figure 2: Model-serving latency and power consumption characterization at varying GPU core frequency levels.

Parallel model execution is necessary when attempting to satisfy the serving latency requirements or support large models that do not fit in the memory of a single device [22].

DNN models are composed of a series of *operators* (e.g., matrix multiplication and activation functions) over multidimensional *tensors*. There are two categories of model parallelism: *intra-operator* and *inter-operator* parallelism [57]. Intra-operator parallelism is to partition a single operator across multiple devices for parallel execution. It reduces the computation latency for serving a request but introduces the communication overhead of splitting the input and merging the output. Inter-operator parallelism is to partition a model's operator execution graph into multiple *stages* that can execute on multiple devices in a pipeline fashion. It allows the model to exceed the memory limitation of a single GPU device but it does not reduce the computation latency of a single request.

Power Efficiency. To the best of our knowledge, μ-Serve is the first power-aware DNN model-serving framework that aims to minimize power consumption while preserving the model-serving performance. State-of-the-art model-serving frameworks and commercial model-serving products use default GPU core frequency settings [27] for two main reasons. First, it is nontrivial to determine the optimal GPU frequency when serving diverse models or a mix of model partitions from different models when considering both stringent SLOs and costs. Second, GPU DVFS (dynamic frequency scaling based on utilization) is shown to be sub-optimal, imprecise, and has non-intuitive implementations in production [27].

Power optimization in ML model training [5, 6, 42, 50] has been active in the past few years. The central idea is to find the optimal GPU core frequency based on the valley trends when scaling frequencies (i.e., energy saving is maximized on the middle-level frequency). Building on top of that, EnvPipe [5] reduces frequencies at stages that are not on the critical path of the training pipeline with interleaving forward and backward passes. Perseus [6] identifies straggler stages (e.g., due to I/O bottleneck or hardware failures) and reduces frequencies to synchronize the speed of all other stages. However, these optimization tricks cannot be applied to model inference with a single forward pass. It gets challenging in model serving due to a mixture of model partition placement and dynamic request arrival patterns (unlike static and iterative training).

2.2 Opportunities for Power Saving

To understand if model serving also exhibits power-saving opportunities like model training, we start with an illustrative experiment to show how GPU frequency affects model serving latency. We use two NVIDIA Tesla V100 (16 GB) GPUs to serve each DNN model from Table 1. During the experiments, we reduce the GPU frequency from 1300 MHz to 200 MHz with a step size of 100 MHz. Fig. 2 (left) shows the end-to-end latency of serving a single request while Fig. 2 (right) shows the corresponding power efficiency.

The model-serving latency-frequency curves of all models exhibit a steep decline at low frequencies, but the decrease rate eventually approaches a horizontal asymptote at high frequencies. In contrast, the power curves for serving all models show a nearly linear relationship between the GPU frequency and the power consumption. This contrasting observation leads to a power-saving opportunity by reducing the GPU frequency when the SLO attainment is high. For example, serving `gpt2-large` can meet the SLO (1000 ms) even when the GPU frequency is reduced from 1300 MHz to 800 MHz. By doing so, the power consumption is reduced by a factor of 1.8× from 214 Watts to 120 Watts.

However, such power-saving opportunity varies as the SLO attainment depends on factors such as request arrival rate, burstiness, and SLO scales (as defined in [22]). In addition, coarse-grained GPU frequency scaling for an entire model cannot fully unlock the power-saving opportunity. We show in §3 how fine-grained model multiplexing and dynamic GPU frequency scaling can help based on the performance-power sensitivity and activation frequency of different operators.

2.3 Autoregressive Patterns

Traditional DNN models (e.g., ResNet and BERT) have deterministic execution patterns while Transformer-based generative models (e.g., GPT) are trained to generate the next token in an *autoregressive* manner. Therefore, to process a request to generative models, multiple *iterations* of the model have to be run; each iteration generates a single output token,

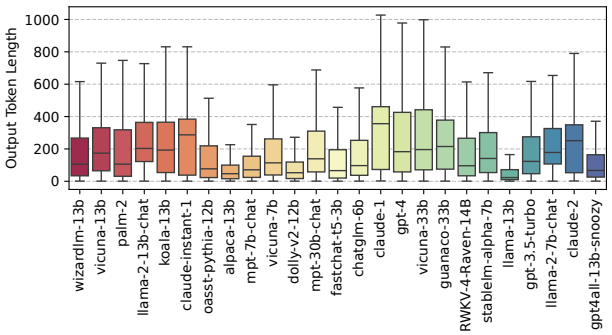


Figure 3: Output token length distributions of various large language models collected in the LMSYS-Chat-1M dataset.

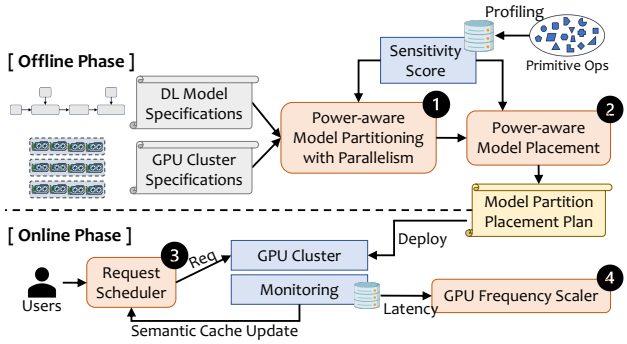


Figure 4: μ-Serve architecture overview and workflow.

which is then appended to the original input in the following iteration (except for the termination token `<EOS>`).

To illustrate this non-deterministic nature of serving model inference workloads due to autoregressive patterns, we analyze the output token length distribution in the LMSYS-Chat-1M dataset [56] which contains one million real-world conversations with 25 state-of-the-art LLMs. The output token length (N) dominates the execution time (T) of a request because $T = C + K * N$, where K is the latency to generate one token and C is the model-serving system's overhead including DNS lookups, proxies, queueing, and input tokenization. K depends on model optimization techniques (e.g., quantization) and execution environment (e.g., hardware), which are the same for all inputs. As shown in Fig. 3, the output token length of the model output varies significantly for the same model. The p95/p50 of the output token length for each model varies from 1.7 (`claude-1`) to 20.5 (`llama-13b`).

Since model executions are non-deterministic, it is challenging to differentiate the negative impact of GPU frequency down-scaling from the inherent non-determinism. Worse, it suffers from *head-of-line blocking* issues when scheduling requests for model-serving as most state-of-the-art LLM inference systems use a first-come-first-serve (FCFS) scheduling policy. FastServe [47] uses a multi-level feedback queue to improve the average job completion time but it does not address the non-determinism nature.

3 μ-Serve Design and Implementation

μ-Serve is a power-aware multi-model model-serving framework designed for a homogeneous GPU cluster that serves Deep Neural Network (DNN) models (including Transformers and Transformer-based generative models [46]) serving systems. From §2, we can see that there are several key challenges to improving power efficiency in DNN model serving:

- Determine the optimal SLO-preserving GPU frequency that minimizes power consumption (i.e., **[C1]**).
- Address non-deterministic execution patterns in generative models for performance predictability and avoiding head-of-line blocking (i.e., **[C2]**).
- Maximize the power-saving opportunity while GPUs do not support fine-grained frequency scaling (i.e., **[C3]**).

We propose μ-Serve to specifically tackle these challenges. The overall architecture of μ-Serve is shown in Fig. 4, which consists of an offline phase and an online phase. In the offline phase, μ-Serve first profiles each *primitive operator* [40] to get its *sensitivity score*, indicating how operator execution latency changes with the GPU frequency reduction. Based on the operator-level sensitivities, μ-Serve then generates power-aware model partitioning plans by utilizing model parallelism (§3.1) and deploys model partitions based on placement plans (§3.2) that maximize power-saving opportunities.

In the online phase, μ-Serve serves model inference requests while dynamically managing device frequency scaling. For each autoregressive model, μ-Serve uses a novel speculative shortest-job-first scheduler (§3.3) that improves SLO attainment by avoiding head-of-line blocking and thus increases power-saving opportunities. For the remaining models, μ-Serve uses the default FCFS scheduling policy. To minimize power consumption while preserving model-serving performance SLOs, μ-Serve dynamically scales GPU frequency at each device with a multiplicative-increase-additive-decrease (MIAD) algorithm (§3.4).

3.1 Power-aware Model Parallelism

Model parallelism is necessary to enable running larger models, or larger batches of inputs, and to serve bursty workloads with better latency and throughput [22, 57]. μ-Serve parallelism module (❶ in Fig. 4) partitions each ML model (by using model parallelism) to generate model partitions that can *maximize power efficiency* while considering the latency and throughput trade-offs so that the placement module (❷ in Fig. 4) can choose the best combination for all models in the whole cluster. To achieve this, μ-Serve relies on profiled *operator sensitivity scores* and AlpaServe [22], an auto ML model parallelization system for inference.

As described in §2.1, when compiling and deploying DNN models for request-serving, each model is commonly defined

Algorithm 1 Operator Clustering

Require: Model specification M, Operator sensitivity score database D (a map of <operator, score> pairs)

```
procedure OperatorClustering(M, D)
    clusters ← ∅
    currCluster ← ∅
    currCluster.add(M.head)
    clusterScore ← avg(currCluster)
    for each op in M.operators do
        if abs(D.get(op) - clusterScore < THRES) then
            currCluster.append(op)
            clusterScore ← avg(currCluster)
        else
            clusters.append(currCluster)
            currCluster ← newList([op])
        end if
    end for
    Return clusters
end procedure
```

Algorithm 2 Model Partition Placement

Require: Device group DP (list of GPUs), Model partition plan MP (multi-model), D (same in Alg. 1), CT (maximum number of devices to place to))

```
procedure ModelPlacement(DP, MP, D)
    bins ← ∅      ▷ a map of device to a list of partitions
    modelBinsCount ← ∅
    for each model in MP do
        // Sort partitions by sensitivity scores
        for each p in model.getSortedPartitions() do
            bestDevice = None
            bestScore = INFINITY
            selectedBins = getAvailableBins(bins, p)
            for each d, partitions in selectedBins do
                if modelbinsCount[model] < CT then
                    // Get similarity in sensitivity scores
                    score ← getSimilarity(p, partitions)
                    if bestScore < score then
                        bestScore = score
                        bestDevice = d
                    end if
                end if
            end for
            bins[bestDevice].append(p)
            modelBinsCount[model] += 1
        end for
    end for
    Return bins
end procedure
```

as a computational (dataflow) graph where nodes are *operators* and edges are *tensors* (or data) [48]. Example operators include matrix multiplication and `relu` or `tanh` activation functions). Serving a request means the completion of the dataflow in the execution graph. Our key insight is that (1) some operators are more *sensitive* to GPU frequency changes (i.e., *sensitive operators*) while others are not (i.e., *insensitive operators*); and (2) insensitive ones often contribute less to the end-to-end latency of serving a request. Since the minimum GPU frequency that a model partition can tolerate is determined by the most sensitive operators, μ-Serve avoids mixing sensitive and insensitive operators into the same partition.

Operator Sensitivity Score Profiling. Since the model graphs are represented in XLA's HLO format [40], a backend-agnostic intermediate representation (IR), we choose to profile only the *primitive operators* [2] defined in HLO that are shared by all models. We define the sensitive score of each operator as $\Delta Latency/\Delta Frequency$ where $\Delta Latency$ is the increase in execution time (normalized) of the operator when decreasing the GPU core frequency from the default setting to half of the default frequency and $\Delta Frequency$ is the frequency decrease[3]. We adopt the tensor dimensions from AI-Matrix [54], a production dataset released by Alibaba. When profiling primitive operators, we run each operator independently with each tensor dimension repeated 5000 times and take the average of all runs. Profiling of primitive operators happens offline and the obtained sensitivity score database is then used in model parallelism and placement plan generation.

Model Parallelism Plan Generation. Since different parallelization configurations (intra-/inter-operator parallelism) have different latency and throughput trade-offs, μ-Serve extends AlpaServe [22] to run an auto-parallelization compiler to generate a list of possible configurations for every single model and let the placement module (§3.2) choose the best combination for all models in the whole cluster.

At the core of AlpaServe runs an optimization algorithm with one inter-operator pass (based on dynamic programming) and one intra-operator pass (based on integer linear programming) to generate efficient model parallel partitions. AlpaServe treats each operator as the smallest unit for partitioning. To prevent operators with similar sensitivities from being split we do not change the AlpaServe auto-parallelism algorithm but extend AlpaServe with a new abstraction, i.e., *operator cluster*, which is a cluster of operators that share similar sensitivity scores. μ-Serve treats each operator cluster instead of each operator as the partitioning unit. As shown in Alg. 1, generating operator clusters can be achieved by taking the model specifications and profiled sensitivity scores and grouping neighbor operators with similar sensitivity scores (below a threshold) together. The output is the model parallelism plan (i.e., partitions and their memory demands for a

[2] As a unifying abstraction for multiple frameworks (e.g., Pytorch and TensorFlow) and hardware platforms (CPUs, GPUs, and TPUs), XLA summarizes common DL operators into around 100 primitive operators.

[3] We draw inspiration from delay sensitivity [28].

given model) that optimizes the per-request execution latency. We leave the decision of model placement (offline) to §3.2.

3.2 Power-aware Model Placement

Given a list of model partitions (generated by §3.1) and a fixed GPU cluster, model placement is to map each model partition to a GPU device within the memory constraint. The goal of model placement is to maximize SLO attainment (i.e., the percentage of requests served within SLO). Power-aware model placement module (❷ in Fig. 4) takes model partitions and the GPU cluster specification as inputs and generates a model partition placement plan that preserves SLOs while maximizing power-saving opportunities.

However, model placement is typically modeled as a bin-packing problem and is a computationally NP-hard problem [18]. For scalability and serving potentially a larger device cluster, we partition the cluster into several groups of devices (same as in AlpaServe). Each group of devices selects a subset of models to serve. Finding the optimal ⟨model partition, device group, device⟩ is a combinatorial optimization problem with a configuration space growing exponentially with the number of devices and the number of models. μ-Serve first selects the ⟨model partition, device group⟩ configurations that maximize SLO attainment based on AlpaServe [22] and then generates ⟨model partition, device⟩ configurations that maximize the power-saving opportunities. After deploying the placement plan, the offline stage is completed.

Mapping Models to Device Groups. μ-Serve runs the model placement algorithm in AlpaServe [22] that (1) uses a simulator-guided greedy algorithm to decide which models to select for each group, and (2) enumerates group partitions and model-parallel configurations to pick the best ⟨model, device group⟩ placement that maximizes SLO attainment.

Mapping Model Partitions to Devices. For each device group, given a list of devices and model partitions that are assigned to the device group from the last step, μ-Serve generates ⟨model partition, device⟩ placement plans that maximize the power-saving opportunities. Based on our insight that (1) operators have diverse sensitivity to GPU frequency scaling and (2) insensitive operators contribute less to the end-to-end latency, the intuition behind μ-Serve's model placement algorithm is to group operators with similar sensitivities together. Since how much the GPU frequency can be reduced is determined by the *most sensitive* model partition on that device, μ-Serve avoids co-locating a mix of sensitive and insensitive partitions onto the same GPU devices.

The overall procedure is shown in Alg. 2. μ-Serve first estimates the sensitivity scores of a model partition based on the weighted sum of operator sensitivity scores (using the operator count and tensor dimensions as the weights). All model partitions are then sorted based on the estimated sensitivity scores. For each model partition in the sorted list, we get the feasible "bins" or devices (by checking memory availability) and assign the partition to a bin that has the best *fitting score*. The fitting score for ⟨partition p, bin b⟩ is defined as the similarity between the sensitivity score of p and the weighted average of the sensitivity scores of all existing partitions in b. To avoid spreading partitions from the same model to too many devices (and thus inducing high communication overhead), we enforce an upper limit CT on how many bins a model can be packed into. CT is set based on the empirical results of the maximum number of devices to partition onto without SLO violations. The outcome is the model placement plan with a list of bins and corresponding model partition assignment. μ-Serve takes the model placement plan for deployment, which completes the offline stage.

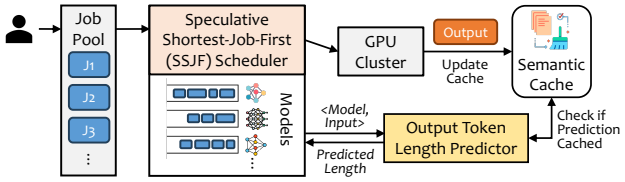


Figure 5: Workflow of the speculative scheduling in μ-Serve.

3.3 Scheduling

User requests are dispatched to corresponding models at runtime. For each model instance, a request queue is maintained, and μ-Serve uses a *speculative shortest-job-first (SJF) scheduler*(❸ in Fig. 4) to decide the request execution order of autoregressive models and a default FCFS scheduler for traditional deep learning models. SJF alleviates the non-determinism of generative models and the head-of-line blocking issue in FCFS scheduling. We must have knowledge or a good estimate to deploy SJF. As shown in Fig. 5, μ-Serve's speculative SJF scheduler relies on the prediction from an *output token length predictor*. The prediction is used to estimate the job execution time in SJF as the output token length dominates the execution time (linear relation) as described in §2.3. The predictor relies on a lightweight proxy model based on our insight that a small model is already quite good at predicting the lengthiness of a significantly larger (>20×) model's output. In addition, we use a *semantic cache* to improve the prediction accuracy of hot input queries.

Output Token Length Predictor. Given the user input to a certain generative model, our predictor tries to predict the length of the model's response in terms of the number of tokens because the model inference time is linearly related to the output token length (§2.3). We use BERT as a proxy model and build a multi-class classifier based on BERT as the output token length predictor. Specifically, we take the last layer hidden state of the first token (i.e., `CLS`) from the BERT output, and pass it through an additional two-layer feed-forward neural network to generate the prediction (Fig. 6). As suggested by the original BERT paper [10], the final hidden

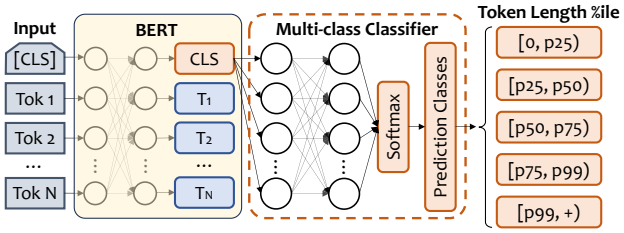


Figure 6: Output token length predictor in μ-Serve.

state corresponding to this token is used as the aggregate sequence representation for classification tasks. We could get better classification performance by averaging or pooling the sequence of hidden states for the whole input sequence. However, taking the whole sequence is more compute-intensive and, therefore, not worth the marginal gains.

Predictor Training. Our predictor adopts the pre-trained BERT model weights as provided by [10] and we further fine-tune the weights (together with the additional two-layer classifier network) on the LMSYS-Chat-1M training data [56]. Pre-training on a large corpus of unlabeled text gives BERT some basic understanding of human languages, while the fine-tuning step allows BERT to learn the *task-specific* knowledge (in our case, the output token length of a generative model given the input query). Our fine-tuning has two phases: In the first phase, the parameters of both BERT and the classifier are tuned, while in the second phase, we fix the weights of BERT and only update the classifier's parameters. Such a two-phase regime turns out to achieve a good balance between prediction accuracy and training efficiency.

Since predicting the exact length of the output can be difficult, we treat the token length prediction problem as a coarse-grained (multi-class) ordinal classification task [45]. Specifically, we characterize the output token length percentiles of each generative model based on its training samples and categorize the length labels of each model into 5 classes, namely [0, p25), [p25, p50), [p50, p75), [p75, p99), and [p99, +). We convert the classes into 5 integers $\{0,1,2,3,4\}$ and let our predictor output a real value as the prediction. We apply a mean squared error (MSE) training loss to minimize the distance between the real-valued predictions and the integer-valued class labels. During inference, we round the prediction value to the nearest integer in $\{0,\ldots,4\}$. As shown in Sec. 4.3, such an ordinal classification approach outperforms pure regression- or classification-based solutions.

Choices of Classification. The granularity of classification (i.e., how many classes to predict) is determined by the end-to-end scheduling results (in latency and throughput) instead of the prediction accuracy. It is because of our observation that better prediction accuracy does ***not*** necessarily lead to better scheduling performance. Empirical evaluations on our datasets suggest that as the number of classes rises, prediction accuracy drops (because it gets harder to predict more fine-grained output lengths), but scheduling performance first rises and then falls, peaking when there are 5 classes. We attribute this observation to the fact that the scheduler benefits from more fine-grained prediction but receives negative impacts when prediction is worse, i.e., when predicting >5 classes.

Use of Semantic Cache. The μ-Serve scheduler includes a simple *semantic cache* based on GPTCache [2] to store the ground truth output token sequence length of hot inputs. Instead of going through the predictor to get an estimated output length prediction, μ-Serve retrieves the cached length for the input that triggers a cache hit. Caching has been commonly used to reduce frequent and computationally expensive data accesses. In μ-Serve, a semantic cache is a $\langle$key, value$\rangle$ memory buffer where keys are *embeddings* and values are output lengths. Embeddings are generated by embedding models that map text inputs into a low-dimensional continuous vector space and are stored in a vector database [2]. For each input, the most similar cached input is retrieved using a similarity evaluation function to determine if the cached input matches the input query semantically. The cached ground truth length is then used by the scheduler. If there is a cache miss, the scheduler still uses the predicted output length. See §3.5 for implementation details.

3.4 GPU Frequency Scaling

μ-Serve's GPU frequency scaler (❹ in Fig. 4) consists of a monitoring and a frequency scaling component. The former monitors per-model request-serving latency while the latter determines appropriate GPU frequency scaling actions for each GPU device. We adopt a multiplicative-increase-additive-decrease (MIAD) algorithm [58] for GPU scaling based on the feedback signal of the slack between the target SLO latency and the actual latency, inspired by the AIMD algorithm in congestion control. The intuition behind adopting MIAD is that we want to be conservative when scaling down GPU frequencies to avoid interruptions to SLO attainment (addictive decrease) while being aggressive when scaling up GPU frequencies to respond fast to workload spikes.

As shown in Alg. 3, the scaler starts from the default frequency (usually the maximum frequency). After initialization, model latencies are sampled every 500 ms, and the frequency scaling step is determined at every iteration by scanning through all model partitions' latency *slack* (as defined in L10 in Alg. 3). For each model partition, any negative or small slack leads to an up-scaling action (we set the threshold = 0.05). Otherwise, we estimate the degradation of down-scaling the frequency based on the linear relationship between frequency and execution latency. A larger-than-degradation slack results in a down-scaling action. Ultimately, after scanning through all model partitions, an up-scaling action doubles the current frequency while a down-scaling action decreases the current frequency by $STEP$.

When it needs to scale up, there is no difference across

Algorithm 3 GPU Frequency Scaling

Require: GPU Device GD, Model Partitions MP (a map of ⟨model, partition⟩ pairs placed on GD)

1: **procedure** GPUFREQUENCYSCALING(GD, MP)
2: // Start from the default high frequency
3: $freq = DEFAULT$
4: $GD.setFrequency(freq)$
5: **while** $True$ **do**
6: $action = SAME$ ▷ one of $\{UP, DOWN, SAME\}$
7: **for each** p in MP **do**
8: $target = p$.getModelSLO()
9: $actual = p$.getModelLatency()
10: $slack = (target - actual)/target$
11: **if** $slack < 0.05$ **then**
12: $action = UP$ ▷ requires up-scaling
13: **break**
14: **end if**
15: $latency = p$.getWeight() * $actual$
16: $degradation = latency * STEP/freq$
17: **if** $degradation/target < slack - 0.05$ **then**
18: $action = DOWN$ ▷ down-scaling
19: **end if**
20: **end for**
21: **if** $action==UP$ **then**
22: $freq = min(MAX, freq * 2)$
23: **else if** $action==DOWN$ **then**
24: $freq = max(MIN, freq - STEP)$
25: **end if**
26: $GD.setFrequency(freq)$
27: **end while**
28: **end procedure**

model partitions because we typically need to aggressively scale up to mitigate SLO violations (by doubling the current frequency). L15 in Alg. 3 refers to the difference in scaling down, i.e., a partition with lower sensitivity scores has lower weights, and thus can tolerate more down-scaling. Note that when the current frequency reaches the maximum but scaling-up is repeatedly called, it can be treated as a signal to replicate the model instance (i.e., autoscaling to scale out). Each iteration is performed every second.

3.5 Implementation

μ-Serve is implemented with 6.1K lines of code in Python on top of an existing model-parallel inference system, AlpaServe [22]. We extend its auto-parallelization algorithms to get the power-aware model-parallel strategies and placement plans. In μ-Serve's scheduler, the semantic cache is implemented with GPTCache [2]. We use the HuggingFace embedding function, the `NumpyNormEvaluation` similarity evaluation function (with a default similarity threshold of 0.8), the SQLiteCache cache manager, and the FAISS vector database. To avoid starvation in scheduling, μ-Serve adopts aging [36] to promote jobs with long waiting times. Preemption could help correct previous suboptimal decisions with a least-slack-time-first policy. However, μ-Serve does not adopt preemption due to its added complexity in context switch [1], memory management [20], and cache replacement policies [47] (for key-value cache in serving LLMs).

Table 1: Details of the models used in experiments. *The latency is measured for a single query with a sequence length of 512 on a single GPU. AR stands for autoregressive.*

Model	# of Params	Size	Latency	AR?
ResNet-50	25M	0.2 GB	51 ms	No
BERT-base	110 M	0.5 GB	123 ms	No
BERT-large	340 M	1.4 GB	365 ms	No
RoBERTa-base	125 M	0.5 GB	135 ms	No
RoBERTa-large	355 M	1.4 GB	382 ms	No
OPT-1.3b	1.3 B	5.0 GB	1243 ms	Yes
OPT-2.7b	2.7 B	10.4 GB	2351 ms	Yes
GPT2-large	774 M	3.3 GB	832 ms	Yes
GPT2-xl	1.5 B	6.4 GB	1602 ms	Yes
CodeGen-350m	350 M	1.3 GB	357 ms	Yes
CodeGen-2b	2.0 B	8.0 GB	2507 ms	Yes
Bloom-1b1	1.1 B	4.0 GB	523 ms	Yes
Bloom-3b	3.0 B	11.0 GB	1293 ms	Yes
Switch-base-16	920 M	2.4 GB	348 ms	Yes
Switch-base-32	1.8 B	4.8 GB	402 ms	Yes

4 Evaluation

Our experiments addressed the following research questions:

§4.2 Does μ-Serve achieve power-saving (and how much) while preserving SLO attainment?

§4.3 How much does the proxy-model-based scheduler in μ-Serve improve latency and throughput?

§4.4 How robust is μ-Serve's power-aware model partitioning and placement under model variations?

4.1 Experiment Setup

Models. Since μ-Serve is a general ML model-serving framework, we consider traditional non-Transformer models, Transformers, and Transformer-based generative models for evaluation. For each model family, we select several most commonly used model sizes and variants (to mimic different fine-tuned versions) for experimentation. Table 1 provides details about model sizes and inference latency on testbed GPUs.

Workloads. We use the Microsoft Azure function traces [55] and production traces from a private datacenter dedicated to ML workloads in SenseTime [17] to drive the inference workloads. Since the SenseTime dataset contains both ML training and inference workloads (without distinction), we select the traces with job completion time $<$ 5 seconds and the number

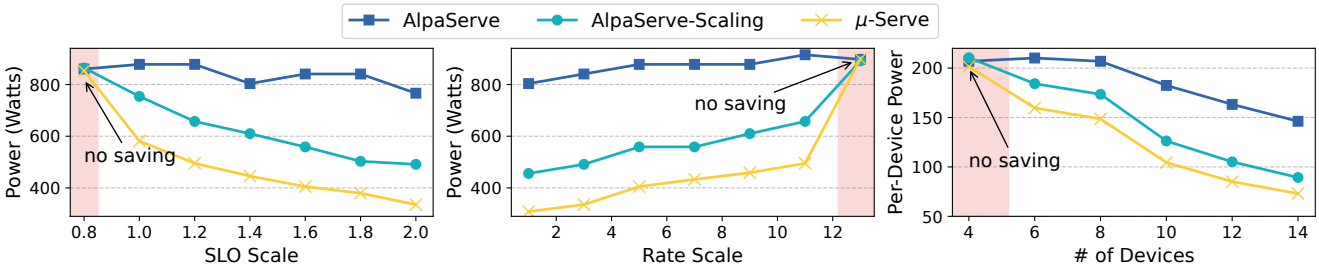


Figure 7: Power saving evaluation.

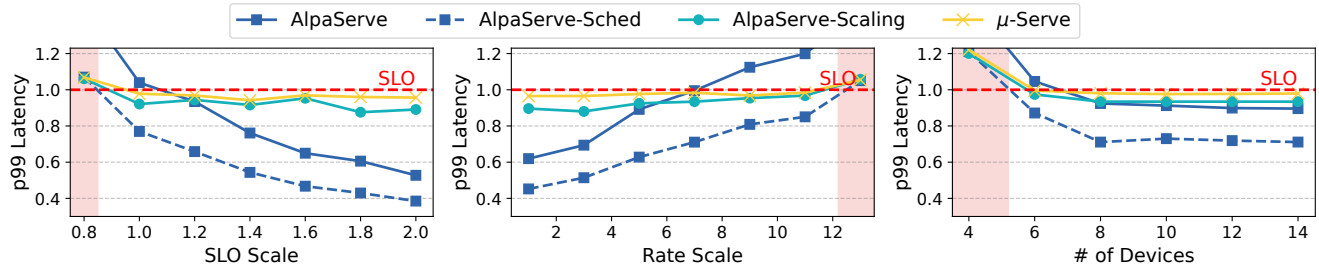


Figure 8: SLO preservation evaluation.

of used GPUs ≥ 1 to ignore training jobs and admin query workloads (CPU workloads, e.g., training progress check). For each model in Table 1, we round-robin serverless functions and ML inference jobs to generate traffic. For vision model inputs, we use the ImageNet dataset [9]. For language model inputs, we use the LMSYS-Chat-1M dataset [56] which contains one million real-world conversations.

Metrics. We use SLO attainment (i.e., the percentage of requests served within the latency SLO) and power consumption (in Watts) as the major evaluation metrics since the goal of μ-Serve is to minimize power consumption while preserving SLO performance. We adopt the definition of *SLO Scales* from AlpaServe [22] which refers to different multiplies of the SLO latency of the model request. The SLO latency of the model request is measured as the single-request latency with the parallelism plan generated by AlpaServe. We are also interested in model-serving throughput and how these measures change with varying numbers of devices, request arrival rates, and traffic burstiness.

Testbed. We deploy μ-Serve on a cluster with 8 nodes and 16 GPUs. Each node is an IBM Cloud `gx2-16x128x2v100` instance with 2 NVIDIA Tesla V100 (16GB) GPUs. Each GPU supports a maximum Streaming Multiprocessor (SM) frequency of 1380 MHz and a minimum of 200 MHz.

4.2 End-to-end Results

In this section, we compare μ-Serve against three baselines (AlpaServe [22], AlpaServe-Sched, and AlpaServe-Scaling) when serving widely used open-source large deep learning models on publicly available workload traces (as described in §4.1). AlpaServe is a state-of-the-art model-serving system that generates model parallelism and placement plans to maximize SLO attainment under varying workload conditions. AlpaServe-Sched is AlpaServe with the FCFS scheduler replaced with μ-Serve's scheduler (§3.3) but without GPU frequency scaling. AlpaServe-Scaling deploys model partitions according to parallelism and placement plans from AlpaServe and runs μ-Serve's GPU frequency scaling algorithm (§3.4) and scheduler at runtime. We are interested in the power-saving comparison and knowing whether μ-Serve achieves power-saving while preserving the SLO attainment.

We deploy all models (in Table 1) in the experiments. Each model instance is driven by an independent request generator using a randomly selected trace set as described in §4.1. The memory consumption of all models (including activation and runtime contexts) is 62.5 GB which can be fit onto 4 devices.

Power Saving. Fig. 7 shows the power consumption of serving all models with varying SLO scales, (combined) arrival rates, and the number of devices. Note that when SLO attainment is violated, there is no power saving (as shown in the red-shaded area, e.g., an SLO scale of 0.8) because the device frequency is the same for all approaches. When SLO attainment is met, μ-Serve reduces the power consumption in AlpaServe by 1.51–2.29× at different SLO scales. At different arrival rates, the reductions in power consumption are 1.85–2.61×, compared to AlpaServe. The larger the SLO scale (less stringent SLO) and the smaller the arrival rate, the more the power saving. We find that adding dynamic GPU frequency scaling on top of AlpaServe has already been able to reduce power consumption by a factor of 1.1–1.5× and 1.4–1.7× at varying SLO scales and rates. μ-Serve further improves the power saving by an average of 1.4× compared to AlpaServe-Scaling. We find that the improvement does not vary with SLO scales or arrival rates because it is the statically set model partition placement strategies that cause different power savings

from the same GPU frequency scaling approach.

In the experiments with varying numbers of devices, we replicate each model instance (so now the total model memory demand is up to 8 devices) and increase the number of devices from 4 to 14. Fig. 7 (right) shows the per-device average power consumption. When there are no idle devices (i.e., at 6 and 8 devices), the power saving from μ-Serve compared with AlpaServe is 1.23–1.39×. When there are more idle devices, all per-device power consumption curves start to decrease. Since μ-Serve and AlpaServe-Scaling actively down-scale the GPU frequency, the power saving increases to 1.75–2× and 1.44–1.64× compared to AlpaServe. AlpaServe-Sched is not shown in Fig. 7 since we observe that it has the same power consumption as AlpaServe. This is because only the execution ordering of the requests is different between the two baselines and there is no idle time. Therefore, the scheduler does not have an effect on power saving.

Model Serving Performance. We evaluate the SLO attainment to understand what the cost of μ-Serve is while achieving power savings. We measure the 99th-percentile (p99) latency of serving a request for each model and normalized it to the SLO latency. When the normalized p99 latency is less than 1, it means the SLO attainment is greater than 99%. On the other hand, the model serving system does not meet the SLO attainment of 99%. Fig. 8 shows the normalized p99 latency of serving all models with varying SLO scales, (combined) arrival rates, and the number of devices.

Same as in the power-saving evaluation, the red-shaded area indicates that the 99% SLO attainment of AlpaServe-Sched is not met. Therefore, the three approaches AlpaServe-Sched, AlpaServe-Scaling, and μ-Serve converge to nearly the same p99 latency because the device frequencies are fixed at the maximum. When there is no SLO violation in AlpaServe-Sched, μ-Serve and AlpaServe-Scaling both meet the 99% SLO attainment. μ-Serve has 2–9% higher p99 latency than AlpaServe-Scaling. By replacing FCFS with μ-Serve's scheduler, AlpaServe-Sched achieves 26–31% lower p99 latency. In the experiments with varying numbers of devices (as shown in Fig. 8 (right)), since serving all models requires at least 8 devices, there are SLO attainment violations at 4 and 6 devices. At 6 devices, only AlpaServe fails to meet the SLO attainment. At 8 devices, all approaches meet the SLO attainment, and μ-Serve has 5% higher p99 latency compared to AlpaServe-Scaling, which also holds when there are idle devices (i.e., at 10–14 devices).

Without SLO violations, μ-Serve achieves 1.5–2.3×, 1.9–2.6×, and 1.8–2× power saving compared to AlpaServe under varying SLO scales, request arrival rates, and number of devices.

4.3 Scheduling

In this section, we evaluate the output length predictor and scheduler performance. Additional ablation studies on the scheduler can be found in appendices A.2 and A.3.

Table 2: Output token length predictor evaluation results. Columns are five predictor types: ordinal classification with MSE/L1 loss, classification, and regression with MSE/L1 loss.

Metrics	Ord. CLS (MSE)	Ord. CLS (L1)	CLS	REG (MSE)	REG (L1)
Accuracy ↑	0.4290	0.4272	**0.4599**	0.4014	0.4383
F1 Score ↑	0.3563	0.3332	**0.4021**	0.3318	0.3477
L1 Error ↓	0.7587	0.7480	N/A	0.6077	**0.5557**
MSE ↓	0.9859	1.0857	N/A	**0.8077**	0.8996

Token Length Predictor Evaluation. We evaluate our output token length predictor on the LMSYS-Chat-1M dataset. Overall, our predictor achieves an average accuracy of 0.42 across 25 models (compared to the accuracy of 0.2 for a random guess). As shown in Fig. 11, despite using a shared predictor, our predictor achieves rather balanced performances across different models, with the prediction accuracy ranging from 0.33 for `llama-13b` to 0.50 for `wizardlm-13b`. Training a customized predictor for each individual model leads to slightly better accuracy. For instance, a customized predictor for `vicuna-13b` achieves an accuracy of 0.429 (>0.42 for a general predictor). However, training customized predictors can incur more training and model management overhead. Therefore, we do not proceed with customized predictors.

We also conduct ablation studies to investigate the effectiveness of our ordinal-classification-based token length predictor. In Table 2, we compare our predictor with multiple alternative approaches, including ordinal classification with L1 loss, (standard) classification with cross-entropy loss and resampling, and regression with MSE & L1 loss. Interestingly, we find that higher prediction accuracy ***does not*** necessarily lead to better scheduling performance (e.g., lower latency or higher throughput). Instead, it matters more to the scheduler that the predictor can rank any two inputs correctly. A predictor with higher mean accuracy may mispredict several long outputs to be very short, leading to HoL and much worse latency (relevant to the findings in [11]). Table 2 shows that our ordinal classifier with MSE loss fails to outperform the alternative predictors in any metrics. However, as we will show in the scheduler evaluation next, μ-Serve's scheduler benefits more from our chosen predictor despite a slightly lower accuracy of 0.4290 (< 0.4599 compared to CLS).

Scheduler Evaluation. We evaluate the scheduling performance regarding the job completion time (JCT) and the serving throughput to understand if prediction accuracy is sufficient. Our comparison baselines include (1) first-come-first-serve (FCFS), which is the default scheduler in state-of-the-art model-serving frameworks such as AlpaServe [22] and Orca [51], and (2) shortest-job-first (SJF) with oracle output token prediction. Fig. 9 and Fig. 10 show the scheduling performances with varying request arrival rates and burstiness. We also compare with three alternative designs, i.e., SJF with

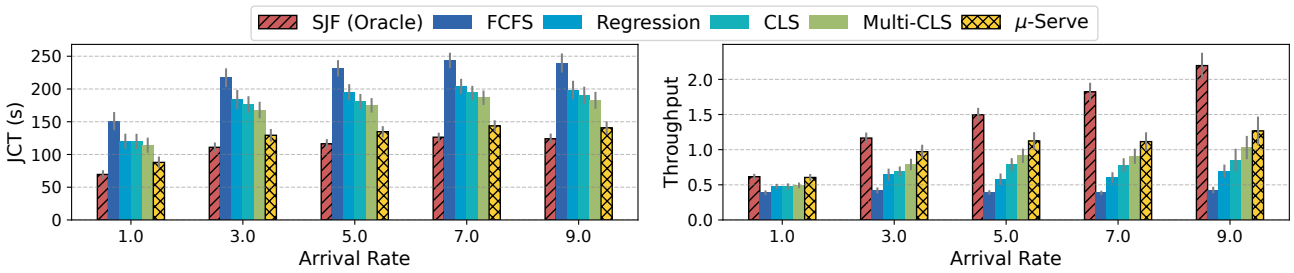


Figure 9: Scheduler evaluation across varying request arrival rates.

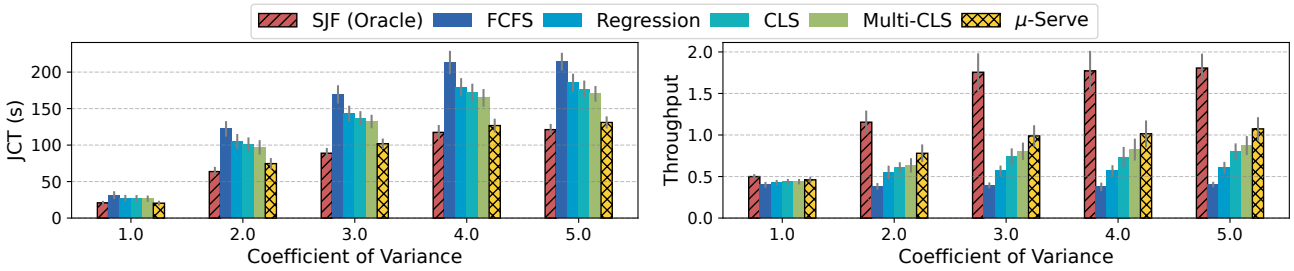


Figure 10: Scheduler evaluation across varying request burstiness.

regression predictor, binary-class predictor, and multi-class predictor (with no semantic cache). With varying request rates, μ-Serve reduces JCT by 41.3% compared to FCFS and increases the throughput by 2.5×. In comparison, the oracle SJF reduces the JCT by 49.8% and increases the throughput by 3.6×. Under different burstinesses, μ-Serve reduces JCT by 38.6% compared to FCFS and increases the throughput by 2.2×. In comparison, the oracle SJF reduces the JCT by 43.8% and increases the throughput by 3.6×. With semantic cache, μ-Serve improves JCT by 23% and increases throughput by 1.2× (i.e., the comparison with Multi-CLS).

Predictor Overhead. We evaluate the latency overhead of the predictor for autoregressive models (as indicated in Table 1) because the output token prediction is only on the critical path of serving autoregressive models. The latency overhead includes both semantic cache lookup and proxy model prediction. Fig. 12 compares the predictor latency overhead and model request execution time. Note that the Y-axis is in the log scale. The average model execution time is 9.8 s, and the p5th execution time is 360 ms. The average predictor latency is 37.6 ms (0.4% of the total latency), and the maximum is 56.3 ms, which is less than the minimum model execution time of 120 ms. Therefore, we conclude that the predictor introduces negligible overhead to the end-to-end model-serving latency of autoregressive models, which supports the design and benefits of using a lightweight proxy model in μ-Serve.

Better prediction accuracy does not necessarily translate to better scheduling performance. With negligible overhead, μ-Serve reduces the average JCT by up to 41.3% and increases the throughput by up to 2.5× at varying workloads.

4.4 Model Partitioning and Placement

In this section, we aim to understand the power-saving opportunity concerning the percentage of insensitive operators in a model, output non-determinism, and workload variation.

Power Saving Opportunity vs. Insensitivity Ratio. Power saving stems from the key insight that operators have diverse sensitivity to GPU frequency scaling, and thus, insensitive operators are leveraged to down-scale GPU frequency while preserving the SLO attainment. We define insensitive operators to be those model operators with sensitivity scores less than 1. Fig. 14 (a) illustrates how power-saving opportunities vary with the percentage of insensitive operators in different open-source models (up to 60%). Compared with AlpaServe-Scaling, the power reduction increases by 2.2× when the percentage increases from 10% to 60%.

Robustness to Model Execution Non-determinism. Fig. 14 (b) shows the relationship between power saving and the model output length. We find that as the average model output length increases (i.e., more iterations per model request), the power reduction of μ-Serve compared to AlpaServe decreases from 59% to 10% because of more intensive model executions (and thus fewer power-saving opportunities).

Robustness to Workload Variations. As shown in §4.2, μ-Serve achieves power saving while preserving SLO attainment by dynamic GPU frequency scaling. In reaction to workload variations, Fig. 13 illustrates the scaling actions of the MIAD algorithm in μ-Serve with the changes in request arrival rates. μ-Serve adopts an aggressive (multiplicative) frequency increase policy and conservatively decreases the frequency at each step (e.g., during the time 10–20 seconds).

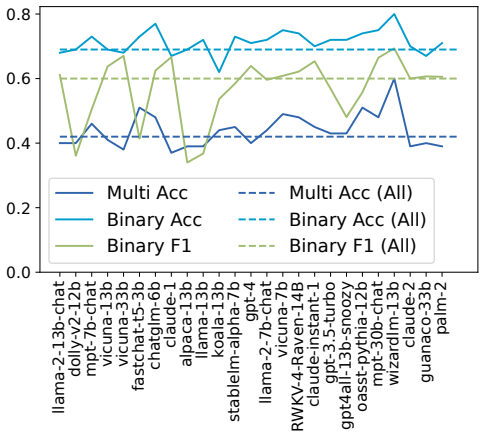


Figure 11: Predictor accuracy.

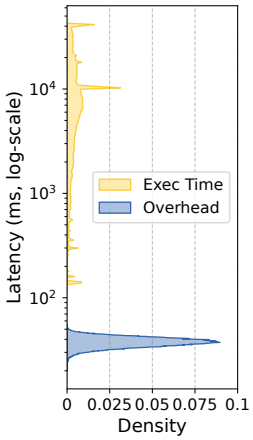


Figure 12: Overhead.

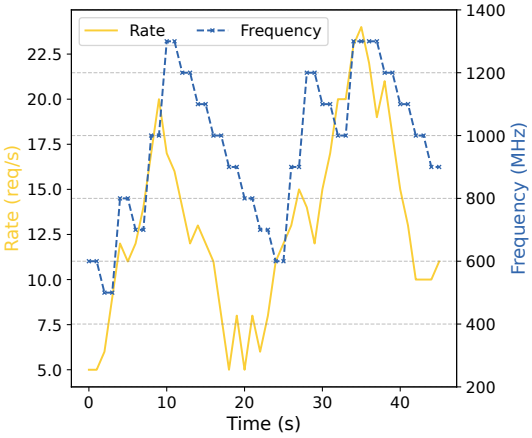
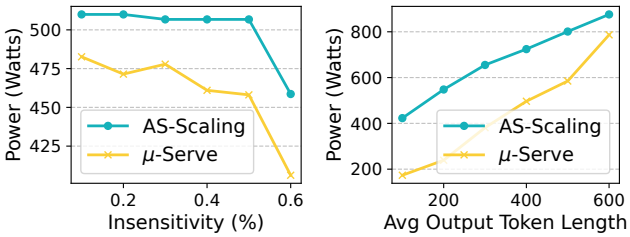


Figure 13: Frequency scaling.



(a) Insensitivity (%). **(b)** Output variation.

Figure 14: Robustness analysis.

Note that if the frequency stays at the maximum frequency but there are continuous SLO attainment violations, a scale-out signal can be sent to the model autoscaler (which μ-Serve does not implement), left to the discussion in §6.

Power saving varies with insensitivity ratio and model output intensities. Power-aware partitioning/placement in μ-Serve gives us maximized saving *opportunities* that only pay off when the frequency-scaler dynamically down-scales device frequencies within the SLO attainment requirement.

5 Related Work

General Model-Serving Systems. The emergence of ML applications motivates the prevalence of ML model-serving systems that provide services such as scheduling, placement, batching, and autoscaling. Clipper [8], TensorFlow-Serving [26], MArk [52], InferLine [7], Shepherd [53], and Clockwork [14] are some earlier general ML model serving systems for serving models like ResNet that are relatively small. They support latency-aware provision to maximize the overall goodput. More recently, AlpaServe [22] utilizes model parallelism for statistical multiplexing. INFaaS [35] and Cocktail [15] propose a model-less serving framework to automate the model selection and autoscaling to meet SLOs. However, these general systems are not power-aware and fail to consider the autoregressive property of LLMs. Instead, dynamic batching and autoscaling are complementary to μ-Serve.

LLM Inference Optimization. Recently, several model-serving systems [20, 47, 51] have been proposed to optimize LLMs. Orca [51] considers the autoregressive token generation pattern of LLMs and introduces iteration-level scheduling. However, it uses a first-come-first-serve (FCFS) scheduling policy that suffers from head-of-line blocking which we address in this paper. FastServe [47] proposes preemptive scheduling with a Multi-Level Feedback Queue. However, it introduces extra memory overhead to maintain intermediate states for unfinished jobs. vLLM [20] is a high-throughput LLM serving engine based on *PagedAttention* for token storage that achieves near-zero waste in KV cache memory. vLLM also adopts an FCFS scheduling policy for all requests. TurboTransformers [12] proposes a memory allocation algorithm to balance the memory footprint and (de-)allocation efficiency and a batching scheduler using dynamic programming to achieve optimal throughput. μ-Serve can benefit from the memory management optimization of these systems and improve the power efficiency of serving LLMs.

Operator Placement Optimization. AlpaServe [22] partitions collections of models using inter- and intra-operator parallelism and generates operator placement strategies to reduce serving latency in the presence of bursty workloads. μ-Serve relies on AlpaServe to generate feasible operator parallelism partitioning plans. Another line of work is GPU-centric offloading which utilizes CPU memory or disk to store portions of the model parameters. For instance, PowerInfer [39] loads frequently-activated neurons onto the GPU for fast access while offloading infrequently-activated neurons to the CPU, thus significantly reducing GPU memory demands and CPU-GPU data transfers. FlexGen [37] proposes a novel scheduling approach to prioritize throughput

over latency, processing batches sequentially for each layer. DejaVu [23] accelerates LLM inference by using activation sparsity to selectively process only those neurons that are predicted to be activated. These systems are orthogonal to μ-Serve as their objective is to reduce GPU memory demand with offloading while μ-Serve aims to improve the power efficiency of model serving.

Speculative LLM Inference. Speculative decoding or look-ahead decoding accelerates LLM token generation with smaller approximation models [21, 25], multiple decoding heads [4], or n-gram generation [13]. We do not consider speculative LLM inference because of its extra computational overhead of token generation and verification.

6 Discussion and Future Challenges

Batching and Autoscaling. Request batching and autoscaling (adding/removing model instances) are two essential orchestration tasks. Adaptive batching [8, 52] has been used to maximize throughput while meeting latency objectives [8]. Model replica autoscaling is another widely used model-serving technique [8, 15, 35, 52] to adapt to workload spikes. SLO-oriented cloud workload autoscaling and frequency scaling techniques [31, 33, 34, 43] can also potentially benefit model serving. These techniques are complementary to μ-Serve. We present how μ-Serve scheduler works with various batching techniques in [32], and we leave it to future work on integrating μ-Serve with dynamic model replication.

Heterogeneous Accelerators. In μ-Serve's system model, we consider only a homogeneous GPU cluster. However, GPU devices in a cluster can be heterogeneous in terms of hardware (e.g., A100, V100, and T4), resource configurations (e.g., memory size), frequency range, and power features [19, 38, 44]. Device heterogeneity raises challenges in both model provisioning (e.g., which type of device to assign to a specific model partition) and dynamic frequency scaling (the power saving, idle power, and power efficiency differ across different types of devices). We leave the study of such complicated optimization space to future work.

Other Power Management Features. In μ-Serve's GPU management model, we consider only SM frequency scaling because of fast actuation and fine-grained increments. Instead, there are other power management features on more advanced GPUs such as power capping, various GPU operation modes, memory frequency scaling, and MIG sharing [27]. It is worthwhile to study the interplay between these power features and explore co-optimization policies for power saving.

Scalability. Fig. 7 shows that μ-Serve consistently facilitates per-device power savings as the cluster undergoes scaling out at no idle devices. Based on these results, we assert that similar power-saving opportunities are likely to be present in cloud-scale deployment while ensuring SLO attainment.

7 Conclusion

We have presented μ-Serve, a system for serving multiple large deep learning models that maximizes power saving while preserving request-serving SLO attainment. The key innovation of μ-Serve lies in the fine-grained modeling of operator sensitivity and power-aware model provisioning that creates power-saving opportunities. Such opportunities can be exploited by dynamic GPU frequency scaling online to achieve power reduction without SLO violations at varying conditions. We quantify and discuss when and to what extent such saving opportunities exist through extensive evaluations.

Acknowledgments

We thank the anonymous reviewers for providing their valuable feedback. This work is supported by the National Science Foundation (NSF) under grant No. CCF 20-29049 and by the IBM-ILLINOIS Discovery Accelerator Institute (IIDAI). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or IBM.

References

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

[2] Fu Bang and Feng Di. GPTCache: An open-source semantic cache for LLM applications enabling faster answers and cost savings. In *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, pages 212–218, 2023.

[3] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[4] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, and Tri Dao. Medusa: Simple framework for accelerating LLM generation with multiple decoding heads, 2023. https://github.com/FasterDecoding/Medusa.

[5] Sangjin Choi, Inhoe Koo, Jeongseob Ahn, Myeongjae Jeon, and Youngjin Kwon. EnvPipe: Performance-preserving DNN training framework for saving energy. In *Proceedings of the 2023 USENIX Annual Technical Conference (ATC 2023)*, pages 851–864, 2023.

[6] Jae-Won Chung, Yile Gu, Insu Jang, Luoxi Meng, Nikhil Bansal, and Mosharaf Chowdhury. Perseus: Removing energy bloat from large model training. *arXiv preprint arXiv:2312.06902*, 2023.

[7] 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 *Proceedings of the 11th ACM Symposium on Cloud Computing*, page 477–491, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2017)*, pages 613–627, 2017.

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

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional Transformers for language understanding. In *Proceedings of the 2019 North American Chapter of the Association for Computational Linguistics*, 2019.

[11] Yi Ding, Nikita Mishra, and Henry Hoffmann. Generative and multi-phase learning for computer systems optimization. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA 2019)*, page 39–52, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. TurboTransformers: An efficient GPU serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[13] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Break the sequential dependency of LLM inference using lookahead decoding, 2023. https://lmsys.org/blog/2023-11-21-lookahead-decoding/.

[14] Arpan Gujarati, Reza Karimi, et al. Serving DNNs like clockwork: Performance predictability from the bottom up. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, pages 443–462, 2020.

[15] 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 *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2022)*, pages 1041–1057, 2022.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016)*, June 2016.

[17] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale GPU datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2021)*, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Klaus Jansen, Stefan Kratsch, Dániel Marx, and Ildikó Schlotter. Bin packing with fixed number of bins revisited. *Journal of Computer and System Sciences*, 79(1):39–49, 2013.

[19] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, pages 463–479, 2020.

[20] 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 *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 2023)*, pages 611–626, 2023.

[21] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*, pages 19274–19286. PMLR, 2023.

[22] 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 *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2023)*, pages 663–679, 2023.

[23] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang,

Yuandong Tian, Christopher Re, et al. DejaVu: Contextual sparsity for efficient LLMs at inference time. In *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*, pages 22137–22176. PMLR, 2023.

[24] Alexandra Sasha Luccioni, Yacine Jernite, and Emma Strubell. Power hungry processing: Watts driving the cost of AI deployment? *arXiv preprint arXiv:2311.16863*, 2023.

[25] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. SpecInfer: Accelerating generative LLM serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 2023. https://arxiv.org/abs/2305.09781.

[26] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. TensorFlow-Serving: Flexible, high-performance ML serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[27] Pratyush Patel, Zibo Gong, Syeda Rizvi, Esha Choukse, Pulkit Misra, Thomas Anderson, and Akshitha Sriraman. Towards improved power management in cloud GPUs. *IEEE Computer Architecture Letters*, 22(2):141–144, 2023.

[28] Archit Patke, Saurabh Jha, Haoran Qiu, Jim Brandt, Ann Gentile, Joe Greenseid, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Delay sensitivity-driven congestion mitigation for HPC systems. In *Proceedings of the ACM International Conference on Supercomputing (ICS 2021)*, page 342–353, New York, NY, USA, 2021. Association for Computing Machinery.

[29] David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. The carbon footprint of machine learning training will plateau, then shrink. *Computer*, 55(7):18–28, 2022.

[30] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017)*, pages 325–341, 2017.

[31] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*, pages 805–825, November 2020.

[32] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. Efficient interactive LLM serving with proxy model-based sequence length prediction. In *The 5th International Workshop on Cloud Intelligence / AIOps at ASPLOS 2024*, volume 5, pages 1–7, San Diego, CA, USA, 2024. Association for Computing Machinery.

[33] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Chen Wang, Hubertus Franke, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. FLASH: Fast model adaptation in ML-centric cloud platforms. In *Proceedings of the 7th Annual Conference on Machine Learning and Systems (MLSys 2024)*, 2024.

[34] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems. In *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC 2023)*, pages 387–402, 2023.

[35] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *Proceedings of 2021 USENIX Annual Technical Conference (ATC 2021)*, pages 397–411, 2021.

[36] Yonatan Shadmi. Fluid limits for shortest job first with aging. *Queueing Systems*, 101(1-2):93–112, 2022.

[37] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*, pages 31094–31116. PMLR, 2023.

[38] Fengguang Song and Jack Dongarra. A scalable framework for heterogeneous GPU-based clusters. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2012)*, pages 91–100, 2012.

[39] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast large language model serving with a consumer-grade GPU. *arXiv preprint arXiv:2312.12456*, 2023. https://arxiv.org/abs/2312.12456.

[40] Google XLA Team. XLA: Optimizing compiler for machine learning, 2017.

[41] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: On the importance of pre-training compact models. *arXiv preprint arXiv:1908.08962*, 2019.

[42] Farui Wang, Weizhe Zhang, Shichao Lai, Meng Hao, and Zheng Wang. Dynamic GPU energy optimization for machine learning training workloads. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):2943–2954, 2021.

[43] Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. SOL: Safe on-node learning in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*, pages 622–634, New York, NY, USA, 2022.

[44] 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 *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2022)*, pages 945–960, 2022.

[45] Christopher Winship and Robert D Mare. Regression models with ordinal variables. *American Sociological Review*, pages 512–525, 1984.

[46] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020)*, pages 38–45, 2020.

[47] 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.

[48] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, et al. GSPMD: General and scalable parallelization for ML computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

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

[50] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. Zeus: Understanding and optimizing GPU energy consumption of DNN training. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2023)*, pages 119–139, 2023.

[51] 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 *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, pages 521–538, 2022.

[52] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proceedings of 2019 USENIX Annual Technical Conference (ATC 2019)*, pages 1049–1062, 2019.

[53] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. Shepherd: Serving DNNs in the wild. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2023)*, pages 787–808, 2023.

[54] Wei Zhang, Wei Wei, Lingjie Xu, Lingling Jin, and Cheng Li. AI-Matrix: A deep learning benchmark for Alibaba data centers. *arXiv preprint arXiv:1909.10562*, 2019.

[55] 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 *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021)*, pages 724–739, 2021.

[56] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric Xing, et al. LMSYS-Chat-1M: A large-scale real-world LLM conversation dataset. *arXiv preprint arXiv:2309.11998*, 2023.

[57] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*, pages 559–578, 2022.

[58] B. Zurita Ares, P. G. Park, C. Fischione, A. Speranzon, and K. H. Johansson. On power control for wireless sensor networks: System model, middleware component and experimental evaluation. In *2007 European Control Conference (ECC 2007)*, pages 4293–4300, 2007.

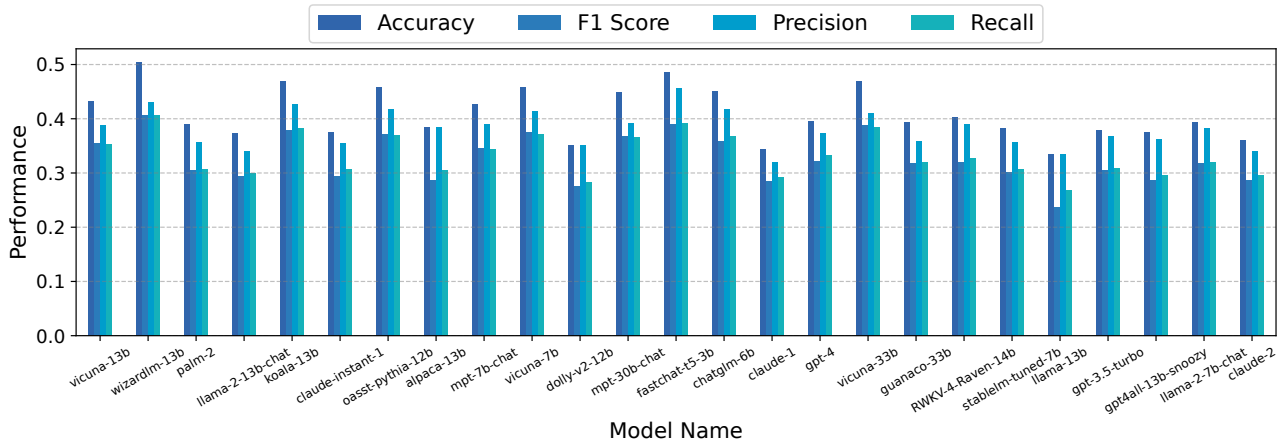


Figure 15: Output token length predictor evaluation results for each individual model.

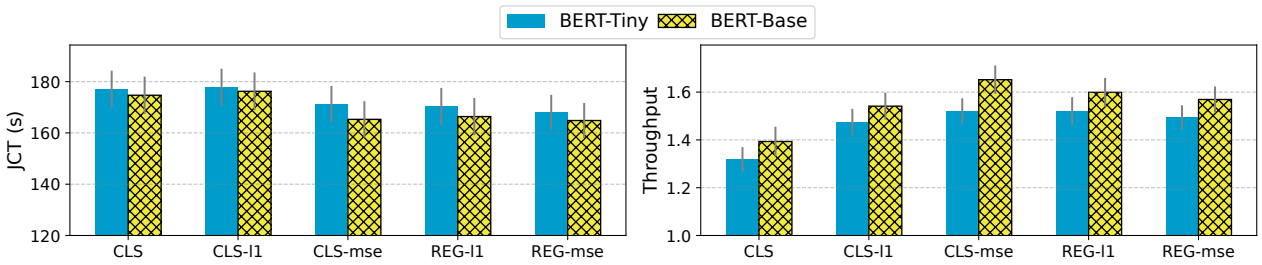


Figure 16: Scheduling performance comparison with BERT-Base and BERT-Tiny as the proxy models.

A Appendix

A.1 Artifact

An open-source implementation of μ-Serve and results are available at: https://gitlab.engr.illinois.edu/DEPEND/power-aware-model-serving.

A.2 Ablation Study on Proxy Models in the Output Length Predictor

Model-Specific Results. In Fig. 15, we plot the detailed evaluation results of our output token length predictor on each individual model from the LMSYS-Chat-1M dataset. We consider standard evaluation metrics including accuracy, precision, recall, and F1 score. Our predictor achieves rather balanced performances across different models, with the prediction accuracy ranging from 0.33 for `llama-13b` to 0.50 for `wizardlm-13b`.

Evaluation Results for Smaller Predictor Models. In the main text of the paper, we primarily build our predictor on top of the "BERT-Base" model [41] with 12 Transformer layers and 768 hidden dimensions. In this appendix, we consider a smaller predictor based on the "BERT-Tiny" model with 2 layers and 128 dimensions. Table 3 summarizes the evaluation results of multiple predictors built upon BERT-Tiny using different training schemes, including ordinal classification, standard classification, and regression. Compared with the BERT-Base results in Table 2, we can see that the smaller BERT-Tiny model has lower prediction performance in almost every metric that we consider. In fact, the BERT-Tiny-based regressors (trained using MSE or L1 loss) seem to significantly underfit the training data and do not perform much better than random guesses. This is expected because the smaller BERT-Tiny architecture is less expressive and can hardly capture the subtle features from the input queries in natural languages.

Table 3: BERT-tiny-based output token length predictor evaluation results.

Metrics	Ord. CLS (MSE)	Ord. CLS (L1)	CLS	REG (MSE)	REG (L1)
Accuracy ↑	0.3947	0.4097	**0.4193**	0.2425	0.2425
F1 Score ↑	0.3121	0.3193	**0.3687**	0.0781	0.0781
L1 Error ↓	0.7752	**0.7558**	N/A	0.9528	0.9530
MSE ↓	**0.9656**	1.0398	N/A	1.8639	1.8643

Scheduling Performance Comparison In terms of the prediction latency overhead, the average, p99.9, and maximum inference latencies of the predictor built on top of BERT-Tiny are 1.8 ms, 3.7 ms, and 10.2 ms, respectively, less than that of the predictor built with BERT-Base. We then use predictors based on BERT-Base and BERT-Tiny proxy models in

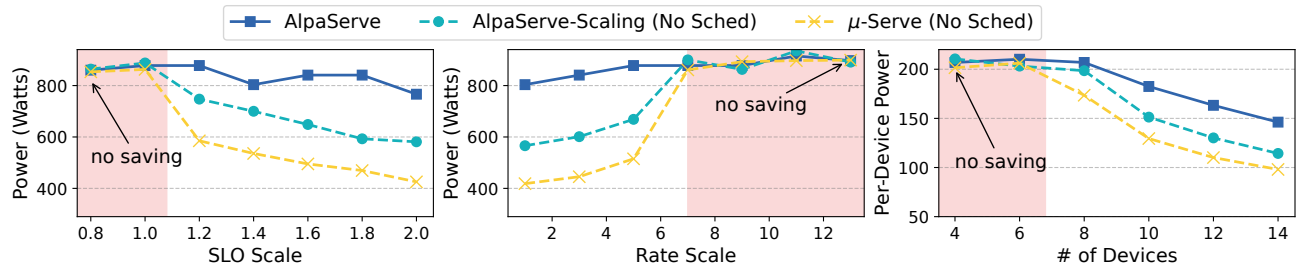


Figure 17: Power saving evaluation (without μ-Serve scheduler).

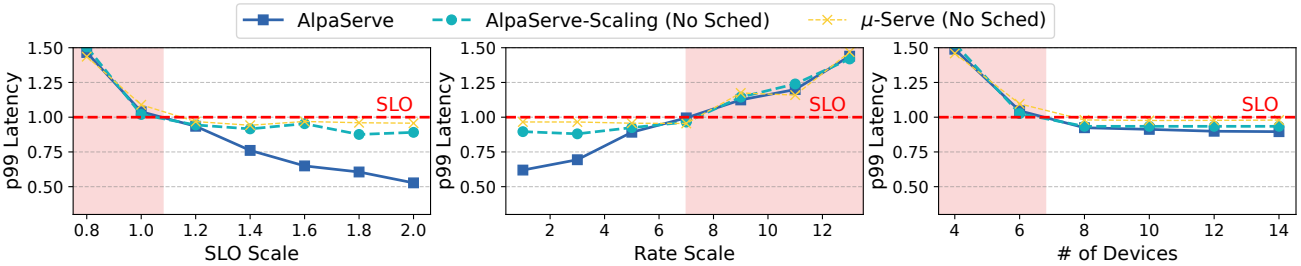


Figure 18: SLO preservation evaluation (without μ-Serve scheduler).

the μ-Serve scheduler and compare the scheduling performance regarding the job completion time (JCT) and throughput. Fig. 16 shows the comparison of the two proxy models when we use different training schemes: multi-class classification, ordinal classification with L1 loss and MSE, and regression with L1 loss and MSE. The scheduler with BERT-Base as the proxy model achieves 1–3.6% less JCT and 4.7–8.5% higher throughput. As expected, the smaller model incurs even less latency overhead but suffers worse prediction accuracy and scheduling performance in return. Therefore, we use BERT-Base as the proxy model in μ-Serve's scheduler.

A.3 Ablation Study on μ-Serve Scheduler

In this section, we conduct experiments to understand the role of the speculative shortest-job-first (SJF) scheduler (§3.3) in power saving. The results in §4.3 show an improvement of 38.6–41.3% in JCT and 2.2–2.5× in throughput. Using the same setup as in §4.2, we compare three methods: AlpaServe, AlpaServe-Scale (No Sched), and μ-Serve (No Sched). All three methods are implemented without μ-Serve scheduler, i.e., they are based on the default FCFS scheduler in AlpaServe. Fig. 17 and Fig. 18 show the comparison of power saving and SLO attainment of the three methods.

In terms of SLO attainment, without μ-Serve scheduler, there are more situations that the p99 latency is above the SLO latency as suggested by the wider red-shaded area compared to Fig. 8 (indicating worse SLO attainment). When there is an SLO attainment violation, online GPU device frequency scaling does not actuate any down-scaling. Even without SLO attainment violation, AlpaServe has a worse SLO attainment compared with AlpaServe-Sched (up to 31% lower p99 latency as shown in Fig. 8). Therefore, the power-saving opportunities are reduced.

Fig. 17 shows the power consumption when running the three approaches. At SLO attainment violations (i.e., in the red-shaded area), there is no power saving. When there are no SLO attainment violations, the power saving achieved by μ-Serve (No Sched) compared to AlpaServe is up to 1.8×, 1.92×, and 1.49× at varying SLO scales, rates, and numbers of devices (less than the improvement of 2.3×, 2.61×, and 2× when μ-Serve runs on its scheduler as shown in Fig. 7). This 21–27% reduction in power saving attributed to the missing of μ-Serve's scheduler justifies the role of the scheduler in μ-Serve empirically. AlpaServe-Scaling (No Sched) has up to 1.42×, 1.42×, and 1.28× power saving compared to AlpaServe at varying SLO scales, rates, and numbers of devices, respectively. Compared to the improvement of AlpaServe-Scaling on top of AlpaServe (as shown in Fig. 7), AlpaServe-Scaling (No Sched) also has less improvement (e.g., 1.42× < 1.7× improvement at varying rates).

Without the speculative SJF scheduler in μ-Serve, the power saving from μ-Serve and AlpaServe-Scaling becomes 21–27% less. The main reason is that replacing μ-Serve's scheduler with FCFS leads to worse SLO attainment and more SLO violations, and consequently less GPU frequency down-scaling (i.e., power saving) opportunities.

Potential Uses of μ-Serve Scheduler for Other Tasks. In this paper, we study the benefit of μ-Serve's scheduler in model-serving latency and throughput, and consequently the SLO attainment. Because of higher SLO attainment, there is more headroom for dynamically down-scaling GPU device

frequencies to save power without introducing SLO attainment violations. However, we anticipate that there can be more potential use cases and extensions of μ-Serve's proxy-model-based scheduler.

- *Optimization in model-serving latency or throughput.* A proxy-model-based model output length predictor can be used with existing model-serving systems (e.g., Clipper [8], TensorFlow-Serving [26], MArk [52], InferLine [7], Shepherd [53], and Clockwork [14]) to maximize the systems throughput/goodput or to support better latency-aware provision by avoiding head-of-line blocking and optimizing the waiting time of queued requests.
- *Optimized batching and GPU utilization.* Output token length prediction can be used to batch requests with similar execution lengths together to improve GPU utilization and prevent shorter requests from waiting for longer ones [51]. We present how μ-Serve scheduler works with various batching techniques in [32].
- *Optimization in load balancing.* Requests with longer predicted output lengths could be routed to more powerful or dedicated servers or clusters, while shorter requests can be handled by less resource-intensive instances.
- *Optimization in model-serving memory and key-value cache management.* The memory management system needs to accommodate a wide range of output lengths [20]. In addition, as the output length of a request grows at decoding, the memory required for its key-value cache also expands and may exhaust available memory for incoming requests or ongoing generation for existing requests. μ-Serve's prediction can potentially help make scheduling decisions, such as deleting or swapping out the key-value cache of some requests from GPU memory.