MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving

Jiangfei Duan [1 2] **Runyu Lu** [3] **Haojie Duanmu** [2 4] **Xiuhong Li** [5] **Xingcheng Zhang** [2] **Dahua Lin** [1 2]
Ion Stoica [6] **Hao Zhang** [7]

Abstract

Large language models (LLMs) have demonstrated remarkable performance, and organizations are racing to serve LLMs of varying sizes as endpoints for use-cases like chat, programming and search. However, efficiently serving multiple LLMs poses significant challenges for existing approaches due to varying popularity of LLMs. In the paper, we present MuxServe, a flexible spatial-temporal multiplexing system for efficient multiple LLM serving. The key insight behind is to colocate LLMs considering their popularity to multiplex memory resources, and leverage the characteristics of prefill and decoding phases to separate and flexibly colocate them to multiplex computation resources. MuxServe formally formulates the multiplexing problem, and proposes a novel placement algorithm and adaptive batch scheduling strategy to identify optimal colocations and maximize utilization. MuxServe designs a unified resource manager to enable flexible and efficient multiplexing. Evaluation results show that MuxServe can achieves up to $1.8\times$ higher throughput or processes $2.9\times$ more requests within 99% SLO attainment. The code is available at: `https://github.com/hao-ai-lab/MuxServe`.

1. Introduction

Recent advances in large language models (LLMs) are transforming the AI industry (Brown et al., 2020; Bommasani et al., 2021; Chowdhery et al., 2023). A variety of versions and scales of LLMs have been pretrained and fine-tuned for various use cases, such as chat, programming, and search. Many organizations, such as Google, OpenAI, Huggingface, are racing to serve these LLMs as endpoints to their users. However, the unprecedented capabilities of LLMs come at a significant inference cost – serving a single 175B LLM (Brown et al., 2020) requires eight A100 (80GB) GPUs; efficiently serving *multiple LLMs*, each catering to different group of users and needs, are even costlier and have emerged as a crucial and time-sensitive demand within the community, especially for LLM endpoint providers.

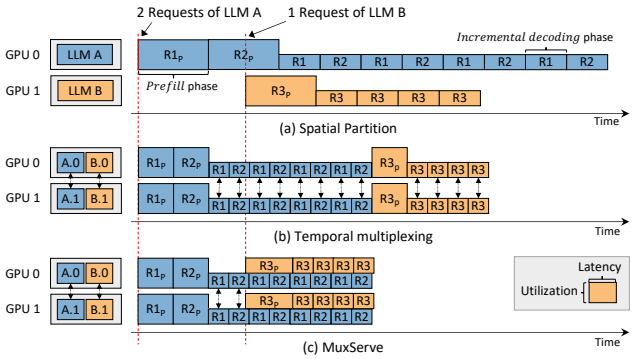

Figure 1. Three multiplexing strategies and GPU utilization of serving two LLMs on two GPUs.

To serve multiple LLMs with a cluster of resources, existing systems (Huggingface, 2023; NVIDIA, 2023a; Kwon et al., 2023) typically use *spatial partitioning* (Figure 1a), which involves allocating separate groups of GPUs for each LLM to accommodate their large model size and the key-value cache (KV cache) generated during inference. However, this spatial partition approach often leads to significant under-utilization of GPUs. Figure 2 shows real traffic observed by an LLM endpoint provider in 20 days: Different LLMs typically exhibit varying levels of popularity among users influenced by factors such as output quality, response speed, and usage patterns. Spatial partitioning disregards the varying popularity of different LLMs – LLMs with low arrival rates tend to receive sparse requests, resulting in idle GPUs for extended periods (as illustrated by GPU 1 in Figure 1a). Conversely, popular LLMs experience a substantial burden in handling incoming requests (GPU 0 in Figure 1a), leading to a potential performance bottleneck.

Another line of work explores temporal multiplexing (Fig-

[1]The Chinese University of Hong Kong [2]Shanghai AI Laboratory [3]Huazhong University of Science and Technology [4]Shanghai Jiao Tong University [5]Peking University [6]UC Berkeley [7]University of California San Diego. Correspondence to: Hao Zhang <haozhang@ucsd.edu>, Xiuhong Li <lixiuhong@pku.edu.cn>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

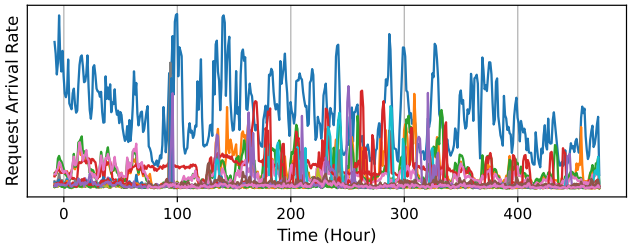

Figure 2. The dynamic request arrival rates of different LLMs over a 20 day period.

ure 1b) to serve multiple large models (Li et al., 2023), resulting in reduced serving latency in the presence of bursty workloads. This approach involves partitioning models onto a shared group of GPUs using intra- and inter-operator parallelism, and scheduling requests in an interleaved manner to share the computation and memory resources. However, this approach does not fully leverage the potential of GPUs when serving multiple LLMs, as it overlooks the unique characteristics of the *prefill* and *incremental decoding* phases of autoregressive LLMs. The *incremental decoding* phase, which typically plays a significant role in the inference process, often falls short in fully utilizing GPUs. Therefore, temporal multiplexing brings a wave-shaped utilization change, and most of the time it is in the trough (Figure 1b).

In this work, we explore to serve multiple LLMs with flexible *spatial-temporal multiplexing* to improve GPU utilization (Figure 1c) motivated by the following two key insights. Firstly, since *prefill* and *incremental decoding* phases have distinct computation characteristics, we separate them into different jobs and flexibly colocate prefill or decoding jobs from *different LLMs* to multiplex computation resources. Secondly, we colocate LLMs considering their popularity to multiplex memory resources and improve utilization. In Figure 1c, request of LLM B can be scheduled at its arrival since the *incremental decoding* phase of LLM A cannot fully utilize the GPUs. This flexible multiplexing allows MuxServe to finish all the requests in a shorter time, thus improving utilization.

We design and implement MuxServe to enable flexible and efficient spatial-temporal multiplexing for multiple LLM serving. Given a cluster configuration, a set of LLMs with workloads, MuxServe first formulates an optimization problem to search for the optimal colocations and batching and scheduling strategy (Section 3.1). To efficiently solve the problem, MuxServe proposes an enumeration-based greedy placement algorithm (Section 3.2) and adaptive batch scheduling algorithm (Section 3.3) to maximize utilization while ensuring fair sharing among LLMs. We discover that spatial-temporal partitioning is achieved by partitioning GPU SMs using CUDA MPS (NVIDIA, 2022b), and MuxServe designs a novel unified resource manager (Section 3.4) to enable efficient multiplexing. We finally evaluate MuxServe with both synthetic and real workload on a 32-GPU cluster. Evaluation results show that MuxServe achieves up to $1.8\times$ higher throughput compared to prior state-of-the-art systems.

In summary, MuxServe makes the following contributions,

1. The first to explore spatial-temporal multiplexing for LLM serving and formally formulate the problem.
2. A novel placement algorithm and adaptive batch scheduling strategy to determine the best collocations and maximize utilization.
3. A viable system design and implementation with unified resource manager to enable efficient multiplexing of LLMs and comprehensive evaluation.

2. Background and Motivation

2.1. LLM Inference

LLMs stack Transformer (Vaswani et al., 2017) blocks and each block consists of multi-head attention and feed-forward networks. Given input prompts, LLMs generate output tokens in autoregressive manner. The inference process includes two phases: *prefill* and *incremental decoding*. In *prefill* phase, LLMs process the entire prompt tokens in parallel and generate the first output token. Subsequently, in *decoding* phase, LLMs iteratively generate one output token at a time, building upon previously generated token. During inference, LLMs save key-value cache (KV cache) for each token, which dynamically increases as tokens are produced.

The two phases of LLM inference exhibit dinstict characteristics. The *prefill* phase, characterized by long input prompts, heavily utilizes computation resources, while the *incremental decoding* phase, with limited generated tokens, results in insufficient GPU utilization despite dominating the inference process due to the need to generate lengthy outputs for each prompt. For example, the average prompt and output length is 161 and 338 tokens in ShareGPT (ShareGPT-Team, 2023), respectively.

2.2. Distributed LLM Inference

Distributed inference is introduced to accommodate LLMs that cannot fit in a single GPU or accelerate inference process, and includes two categories. *Intra-operator parallelism* (Zheng et al., 2022) splits individual LLM layers across multiple GPUs and requires collective communications to transform input and output tensors during inference. It can significantly reduce inference latency, but also introduces additional communication overheads. *Inter-operator parallelism* partitions LLMs into multiple stages. Each stage is placed on one GPU with data dependency, and executed

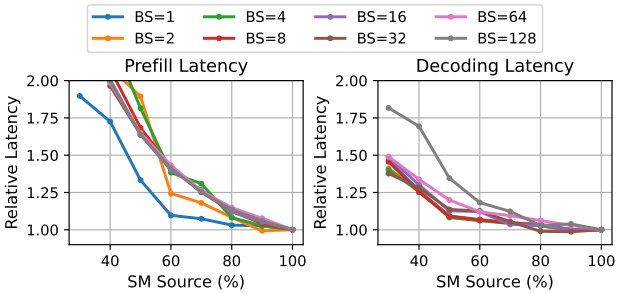

Figure 3. Relative batch inference latency as the fraction of computing resources assigned to LLaMA-7B changes from 30% to 100%. The input sequence length is 128.

in a pipeline fashion (i.e. pipeline parallelism (Narayanan et al., 2019)). Pipeline parallelism comes with negligible overhead, but does not reduce inference latency.

2.3. LLM Popularity Varies

Figure 2 displays the serving traffic of multiple LLMs over 20 days, as observed from an LLM endpoint provider. It is evident that the popularity varies significantly, and each LLM experiences distinct and changing arrival rates. Popular LLMs (blue line) consistently receive a considerably higher volume of serving traffic compared to other LLMs, resulting in higher resources demands. In contrast, less popular LLMs may exhibit consistently low arrival rates throughout the observed period, occupying fewer resources. This dynamic and diverse nature of request arrival rates emphasizes the need for a flexible and adaptive approach to efficiently serve multiple LLMs based on their individual popularity and demand, which would translate into significant cost reduction for LLM endpoint providers.

2.4. Multiplexing Opportunity on GPU

Spatial multiplexing is a resource sharing technique that splits the GPU resource (memory or/and SMs, i.e. Streaming Multiprocessors) into smaller partitions. Each partition is then allocated to perform different tasks simultaneously. *Temporal multiplexing* enables sharing of GPUs where each task occupies the entire computation resource during a specific time interval. With multiplexing, GPUs can handle multiple workloads concurrently, leading to increased efficiency and throughput. NVIDIA also offers Multi-Instance GPU (MIG) (NVIDIA, 2022a) to split memory and SMs into independent instances, and CUDA MPS (NVIDIA, 2022b) to partition SMs for different processes.

Prior works (Dhakal et al., 2020; Tan et al., 2021) have explored spatial multiplexing to serve multiple DNNs by assigning one DNN to a separate partition for enhanced performance. However, serving LLMs presents non-trivial challenges due to their unique characteristics. A significant difference lies in the memory bottleneck: *the huge memory requirements render previous approaches ineffective since it is unfeasible to hold multiple LLMs in a single GPU.*

To mitigate the memory bottleneck, AlpaServe (Li et al., 2023) involves parallelism to distribute several large models on multiple GPUs and utilizes temporal multiplexing to serve these models. However, temporal multiplexing ignores the characteristics of prefill and decoding phases of LLMs. As illustrated in Figure 3, when the amount of computation resources allocated to the dominant *decoding* phase is reduced, it does not lead to a substantial decrease in latency or throughput. Moreover, parallelization across multiple GPUs further reduces the computation requirements of LLMs. Temporal multiplexing thus results in significant resource under-utilization.

Recognizing the distinct resource requirements of *prefill* and *decoding* phases, we reveal that different LLMs can be colocated to multiplex the computation resources flexibly for improved efficiency and utilization. In the following sections, we will present MuxServe, the first system that explores spatial-temporal multiplexing for multiple LLM serving.

3. Method

3.1. Problem Formulation

Consider we have a cluster C and a set of LLMs M with workload W[1] to be served, one key insight MuxServe leveraged to improve system utilization is to colocate LLMs considering their popularity. LLMs with high request rates can be colocated with LLMs with low request rates to efficiently utilize resources. To achieve this, we introduce *LLM unit*, which refers to a group of LLMs that will be colocated together with the GPUs they are assigned. Our goal is to find the best group of *LLM units* B^* that maximize GPU utilization (i.e. throughput), hence the problem can be formulated as,

$$B^* = \arg\max_{B \in \mathcal{B}} \sum_{b \in B} \mathtt{F}(b, W_b) \tag{1}$$

where $\mathcal{B}$ represents all possible LLM units group, and $\mathtt{F}(\cdot, \cdot)$ estimates the throughput for a unit b with workload W_b.

Within an LLM unit, MuxServe leverages the insight that *prefill* and *decoding* phases of LLM inference exhibit distinct computation resources requirements, and splits them into *prefill* and *decoding* jobs. Each job occupies a fixed amount of SM resources and executes a prefill or decoding step for a batch requests of an LLM. Different jobs can be

[1] Suppose the workload is known. Otherwise, the workload can be estimated from history traffic since it changes slowly.

Algorithm 1 Enumeration-based Greedy LLM Placement

Input: LLM list M, cluster C, workload W
Output: The optimal group of LLM unit $best_units$
$\hat{M} \leftarrow$ llm_parallel_candidate(M, W) // Algorithm 2
$\mathcal{D} \leftarrow$ get_potential_device_mesh_groups(C, M)
$best_tpt, best_units \leftarrow 0, None$
for $D \in \mathcal{D}$ **do**
  // Greedily place LLMs on mesh group D
  M'=sort($\hat{M}$, key=$computation$, $descend$=$True$)
  for $m \in M'$ **do**
    $best_mesh, max_delta \leftarrow$ None, -1
    for $d \in D$ **do**
      u = make_unit(m, d)
      $delta$ = $\mathtt{F}(u, W_u)$ - $\mathtt{F}(d.u, W_{d.u})$
      if $delta > max_delta$ **then**
        $best_mesh, max_delta$ = $d, delta$
      end if
    end for
    $best_mesh.u$ = make_unit(m, $best_mesh$)
  end for
  tpt = sum($\mathtt{F}(d.u, W_{d.u})$ for $d \in D$)
  if $best_tpt < tpt$ **then**
    $best_tpt, best_units \leftarrow tpt$, [$d.u$ for $d \in D$]
  end if
end for

flexibly colocated to share the computation and memory resources. However, as there are multiple LLMs with distinct workload characteristics, different batching and scheduling strategies can lead to different throughputs, and different LLMs may also compete for resources. Therefore, given an LLM unit b that contains colocated LLMs b_{llm}, we need to find the optimal batching and scheduling strategy S that can maximize the throughput of the entire unit ,while ensuring fair resource sharing among LLMs within the unit. Therefore, the problem $\mathtt{F}(b, W_b)$ can be formulated as,

$$\begin{aligned} \mathtt{F}(b, W_b) = \max_{S} \sum_{m \in b_{llm}} \mathtt{tpt}_S(m, b, W_b) \quad s.t. \\ |\mathtt{R}(m_i, W_{m_i}) - \mathtt{R}(m_j, W_{m_j})| \leq \epsilon, \forall m_i, m_j \in b_{llm} \end{aligned} \tag{2}$$

where $\mathtt{tpt}_S(\cdot, \cdot, \cdot)$ estimates the throughput of an LLM m in the unit b using strategy S, $\mathtt{R}(\cdot, \cdot)$ estimates the normalized computation or memory resources consumption of an LLM m with workload W_m, and ϵ is a small number ensuring fairness. $\mathtt{R}(\cdot, \cdot)$ is normalized to account for varying LLM scales and popularity, since large and popular LLMs typically requires more resources.

Given the formulation above, we first introduce our placement algorithm to solve the problem (Equation (1)) in Section 3.2, which will maximize the intra-unit throughput (Equation (2)) with our batching and scheduling strategy (Section 3.3). Finally we describe our unified resource manager to enable efficient multiplexing in Section 3.4.

Algorithm 2 LLM Parallel Candidate Generation

Input: LLM list M, workload W
Output: The parallel candidate $\hat{M}$
$\hat{M} \leftarrow []$
for $m \in M$ **do**
  $sm_list \leftarrow$ get_potential_sm_list(m)
  $tp_list \leftarrow$ get_potential_tp_degree(m)
  for $p \in tp_list$ **do**
    for $num_sm \in$ sorted(sm_list) **do**
      $tpt, bs \leftarrow$ estimate_throughput(m, num_sm, p)
      if $tpt \geq W_m$ **then**
        $m.candidate$.add((p, num_sm, bs))
        break
      end if
    end for
  end for
  $\hat{M}$.append($m.candidate$)
end for

3.2. Placement Algorithm

Determining the optimal group of LLM units poses a challenging combinatorial optimization problem. As the number of devices and LLMs increases, the total number of possible LLM unit combinations grows exponentially. To solve Equation (1) efficiently, we design an enumeration-based greedy algorithm as outlined in Algorithm 1. The insight behind is to prioritize the placement selection for LLMs with large computation requirements, which considers both the model scale and popularity. With this algorithm, MuxServe can find a good solution efficiently.

In Algorithm 1, MuxServe first calls Algorithm 2 to generate all possible parallel candidates $\hat{M}$ considering the workload W. A parallel candidate refers to a configuration that meets the workload requirements while utilizing the fewest number of SMs. For each LLM m, MuxServe enumerates all possible combinations of configurations by varying the number of SMs and intra-operator parallelism degrees to find a set parallel candidates. For each intra-operator parallelism degree, MuxServe has one possible parallel candidate.

MuxServe then enumerates all potential device mesh groups to find the best LLM units. Each mesh comprises several GPUs that will be used to serve a set of LLMs concurrently. Given a device mesh group D and a list of parallel partition candidates $\hat{M}$, MuxServe greedily places LLMs on meshes to find the optimal group of LLM units. MuxServe prioritizes the placement selection for LLMs with large computation requirements to maximize serving utilization. For

Algorithm 3 Adaptive Batch Scheduling (ADBS)

Input: LLM list M
$prefill_waiting \leftarrow$ false
$quota \leftarrow$ init_token_block_quota(M)
while True **do**
  if no prefill jobs in execution **then**
    $prefill_waiting \leftarrow$ True
    $m \leftarrow$ round-robin a prefill job from M
    if resource_enough(m, $quota$) **then**
      execute_and_update(m, $quota$)
      $prefill_waiting \leftarrow$ False
    end if
  end if
  if not $prefill_waiting$ **then**
    $m \leftarrow$ round-robin a decoding job from M
    while resource_enough(m, $quota$) **do**
      execute_and_update(m, $quota$)
      $m \leftarrow$ round-robin a decoding job from M
    end while
  end if
  $quota =$ adapt_quota_periodically(M, $quota$)
end while

a specified LLM m, MuxServe iterates over all available meshes and approximates the expected increase in throughput with $\mathtt{F}(\cdot, \cdot)$. The LLM m is then placed on the mesh that yields the maximum throughput increase. This process is repeated for all LLMs. Subsequently, MuxServe estimates the serving throughput of the LLM units after the placement, and selects the LLM units that offer the best serving throughput as the optimal group of LLM units.

The complexity of Algorithm 1 is $O(MCD)$, where M is the number of LLMs, C is the number of devices and D is the number of potential device mesh groups. Given a large cluster, enumerating all possible device mesh groups can be slow. We prune the search space effectively incorporating the following heuristics: the intra-operator parallelism is typically adopted within a node, and workload imposes constraints on the possible mesh size.

3.3. Maximize Intra-unit Throughput

If there is only one LLM in a unit, the situation is reduced to single LLM serving, which has been extensively studied. However, when it comes to multiplexing multiple colocated LLMs with dynamically varying requests arrival times (Equation (2)), the solution is non-trivial due to the following challenges: requests for different LLMs cannot be batched together, LLMs in the unit have distinct workload characteristics, and different LLMs may compete for resources. It is impractical to find an optimal exact solution due to the complexity of the problem.

To address these challenges, we first define $\mathtt{R}(\cdot, \cdot)$ as the token block usage (Sheng et al., 2024) of an LLM, based on the observation that KV cache size poses a significant performance bottleneck for LLM serving. MuxServe assigns a token block quota to each LLM to ensure fair sharing. Counting token blocks provides a more intuitive way to consider the scale of LLMs, as tokens from different LLMs consume varying amounts of KV cache. Moreover, to consider variations in workload characteristics, $\mathtt{R}(\cdot, \cdot)$ is also normalized by request rates.

Then we maximize the intra-unit throughput by exploring *prefill-decoding* and *decoding-decoding* collocation. MuxServe prioritizes prefill jobs and fills remaining resources with decoding jobs. This is motivated by the observation that *decoding* jobs of a single LLM typically requires a few computation resources and can be batched together. Prioritizing prefill jobs can maximize the opportunity for colocation and batching.

Algorithm 3 describes our *adaptive batch scheduling* (ADBS) algorithm to maximize intra-unit throughput while ensuring fairness. ADBS first restricts the usage of token blocks by setting a quota for each LLM. In the main loop, if there is no prefill jobs in execution, ADBS employs a round-robin approach to select and execute a prefill job from served LLMs. If the resource is not enough, ADBS stops scheduling decoding jobs until resource capacity is met. Otherwise, ADBS schedules decoding jobs with round-robin until resource is not enough to maximize colocation.

To further improve utilization, ADBS adapts the token block quota for each LLM periodically. During runtime, MuxServe monitors the KV cache utilization. MuxServe identifies low-utilization LLMs and proactively transfers KV cache blocks from these LLMs to high-utilization LLMs. This dynamic reallocation of KV cache blocks ensures optimal utilization of resources and promotes efficient sharing among the LLMs within a unit.

ADBS approximates the solution of Equation (2) to maximize intra-unit throughput. But the concrete throughput $\mathtt{tpt}_S(\cdot, \cdot, \cdot)$ cannot be obtained without profiling. To address this, we build a simple yet effective analytical estimator to estimate the throughput of LLM m with

$$\mathtt{tpt}_S(m, b, W_b) = min\{\frac{b^m}{\sum_{i \in b} t_p^i + t_d^m \cdot l_o^m}, W_b\} \quad (3)$$

where t_p^m, t_d^m and l_o^m represent the prefill latency, decoding latency and average generation length of a batch requests with size b^m for LLM m, respectively. This formulation is based on the observation that prefill phases of different LLMs are generally executed sequentially and decoding phases can be executed concurrently, and different

phases are interleaved. Therefore, the latency of a batch requests is equal to the sum of all prefill phases of different LLMs and the decoding phases of the LLM. The prefill and decoding latency of different batches and request length can be profiled in advance. The average generation length can be estimated from requests history or specific dataset, ShareGPT (ShareGPT-Team, 2023) for instance. Given the request arrival rates, we can use binary search to find the batch size b that can satisfy the traffic. More details about the formulation please refers to Appendix A.2.

3.4. Resource Manager

After finding the optimal LLM units and determining the batching and scheduling strategy, MuxServe requires a new mechanism to support flexible and efficient spatial-temporal multiplexing of LLMs due to the following challenges: different prefill and decoding jobs need to flexibly share the computation resources, and share the weights and KV cache to reduce memory waste and fragmentation. To address these challenges, MuxServe proposes a unified resource manager for flexible and efficient multiplexing. Each LLM unit hosts a unified resource manager, and Figure 4 shows the overview of GPU resource management in an LLM unit.

The parallel runtime manages computation resources of an LLM unit in the granularity of SM based on NVIDIA MPS (NVIDIA, 2022b). MuxServe schedules prefill and decoding jobs from colocated LLMs with ADBS algorithm, then the parallel runtime dynamically assigns SMs to each job at runtime rather than statically allocating. This enables MuxServe to flexibly multiplex the execution of different LLMs. As illustrated in the right part of Figure 4, the SMs are all allocated to one job at step 1 due to its large computation intensity. After the execution, MuxServe schedules two jobs that can be executed concurrently at step 2 to share the SM resources.

The prominent challenge lies in sharing the memory resources among different jobs is to reduce memory waste and fragmentation. LLM weights and KV cache consume a huge amount of memory and need to be shared among jobs. Furthermore, KV cache increases dynamically, and different LLMs posses varying sizes of KV cache due to differences in the number of attention heads, hidden layers, and hidden sizes.

To efficiently share memory resources, MuxServe divides them into three partitions. The first partition is a unified KV cache space enabled by our head-wise cache. Leveraging the observation that the size of each attention head is often consistent or has limited variations across different LLMs, for example LLaMAs (Touvron et al., 2023) and GPT-3 (Brown et al., 2020) all use 128. MuxServe divides the KV cache table into small blocks, and each block holds the KV cache of one head for several tokens. This head-wise granularity enables MuxServe to accommodate the KV cache of different LLMs in a unified space to share the memory. To reduce redundancy, the second partition stores a single replica of LLM weights that can be shared among prefill and decoding jobs. The final partition reserves space for activation, which is utilized during inference runtime.

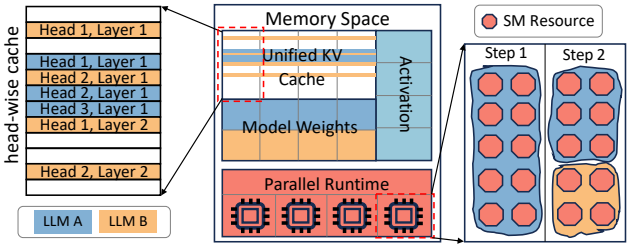

Figure 4. Overview of GPU resource management in an LLM unit. The memory is divided into 3 partitions to store KV cache, weights and activations, respectively. The parallel runtime partitions SM dynamically to different jobs.

MuxServe adopts a unified KV cache instead of reserving separate KV cache for each LLM. This shared cache enables MuxServe to dynamically adjust the cache allocation during runtime with minimal overhead. As a result, MuxServe can handle bursty and changing workload better. Notably, vLLM (Kwon et al., 2023) proposes Paged Attention to improve memory utilization for single LLM serving, while our unified KV cache addresses a distinct scenario, where multiple LLMs of varying sizes, popularities, and configurations need to share the cache.

Implementation. MuxServe is built atop vLLM (Kwon et al., 2023), an efficient single LLM serving system based on PyTorch (Paszke et al., 2019), and utilizes NVIDIA MPS (NVIDIA, 2022b) to partition SM resources. The major component of MuxServe is implemented in a global scheduler, which runs on each LLM unit. The global scheduler manages a request queue and runs the ADBS algorithm to schedule requests from the queue. MuxServe disaggregates the prefill and decoding phases and launches separate vLLM processes, configured with different number of SM resources with NVIDIA MPS, as runtime engines for them. The global scheduler schedules prefill or decoding jobs to runtime engine processes via python mutiprocessing shared memory. MuxServe additionally runs a memory manager process to manage the unfied memory space. The runtime engines access to the memory space via CUDA IPC handler. When the runtime engine needs to allocate some KV cache for execution, it requests new cache blocks from the memory manager and fills the KV cache to the assigned blocks. After the execution of prefill jobs, the memory manager maintains the allocated KV cache blocks for later usage of corresponding decoding jobs. The KV cache will be released only when the request is finished. We implement the memory manager with C++ to optimize the overhead of

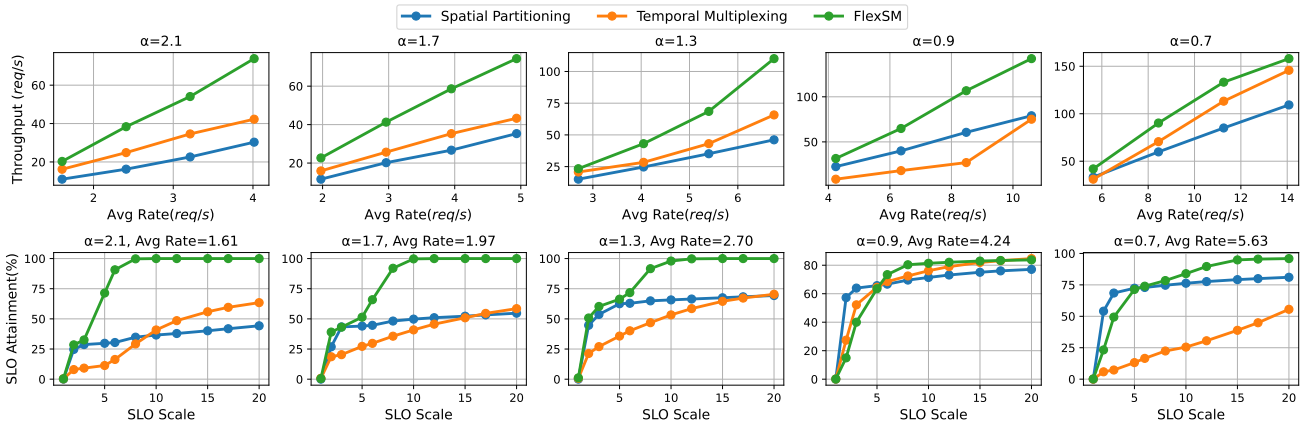

Figure 5. Throughput and SLO attainment on synthetic workloads.

Table 1. The number of LLMs to be served in different sizes.

Scale	4B-8B	8B-21B	21B-41B	41B-70B
#LLMs	12	4	2	1

block allocation.

4. Evaluation

In this section, we evaluate MuxServe on both synthetic and real workloads. We also perform ablation studies to verify the effectiveness of individual components.

4.1. Experimental Setup

Cluster. We conduct experiments on a 4 node cluster, each is equipped with 8 NVIDIA A100 (80GB) GPUs. The intra-node connection is 600GB/s NVLink, and the inter-node connection is 200Gbps IB.

Metrics. We use the *aggregated throughput* as our evaluation metric since our target is to evaluate the GPU utilization. Since different LLMs have different arrival rates, we use the rate to compute a weighted average of throughput. We also evaluate the *SLO attainment* to study MuxServe's effect on latency. SLO attainment measures the percentage of requests that can be finished within the latency target. In this paper, we scale the latency to different multiples of single device execution latency (i.e. SLO scale).

Baselines. We compare MuxServe with two baselines. The first is widely used spatial partitioning, which serves each LLM separately on a group of GPUs. We serve each LLM with vLLM (Kwon et al., 2023), a state-of-the-art serving framework. Another baseline is temporal multiplexing similar to AlpaServe (Li et al., 2023). Since AlpaServe does not support multiplexing of multiple LLMs, we implement this baseline by ourselves with our unified KV cache. For temporal multiplexing, we colocate LLMs with the placement optimized by our placement algorithm, schedule LLMs with first-come-first-serve in a temporal manner, and batch the requests of each LLM with continuous batching.

4.2. End-to-End Results for Synthetic Workloads

Models. LLaMA (Touvron et al., 2023) is the most popular LLM architecture. According to the sizes of LLaMA models, the LLMs can be divided into 4 size buckets. Table 1 shows the number of LLMs to be served in different sizes.

Workloads. For synthetic workloads, we first generate request rates for each LLM using power-law distribution with an exponent α, then sample the arrival time of each request with poisson processes. The requests are sampled from ShareGPT. We vary α and rate scales to evaluate a diverse workloads. For each α, we first set the maximal request rate for each LLM to 20 req/s, and then scale up the max rate and average rate for evaluation. The α decides the popularity of LLMs, and a larger α means that the fewer LLMs are more popular and receive a higher rates. Figure 6 shows the LLM traffic distribution as we vary α. Typically, $\alpha = 0.9$ or 2.1 represent 20% LLMs receives 50% or 90% of the total request rates.

Results. Figure 5 shows the throughput and SLO attainment with varying α and average rates. The throughput of MuxServe outperforms two baselines in all scenarios, achieving up to $1.8\times$ improvement. MuxServe can also process up to $2.9\times$ more requests within 99% SLO attainment. When α is small, the request rates are more even and MuxServe can efficiently colocate *prefill-decoding* and *decoding-decoding* jobs to improve utilization. But the interference of colocation also brings some overhead, leading to a slightly lower SLO attainment with small SLO scale. With a larger α, popular LLMs can be colocated with unpopular LLMs to multiplex memory resources, thus achieving a higher throughput with more SMs and larger KV caches. Popular LLMs can process more requests thus achieving a

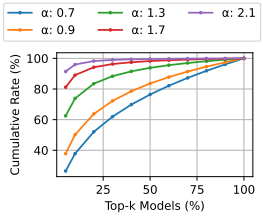

Figure 6. Cumulative rate distribution as we vary α.

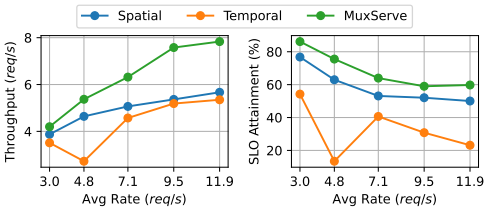

Figure 7. Throughput and SLO attainment as we vary the rates on real workloads.

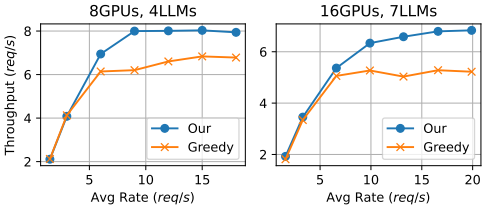

Figure 8. Ablation study of placement algorithm.

higher SLO attainment. The results also demonstrate that MuxServe can achieve much more significant improvement when the popularity among LLMs is more different (i.e. α is larger). The detailed results of P99 latency please refer to Appendix A.1.

4.3. End-to-End Results for Real Workloads

To evaluate MuxServe on real workloads, we sample LLMs and workloads from ChatLMSYS trace and rescale the rates to evaluate MuxServe. ChatLMSYS is a web application that serves multiple LLMs of different scales. In this real workload trace, we serve 16 LLMs with 32 GPUs, and 20% popular LLMs get 50% request traffic. Figure 7 shows the throughput and SLO attainment under SLO scale 8. MuxServe achieves up to $1.38\times$ and $1.46\times$ higher throughput compared with spatial partitioning and temporal multiplexing, respectively. As we vary the average rates, MuxServe always achieves a higher SLO attainment. When the average rate is 4.8, several LLMs with medium rates are co-located on a large mesh. Temporal multiplexing cannot efficient multiplex these LLMs thus performing quite bad.

4.4. Ablation Studies

In this section, we study the effectiveness of our proposed approaches: placement algorithm in Section 3.2, *adaptive batch scheduling* (ADBS) mechanism in Section 3.3 and unified resource manager in Section 3.4.

Effectiveness of placement algorithm. To show the effectiveness of our placement algorithm, we conduct a comparison with a greedy algorithm. The greedy algorithm prioritizes the placement of LLM with high arrival rates and greedily assigns it to the mesh with the largest available free memory. The ablation study is conducted on two scales: 8 GPUs with 4 LLMs and 16 GPUs with 7 LLMs. For each scenario, 50% LLMs are popular and occupy more than 70% request traffic. The request arrivals follow poisson distribution. As shown in Figure 8, our placement algorithm achieves $1.3\times$ higher throughput compared with greedy algorithm in the right subfigure.

The optimized placement also verifies our insight to prioritize the placement selection for LLMs with large computation requirements. In the 8 GPUs and 4 LLMs case, MuxServe colocates two popular small LLMs with one less popular small LLM on 4 GPUs, while allocating 4 GPUs for the less popular large LLM. In the 16 GPUs and 7 LLMs case, MuxServe colocates two large LLMs with low arrival rates on 4 GPUs, and splits the rest (majority) of GPUs into groups of 2 or 4 GPUs. FlexSM places the other 5 LLMs with high arrival rates on different groups considering their computation demands. To understand this, consider the contrary: if we prioritize the placement selection for LLMs with large memory requirements (i.e. try to balance memory consumption across GPUs). Large LLMs would be prioritized to be placed on large GPU groups without considering their popularity. Then small LLMs will be placed in the manner that tries to fit in the remaining GPU memory space. This is counterintuitive since popular small LLMs could need

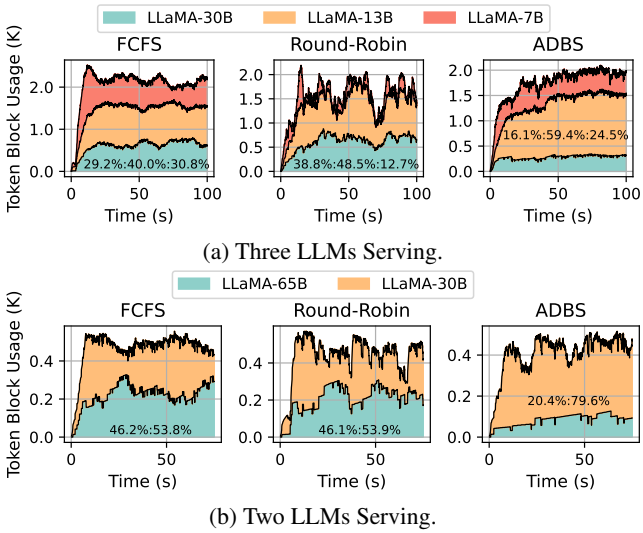

Figure 9. Comparison of cache usage of different schedule approaches on 4 GPUs. The relative proportion of token block usage is annotated in the figure. FCFS: First Come First Serve, ADBS: ADaptive Batch Size. (a) Request rate: 2:8:8 req/s. Throughput (req/s): FCFS (3.8), Round-Robin (4.1), ADBS (6.2). (b) Request rate: 1:8 req/s. Throughput (req/s): FCFS (3.2), Round-Robin (4.9), ADBS (6.6).

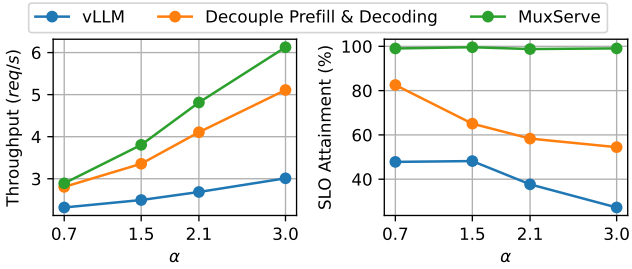

Figure 10. Ablation study of the unified resource manager.

massive GPUs to fulfill the request traffic requirements.

Effectiveness of ADBS. We compare MuxServe's ADBS to First Come First Serve (FCFS) and Round-Robin in two serving settings to verify the effectiveness (Figure 9). In Figure 9a, the average request length is $2:1:1$ for LLaMA-30B, 13B, and 7B, respectively. In Figure 9b, the average request length is $4:1$ for LLaMA-65B and 30B, respectively. In both scenarios, the token block usage of ADBS is more closely aligned with the distribution of arrival rates, thus achieving a *fairer memory resource sharing*. ADBS also achieves $1.43\times$ and $1.85\times$ higher throughput compared with Round-Robin and FCFS scheduling, since the unfair sharing of KV cache blocks hurts the performance of popular LLMs. In addition, FCFS cannot efficiently multiplexing different LLMs.

Effectiveness of resource manager. We study the effectiveness of MuxServe's unified resource manager by gradually enabling computation and memory management. We serve 4 LLMs on 4 GPUs and generate arrival rates using power law distribution. Figure 10 shows the throughput and SLO attainment (SLO scale 8) as we vary α. By separating prefill and decoding phases with computation resource management, the throughput improves $1.7\times$. With a unified memory manager, MuxServe achieves another $1.2\times$ higher throughput and improves SLO attainment by $3.6\times$.

5. Related Work

DL serving systems. A wide variety of deep learning serving systems have been proposed to improve the serving efficiency (Olston et al., 2017; NVIDIA, 2023a). Recent works (Crankshaw et al., 2017; Shen et al., 2019; Gujarati et al., 2020; Zhang et al., 2023; Romero et al., 2021) utilize temporal multiplexing and introduce better batching and scheduling strategies to improve GPU utilization and meet SLO target. These approaches focus on small DNN models and ignores the parallelism needed in serving LLMs. A more related work is AlpaServe (Li et al., 2023), which explores the space of parallelism and serves multiple large DNN models with temporal multiplexing. However, AlpaServe is not designed for LLMs and misses the characteristics of LLM inference phases.

LLM serving systems. Recent years, the outstanding performance of LLMs has aroused strong interests in LLM serving systems. Some prior approaches customize GPU kernels to optimize transformer model inference, for example TensorRT-LLM (NVIDIA, 2023b) and LightSeq (Wang et al., 2021b). Recent works such as FasterTransformer (NVIDIA, 2021), DeepSpeed-Inference (Aminabadi et al., 2022), vLLM (Kwon et al., 2023) and TGI (Huggingface, 2023) incorporate intra- and inter-operator parallelism to accelerate LLM inference on multiple GPUs. In addition, memory management (Kwon et al., 2023), offloading (Sheng et al., 2023), iteration-level batching (Yu et al., 2022), speculative decoding (Miao et al., 2023) and cheap instances (Miao et al., 2024) have been introduced to enhance the throughput and reduce the cost of LLM serving. MuxServe is orthogonal to these works since they focus on optimizing single LLM inference.

GPU sharing. GPU sharing mainly can be categorized into temporal (Lim et al., 2021; Wang et al., 2021a; Xiao et al., 2020) and spatial sharing (Tan et al., 2021; Zhao et al., 2023; Han et al., 2022). Salus (Yu & Chowdhury, 2019) proposes fast job switching and memory management to facilitate temporal sharing. NIVIDIA MIG (NVIDIA, 2022a) and MPS (NVIDIA, 2022b) are native support to multiplex jobs on GPUs. GSLICE (Dhakal et al., 2020) proposes a dynamic GPU resource allocation and management framework to improve utilization. To overcome the inefficiency of temporal or spatial sharing, Gpulet (Choi et al., 2022) introduces mixed spatial-temporal sharing to multiplexing jobs. These works target on multiplexing small DNN jobs, while MuxServe explores flexible spatial-temporal multiplexing in emerging LLM serving application.

6. Conclusion

In this paper, we introduce MuxServe, a flexible and efficient spatial-temporal multiplexing system to serve multiple LLMs concurrently. MuxServe colocates LLMs considering their popularity and colocates prefill and decoding jobs leveraging their characteristics to improve GPU utilization.

Acknowledgements

We thank Chang Chen, Qianchao Zhu and Jinming Ma for their insightful review and discussions. The research is supported in part by Shanghai AI Laboratory, CUHK Interdisciplinary AI Research Institute, and the Centre for Perceptual and Interactive Intelligence (CPII) Ltd under the Innovation and Technology Commission (ITC)'s InnoHK.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., and He, Y. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022. ISBN 9784665454445.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R. B., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., Brynjolfsson, E., Buch, S., Card, D., Castellon, R., Chatterji, N. S., Chen, A. S., Creel, K., Davis, J. Q., Demszky, D., Donahue, C., Doumbouya, M., Durmus, E., Ermon, S., Etchemendy, J., Ethayarajh, K., Fei-Fei, L., Finn, C., Gale, T., Gillespie, L., Goel, K., Goodman, N. D., Grossman, S., Guha, N., Hashimoto, T., Henderson, P., Hewitt, J., Ho, D. E., Hong, J., Hsu, K., Huang, J., Icard, T., Jain, S., Jurafsky, D., Kalluri, P., Karamcheti, S., Keeling, G., Khani, F., Khattab, O., Koh, P. W., Krass, M. S., Krishna, R., Kuditipudi, R., and et al. On the opportunities and risks of foundation models. *CoRR*, abs/2108.07258, 2021. URL https://arxiv.org/abs/2108.07258.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

Choi, S., Lee, S., Kim, Y., Park, J., Kwon, Y., and Huh, J. Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 199–216, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-29-53. URL https://www.usenix.org/conference/atc22/presentation/choi-seungbeom.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113, 2023. URL http://jmlr.org/papers/v24/22-1144.html.

Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 613–627, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

Dhakal, A., Kulkarni, S. G., and Ramakrishnan, K. K. GSLICE: controlled spatial sharing of gpus for a scalable inference platform. In Fonseca, R., Delimitrou, C., and Ooi, B. C. (eds.), *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pp. 492–506. ACM, 2020. doi: 10.1145/3419111.3421284. URL https://doi.org/10.1145/3419111.3421284.

Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., and Mace, J. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 443–462. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/gujarati.

Han, M., Zhang, H., Chen, R., and Chen, H. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 539–558, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/han.

Huggingface. Text generation inference.

https://github.com/huggingface/text-generation-inference, 2023.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In Flinn, J., Seltzer, M. I., Druschel, P., Kaufmann, A., and Mace, J. (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Li, Z., Zheng, L., Zhong, Y., Liu, V., Sheng, Y., Jin, X., Huang, Y., Chen, Z., Zhang, H., Gonzalez, J. E., and Stoica, I. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. In Geambasu, R. and Nightingale, E. (eds.), *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pp. 663–679. USENIX Association, 2023. URL https://www.usenix.org/conference/osdi23/presentation/li-zhouhan.

Lim, G., Ahn, J., Xiao, W., Kwon, Y., and Jeon, M. Zico: Efficient GPU memory sharing for concurrent DNN training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 161–175. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/lim.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Wang, Z., Wong, R. Y. Y., Zhu, A., Yang, L., Shi, X., Shi, C., Chen, Z., Arfeen, D., Abhyankar, R., and Jia, Z. Specinfer: Accelerating generative large language model serving with speculative inference and token tree verification, 2023.

Miao, X., Shi, C., Duan, J., Xi, X., Lin, D., Cui, B., and Jia, Z. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 1112–1127, 2024.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 1–15, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646.

NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2021.

NVIDIA. Nvidia mig. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2022a.

NVIDIA. Multi-process service. https://docs.nvidia.com/deploy/mps/index.html, 2022b.

NVIDIA. Triton inference server: An optimized cloud and edge inferencing solution. https://github.com/triton-inference-server, 2023a.

NVIDIA. Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM, 2023b.

Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., and Soyke, J. Tensorflow-serving: Flexible, high-performance ML serving. *CoRR*, abs/1712.06139, 2017. URL http://arxiv.org/abs/1712.06139.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 8024–8035, 2019.

Romero, F., Li, Q., Yadwadkar, N. J., and Kozyrakis, C. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 397–411. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/romero.

ShareGPT-Team. Sharegpt. https://sharegpt.com/, 2023.

Shen, H., Chen, L., Jin, Y., Zhao, L., Kong, B., Philipose, M., Krishnamurthy, A., and Sundaram, R. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 322–337, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359658. URL https://doi.org/10.1145/3341301.3359658.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single GPU. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 31094–31116. PMLR, 2023. URL https://proceedings.mlr.press/v202/sheng23a.html.

Sheng, Y., Cao, S., Li, D., Zhu, B., Li, Z., Zhuo, D., Gonzalez, J. E., and Stoica, I. Fairness in serving large language models. *CoRR*, abs/2401.00588, 2024. doi: 10.48550/ARXIV.2401.00588. URL https://doi.org/10.48550/arXiv.2401.00588.

Tan, C., Li, Z., Zhang, J., Cao, Y., Qi, S., Liu, Z., Zhu, Y., and Guo, C. Serving DNN models with multi-instance gpus: A case of the reconfigurable machine scheduling problem. *CoRR*, abs/2109.11067, 2021. URL https://arxiv.org/abs/2109.11067.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023. doi: 10.48550/ARXIV.2302.13971. URL https://doi.org/10.48550/arXiv.2302.13971.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017.

Wang, G., Wang, K., Jiang, K., Li, X., and Stoica, I. Wavelet: Efficient DNN training with tick-tock scheduling. In Smola, A., Dimakis, A., and Stoica, I. (eds.), *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org, 2021a.

Wang, X., Xiong, Y., Wei, Y., Wang, M., and Li, L. Lightseq: A high performance inference library for transformers. In Kim, Y., Li, Y., and Rambow, O. (eds.), *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers, NAACL-HLT 2021, Online, June 6-11, 2021*, pp. 113–120. Association for Computational Linguistics, 2021b. doi: 10.18653/V1/2021.NAACL-INDUSTRY.15. URL https://doi.org/10.18653/v1/2021.naacl-industry.15.

Xiao, W., Ren, S., Li, Y., Zhang, Y., Hou, P., Li, Z., Feng, Y., Lin, W., and Jia, Y. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 533–548. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/xiao.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Yu, P. and Chowdhury, M. Salus: Fine-grained GPU sharing primitives for deep learning applications. *CoRR*, abs/1902.04610, 2019. URL http://arxiv.org/abs/1902.04610.

Zhang, H., Tang, Y., Khandelwal, A., and Stoica, I. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 787–808, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL https://www.usenix.org/conference/nsdi23/presentation/zhang-hong.

Zhao, Y., Liu, X., Liu, S., Li, X., Zhu, Y., Huang, G., Liu, X., and Jin, X. Muxflow: Efficient and safe GPU sharing in large-scale production deep learning clusters. *CoRR*, abs/2303.13803, 2023. doi: 10.48550/ARXIV.2303.13803. URL https://doi.org/10.48550/arXiv.2303.13803.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., Gonzalez, J. E., and Stoica, I. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In Aguilera, M. K. and Weatherspoon, H. (eds.), *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pp. 559–578. USENIX Association, 2022. URL https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.

A. Appendix

A.1. P99 Latency Comparison

Figure 12 shows the P99 average latency, TPOT (Time-Per-Output-Token) and TTFT (Time-To-First-Token) latency of three multiplexing approaches on synthetic workloads. The results demonstrate that MuxServe can achieve a lower P99 average latency compared with prior approaches. The P99 TPOT latency of MuxServe is a little bit higher than spatial partitioning due to interference, and significantly lower compared wth temporal multiplexing. The P99 TTFT latency of MuxServe is lower than both spatial partitioning and temporal multiplexing since MuxServe can significantly reduces queuing time. When α is large, the P99 TTFT latency is dominated by the most popular model. Temporal multiplexing can reduces the queuing time, thus obtaining a lower P99 TTFT compared with spatial partitioning.

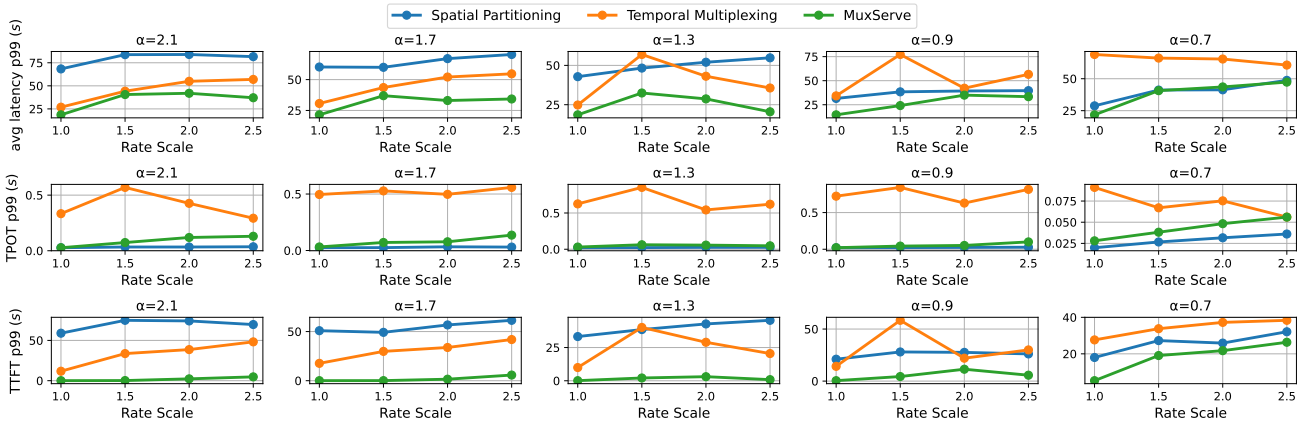

Figure 11. The P99 average latency, TPOT (Time-Per-Output-Token) and TTFT (Time-To-First-Token) latency on synthetic workloads.

A.2. Throughput Estimator

Figure 12 shows an approximated execution timeline of MuxServe, which motivates the formulation of Equation (3). In a stable serving setting, requests arrive at a fixed interval. The prefill requests are executed one by one, with their decoding steps overlapping. Since the system is stable, it completes some requests every interval and receives new requests. Therefore, we can estimate the execution time of a stable batch b^m as $\sum_{i \in b} t^i p + t^m d$. Hence we can estimate the throughput with Equation (3).

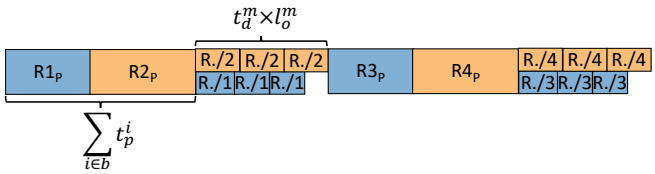

Figure 12. Execution timeline of MuxServe in a stable serving setting. $R.$ represents other requests in the batch.