S-LoRA: Serving Thousands of Concurrent LoRA Adapters

Ying Sheng [*1 2] Shiyi Cao [*1] Dacheng Li [1] Coleman Hooper [1] Nicholas Lee [1] Shuo Yang [1 3]
Christopher Chou [1] Banghua Zhu [1] Lianmin Zheng [1] Kurt Keutzer [1] Joseph E. Gonzalez [1] Ion Stoica [1]

Abstract

The "pretrain-then-finetune" paradigm is commonly adopted in the deployment of large language models. Low-Rank Adaptation (LoRA), a parameter-efficient fine-tuning method, is often employed to adapt a base model to a multitude of tasks, resulting in a substantial collection of LoRA adapters derived from one base model. We observe that this paradigm presents significant opportunities for batched inference during serving. To capitalize on these opportunities, we present S-LoRA, a system designed for the scalable serving of many LoRA adapters. S-LoRA stores all adapters in the main memory and fetches the adapters used by the currently running queries to the GPU memory. To efficiently use the GPU memory and reduce fragmentation, S-LoRA proposes Unified Paging. Unified Paging uses a unified memory pool to manage dynamic adapter weights with different ranks and KV cache tensors with varying sequence lengths. Additionally, S-LoRA employs a novel tensor parallelism strategy and highly optimized custom CUDA kernels for heterogeneous batching of LoRA computation. Collectively, these features enable S-LoRA to serve thousands of LoRA adapters on a single GPU or across multiple GPUs with a small overhead. Compared to state-of-the-art libraries such as HuggingFace PEFT and vLLM (with naive support of LoRA serving), S-LoRA can improve the throughput by up to 4 times and increase the number of served adapters by several orders of magnitude. As a result, S-LoRA enables scalable serving of many task-specific fine-tuned models and offers the potential for large-scale customized fine-tuning services. The code is available at https://github.com/S-LoRA/S-LoRA.

1 Introduction

Large language models (LLMs) have become ubiquitous in modern applications, ranging from natural language processing to more general tasks (OpenAI, 2023; Touvron et al., 2023b; Alayrac et al., 2022). Within these domains, LLMs have consistently demonstrated superior performance, especially when fine-tuned for specific tasks (Kenton & Toutanova, 2019; Houlsby et al., 2019; Ouyang et al., 2022). This "pretrain-then-finetune" paradigm has led to the proliferation of numerous fine-tuned variants of a single base LLM, each tailored to a specific task or domain.

When scaling the fine-tuning of a base model for numerous tasks, such as personalized assistants, which could involve thousands or millions of users, the associated training and serving costs can become substantial. To address this, several parameter-efficient fine-tuning methods have been developed. A prime exemplar is Low-Rank Adaptation (LoRA) (Hu et al., 2021), which enables efficient fine-tuning by updating only low-rank additive matrices. These matrices consist of a small number of parameters, referred to as adapter weights. LoRA has shown that by fine-tuning just these adapter weights, it is possible to achieve performance on par with full-weight fine-tuning. However, despite considerable research into fine-tuning, the question of how to serve these fine-tuned variants at scale remains unexplored.

One of the key innovations in the LoRA paper was the elimination of adapter inference latency by directly merging the adapter with the model parameters. Additionally, to support multiple models on a single machine, the same paper proposes swapping adapters by adding and subtracting LoRA weights from the base model. While this approach enables low-latency inference for a single adapter and serial execution across adapters, it significantly reduces overall serving throughput and increases total latency when serving multiple adapters concurrently. Moreover, the paper does not consider the opportunity to leverage host memory to increase the number of adapters hosted by a single machine.

In this paper, we study how to scalably serve thousands of LoRA adapters on a single machine. We observe that the shared base model, which underpins numerous LoRA adapters, presents a substantial opportunity for batched inference. To achieve high-throughput multi-adapter serving,

[*]Equal contribution. Part of the work was done when Ying was visiting UC Berkeley. [1]UC Berkeley [2]Stanford University [3]Shanghai Jiao Tong University. Correspondence to: Ying Sheng <ying1123@stanford.edu>, Shiyi Cao <shicao@berkeley.edu>.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2024.

it is advantageous to separate the batchable base model computation from individual LoRA computations.

While leveraging batching in the base model is straightforward (as all queries share the base model), extending batching to the adapters is challenging. First, serving many LoRA adapters simultaneously requires efficient memory management. Since GPU memory is limited, we must store adapter weights outside the GPU and dynamically fetch them when needed. However, dynamically loading and unloading adapters of varying sizes, coupled with the dynamic allocation and deallocation of KV cache tensors for requests with different sequence lengths, can lead to significant memory fragmentation and I/O overhead. Second, apart from the easily batchable base model computation, the separated computation of many adapters with distinct ranks in non-contiguous memory is challenging to batch and demands the development of new computation kernels. Third, leveraging multiple GPUs on a single machine requires novel parallelism strategies to accommodate the added LoRA weights and computations. It is essential to carefully design this strategy to minimize communication and memory overheads.

To this end, we introduce S-LoRA, a scalable LoRA serving system. S-LoRA exploits batching opportunities, efficiently manages both host and GPU memory, and orchestrates parallelism across multiple GPUs. The primary contributions of S-LoRA are summarized as follows:

- *Unified Paging*: To reduce memory fragmentation and increase batch size, S-LoRA introduces a unified memory pool. This pool manages dynamic adapter weights and KV cache tensors by a unified paging mechanism.
- *Heterogeneous Batching*: To minimize the latency overhead when batching different adapters of varying ranks, S-LoRA employs highly optimized custom CUDA kernels. These kernels operate directly on non-contiguous memory and align with the memory pool design, facilitating efficient batched inference for LoRA.
- *S-LoRA TP*: To ensure effective parallelization across multiple GPUs, S-LoRA introduces a novel tensor parallelism strategy. This approach incurs minimal communication cost for the added LoRA computation compared to that of the base model. This is realized by scheduling communications on small intermediate tensors and fusing the large ones with the communications of the base model.

We evaluate S-LoRA by serving Llama-7B/13B/30B/70B. Results show that S-LoRA can serve thousands of LoRA adapters on a single GPU or across multiple GPUs with a small overhead. When compared to the state-of-the-art parameter-efficient fine-tuning library, Huggingface PEFT, S-LoRA can enhance throughput by up to 30×. In comparison to the high-throughput serving system vLLM using a naive support of LoRA serving, S-LoRA can improve throughput by up to 4× and increase the number of served adapters by several orders of magnitude.

2 Background

Low-Rank Adaptation (LoRA) (Hu et al., 2021) is a parameter-efficient fine-tuning method designed to adapt pre-trained large language models to new tasks. The motivation behind LoRA stems from the low intrinsic dimensionality of model updates during adaptation. In the training phase, LoRA freezes the weights of a pre-trained base model and adds trainable low-rank matrices to each layer. This approach significantly reduces the number of trainable parameters and memory consumption. When compared to full parameter fine-tuning, LoRA can often reduce the number of trainable parameters by orders of magnitude (e.g., 10000×) while retaining comparable accuracy. For the inference phase, the original paper suggests merging the low-rank matrices with the weights of the base model. As a result, there is no added overhead during inference, setting it apart from previous adapters like (Houlsby et al., 2019) or prompt tuning methods such as (Lester et al., 2021).

Formally, for a pre-trained weight matrix $W \in \mathbb{R}^{h \times d}$, LoRA introduces the update as $W' = W + AB$, where $A \in \mathbb{R}^{h \times r}$, $B \in \mathbb{R}^{r \times d}$, and the rank $r \ll \min(h, d)$. If the forward pass of a base model is defined by $h = xW$, then after applying LoRA, the forward pass becomes

$$h = xW' = x(W + AB) \tag{1}$$
$$= xW + xAB. \tag{2}$$

Typically, this adjustment is only applied to the query, key, value, and output projection matrices in the self-attention module, excluding the feed-forward module.

Because LoRA greatly reduces the training and weight storage costs, it has been widely adopted by the community, and people have created hundreds of thousands of LoRA adapters for pre-trained large language models and diffusion models (Mangrulkar et al., 2022).

2.1 Serving Large Language Models

Most large language models (LLMs) are based on the transformer architecture (Vaswani et al., 2017). The number of parameters in an LLM ranges from several billion to several trillion (Brown et al., 2020; Chowdhery et al., 2022; Fedus et al., 2022), corresponding to disk sizes spanning several gigabytes to even terabytes. This scale results in LLM serving having significant computational and memory demands.

Additionally, the inference process for LLMs requires iterative autoregressive decoding. Initially, the model carries out a forward pass to encode the prompt. Following this, it decodes the output one token at a time. The sequential process makes decoding slow. Since each token attends to the

hidden states of all its preceding tokens, it becomes essential to store the hidden states of all previous tokens. This storage is referred to as the "KV cache". Such a mechanism adds to the memory overhead and causes the decoding process to be more memory-intensive than computation-intensive.

The challenges become even more pronounced in online settings, where requests of varying sequence lengths arrive dynamically. To accommodate such dynamic incoming requests, Orca (Yu et al., 2022) introduces a method of fine-grained, iteration-level scheduling. Instead of scheduling at the request level, Orca batches at the token level. This approach allows for the continuous addition of new requests to the currently running batch, resulting in substantially higher throughput. vLLM (Kwon et al., 2023) further optimizes Orca's memory efficiency using PagedAttention. PagedAttention adopts concepts from virtual memory and paging in operating systems and manages the storage and access of dynamic KV cache tensors in a paged fashion. This method efficiently reduces fragmentation, facilitating larger batch sizes and higher throughput.

When serving very large models that exceed the memory capacity of a single GPU, or when there are stringent latency requirements, it is necessary to parallelize the model across multiple GPUs. Several model parallelism methods have been proposed, such as tensor parallelism (Shoeybi et al., 2019), sequence parallelism (Korthikanti et al., 2023), pipeline parallelism (Huang et al., 2019), and their combinations (Narayanan et al., 2021; Zheng et al., 2022).

3 Overview of S-LoRA

S-LoRA encompasses three principal components of innovation. In Section 4, we introduce our batching strategy, which decomposes the computation between the base model and the LoRA adapters. Additionally, we discuss adapter clustering and admission control when scheduling the requests. The ability to batch across concurrent adapters, introduces new challenges around memory management. In Section 5, we generalize PagedAttention (Kwon et al., 2023) to Unfied Paging, which supports dynamically loading LoRA adapters. This approach uses a unified memory pool to store the KV caches and adapter weights in a paged fashion, which can reduce fragmentation and balance the dynamic changing size of the KV caches and adapter weights. In Section 6, we introduce our new tensor parallelism strategy that enables us to efficiently decouple the base model and LoRA adapters.

4 Batching and Scheduling

4.1 Batching

Our batching strategy aims to support online and high-throughput serving of many LoRA adapters simultaneously.

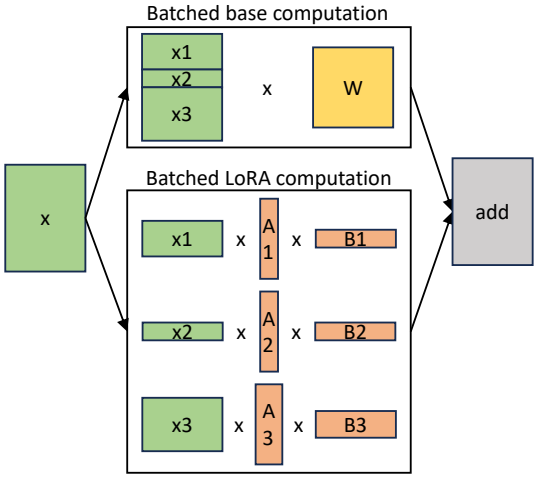

Figure 1. Separated batched computation for the base model and LoRA computation. The batched computation of the base model is implemented by GEMM. The batched computation for LoRA adapters is implemented by custom CUDA kernels which support batching various sequence lengths and adapter ranks.

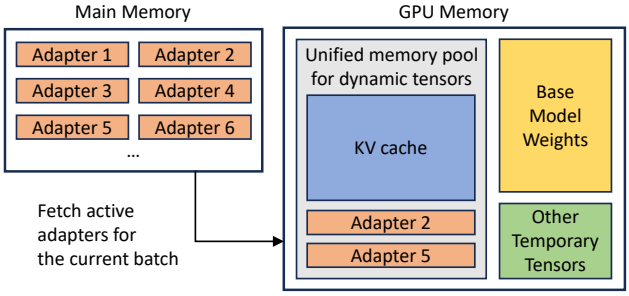

Figure 2. Overview of memory allocation in S-LoRA. S-LoRA stores all adapters in the main memory and fetches the active adapters for the current batch to the GPU memory. The GPU memory is used to store the KV cache, adapter weights, base model weights, and other temporary tensors.

For a single adapter, the method recommended by (Hu et al., 2021) is to merge the adapter weights into the base model weights, resulting in a new model (see Eq. 1). This has the advantage that there is no additional adapter overhead during inference, since the new model has the same number of parameters as the base model. In fact, this was a prominent feature of the original LoRA work.

However, when there are multiple adapters, merging the weights into the base model leads to multiple weight copies and missed batching opportunities. Directly merging the models requires maintaining many copies of the full language model. In the original LoRA paper, the authors proposed adding and subtracting LoRA weights on the fly to enable serving multiple models without increasing the memory overhead. However, this approach doesn't support con-

current inference on separate LoRA adapters and therefore limits batching opportunities.

In this paper, we show that merging LoRA adapters into the base model is inefficient for the multi-LoRA high-throughput serving setting. Instead, we propose computing the LoRA computation xAB on-the-fly as shown in Eq. 2. This avoids weight duplication and enables batching of the more costly xW operation. But this approach also increases the computation overhead. However, because the cost of xAB is substantially lower than xW and there is a considerable savings from batching xW across different adapters, we show that the savings far exceed the additional overhead.

Unfortunately, directly implementing the factored computation of the base model and individual LoRA adapters using the batch GEMM kernel from the existing BLAS libraries would require significant padding and result in poor hardware utilization. This is because of the *heterogeneity* of sequence lengths and adapter ranks.

In S-LoRA, we batch the computation of the base model and then employ custom CUDA kernels to execute the additional xAB for all adapters separately. This process is illustrated by Figure 1. Instead of naively using padding and using the batch GEMM kernel from the BLAS library for the LoRA computation, we implement custom CUDA kernels for more efficient computation without padding. In Subsection 5.3, we discuss the implementation details.

While the number of LoRA adapters can be large if we store them in main memory, the number of LoRA adapters needed for the currently running batch is manageable, because the batch size is bounded by the GPU memory. To take advantage of this, we store all LoRA adapters in the main memory and fetch only the LoRA adapters needed for the currently running batch to the GPU RAM when running the inference for that batch. In this case, the maximum number of adapters that can be served is bounded by the main memory size. This process is illustrated by Figure 2. To achieve high-throughput serving, we adopt the iteration-level scheduling batching strategy from Orca (Yu et al., 2022). In this approach, requests are scheduled at the token level. We immediately incorporate a new request into the running batch if space is available. The request will exit the batch once it reaches the maximum number of generated tokens or fulfills other stopping criteria. This process reduces GPU memory usage but introduces new memory management challenges. In Section 5, we will discuss our techniques to manage memory efficiently.

4.2 Adapter Clustering

To enhance batching efficiency, one potential strategy is reducing the number of active adapters in a running batch. By using fewer adapters, there is an opportunity to allocate more memory to the KV cache, which in turn can facilitate larger batch sizes. Given the common memory capacities of GPUs, they are often underutilized while decoding. Consequently, increasing the batch size can lead to higher throughput. A direct approach to reducing the number of adapters in a running batch is to prioritize batching requests that use the same adapter, a strategy we term "adapter clustering". However, adapter clustering comes with its own set of trade-offs. For example, it can hurt the average latency or fairness among adapters. We provide an ablation study in Appendix A to illustrate how throughput and latency change according to the cluster size.

4.3 Admission Control

In S-LoRA, we also applied an admission control strategy to sustain good attainment when the traffic is higher than the serving system capacity. A serving system is typically characterized by a service level objective (SLO) which specifies the desired latency of processing requests. If the serving system has fixed capacity, it must implement an admission control mechanism, that drops a request, if the system cannot meet its SLO. Otherwise, if no request is dropped, and the number of incoming requests is larger than the system capacity for long enough, the serving system is bound to violate the SLO. We implemented an abort strategy to mimic admission control in S-LoRA, called early abort strategy. Intuitively, we estimate the set of latest requests that we can serve in SLO, and then serve them in the order of arrival time. More implementation details and mathematical justifications are deferred to Appendix B.

5 Memory Management

Compared to serving a single base model, serving multiple LoRA adapters simultaneously presents new memory management challenges. To support many adapters, S-LoRA stores them in the main memory and dynamically loads the adapter weights needed for the currently running batch into GPU RAM. During this process, there are two noticeable challenges. The first is memory fragmentation, resulting from the dynamic loading and offloading adapter weights of various sizes. The second is the latency overhead introduced by adapter loading and offloading. To tackle these challenges efficiently, we propose Unfied Paging and overlap the I/O with computation by prefetching adapter weights.

5.1 Unified Paging

Understanding the nature of adapter weights is essential for optimizing memory usage. Our primary observation is that these dynamic adapter weights are analogous to dynamic KV caches in several ways:

- **Variable sizes and operations:** Just as the size of

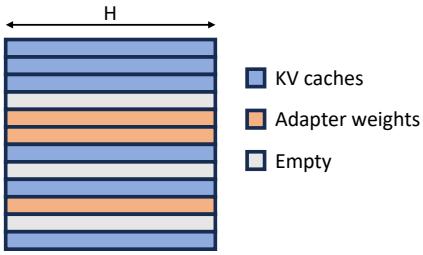

Figure 3. Unified memory pool. We use a unified memory pool to store both KV caches and adapter weights in a non-contiguous way to reduce memory fragmentation. The page size is H elements.

KV cache size fluctuates with the sequence length, the ranks of the active adapters can also depend on the choice of adapter associated with each request. KV caches are allocated when requests arrive and deallocated once the requests are completed. Similarly, adapter weights are loaded and cleared with each request. If not managed properly, this variability can result in fragmentation.

- **Dimensionality:** A KV cache tensor for a request in a layer has a shape of (S, H), where S denotes the sequence length and H represents the hidden dimension. Meanwhile, the shape of a LoRA weight is (R, H), with R standing for the rank and H the hidden dimension. Both share a dimension size of H that can be leveraged to reduce fragmentation.

Motivated by these parallels, we extend the idea of PagedAttention (Kwon et al., 2023) to Unified Paging which manages adapter weights in addition to the KV cache. Unified Paging uses a unified memory pool to jointly manage both KV cache and adapter weights. To implement this, we first allocate a large buffer statically for the memory pool. This buffer uses all available space except for the space occupied by the base model weights and temporary activation tensors. Both KV caches and adapter weights are stored in this memory pool in a paged manner, with each page corresponding to a vector of H. Thus, a KV cache tensor with a sequence length of S uses up S pages, while a LoRA weight tensor of rank R takes up R pages. Figure 3 illustrates the layout of our memory pool, where KV caches and adapter weights are stored interleaved and non-contiguously. This approach significantly reduces fragmentation, ensuring that adapters weights of various ranks can coexist with dynamic KV caches in a structured and systematic manner.

5.2 Prefetching and Overlapping

Although the unified memory pool mitigates fragmentation, the I/O overhead from loading and offloading remains a concern—especially when dealing with numerous or large adapters. The latency introduced by waiting to load these adapters can compromise the efficiency of the system.

To proactively address this issue, we introduce a dynamic prediction mechanism. While running the current decoding batch, we predict the adapters required for the next batch based on the current waiting queue. This prediction allows us to prefetch and store them in available memory. Such a forward-looking strategy keeps most of the adapters needed for the next batch already in place before running it, which reduces I/O time for adapter swapping.

5.3 Custom Kernels for heterogeneous LoRA batching on Non-Contiguous Memory

Due to the design of the unified memory pool, the adapter weights are stored in non-contiguous memory. To run computations efficiently under this design, we implement custom CUDA kernels that support batching LoRA computations with *varying ranks* and sequence lengths in a *non-contiguous* memory layout.

In the prefill stage, the kernel handles a sequence of tokens and gathers adapter weights with different ranks from the memory pool. We call this kernel Multi-size Batched Gather Matrix-Matrix Multiplication (MBGMM). It is implemented in Triton (Tillet et al., 2019) with tiling.

In the decode stage, the kernel handles a single token and gathers adapter weights with different ranks from the memory pool. We call this kernel Multi-size Batched Gather Matrix-Vector Multiplication (MBGMV). We implemented two versions of this kernel: one in Triton and another by modifying an earlier version of Punica kernels (Chen, 2023) to extend support for non-contiguous memory, multiple ranks in a batch, and more fine-grained memory gathering. We found the latter one was faster, so we used it in the experiments.

Punica (Chen et al., 2023) is concurrent work on serving multiple LoRA adapters, which will be discussed in Section 8. In addition to Triton and Pucina kernels, NVIDIA CUTLASS also provides high-performance kernels for grouped GEMM (NVIDIA) that can be used for heterogeneous batching.

6 Tensor Parallelism

We design novel tensor parallelism strategies for batched LoRA inference to support multi-GPU inference of large transformer models. Tensor parallelism is the most widely used parallelism method because its single-program multiple-data pattern simplifies its implementation and integration with existing systems. Tensor parallelism can reduce the per-GPU memory usage and latency when serving large models. In our setting, the additional LoRA adapters introduce new weight matrices and matrix multiplications, which calls for new partition strategies for these added items.

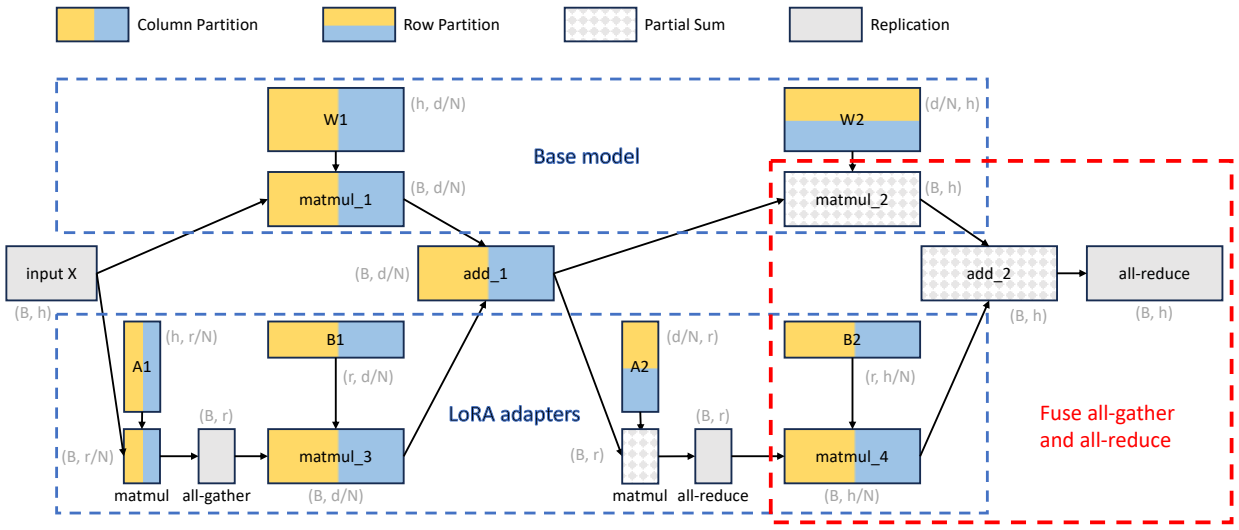

Figure 4. Tensor parallelism partition strategy for batched LoRA computation. This is a computational graph where nodes represent tensors/operators and the edges represent dependency. We use different colors to represent different partition strategies, which include column partition, row partition, partial sum, and replication. The per-GPU shape of each tensor is also annotated in gray. Note that B is the number of tokens, h is the input dimension, N is the number of devices, d is the hidden size, and r is the adapter rank.

6.1 Partition Strategy

Since the base model uses the Megatron-LM tensor parallelism strategy (Shoeybi et al., 2019), our approach aims to align the partition strategies of inputs and outputs of the added LoRA computation with those of the base model. In this way, we can minimize the communication costs by avoiding unnecessary communications and fusing some communications.

We use the feed-forward module (2-layer MLP) to illustrate our partition strategy. We will explain later how this strategy can easily be adapted to the self-attention layer. As depicted in Figure 4, the upper box illustrates the base model's Megatron-LM partition strategy: the first weight matrix ($W1$) is column-partitioned, and the second ($W2$) is row-partitioned. An all-reduce communication is required to accumulate the partial sum from distributed devices.

The lower box illustrates the partitioning strategy for the added LoRA computation. The matrices $A1$ and $B1$ for the adapter of the first weight matrix ($W1$) are column-partitioned. An all-gather operation is used to collect the intermediate results. The matrices $A2$ and $B2$ for the adapter of the second weight ($W2$) are row-partitioned and column-partitioned, respectively. An all-reduce operation is used to sum up the intermediate results. Finally, the result from the LoRA computation is added to that from the base model (`add_2`). A single all-reduce operation is sufficient to accumulate the final results. It is worth noting that we are essentially fusing an all-gather operation for `matmul_4` with the final all-reduce. To our knowledge, this parallelization strategy has not been studied before.

Next, we discuss adapting the strategy from the 2-layer MLP to the self-attention layer. Similar to the Megatron-LM strategy, we partition the head dimension of the self-attention layer. The query-key-value projection weight matrix can be seen as $W1$ in our example and the output projection weight matrix can be seen as $W2$ in our example.

6.2 Communication and Memory Cost Analysis

Let N be the number of devices, B be the number of tokens, h be the hidden size, and r be the adapter rank. The communication cost of the base model is one all-reduce, or $\frac{2(N-1)Bh}{N}$. The communication cost of the added LoRA computation is three all-gather for query, key, and value projections, and one all-reduce for the output projection. Formally, it is $3\frac{(N-1)Br}{N} + \frac{2(N-1)Br}{N} = \frac{5(N-1)Br}{N}$.

Under our strategy, the additional communication cost introduced by LoRA is negligible when compared to the communication cost of the base model, because $r \ll h$. Intuitively, this is achieved by carefully scheduling communications on the small intermediate tensors of LoRA computation and fusing communications with base models.

In terms of memory usage, our strategy is optimal because we partition all weight matrices among all devices and there is no replicated weight matrix.

7 Evaluation

We evaluate the performance of S-LoRA on both synthetic and real production workloads. S-LoRA is built on top of LightLLM (ModelTC, 2023), a single-model LLM serv-

ing system based on PyTorch (Paszke et al., 2019) and Triton (Tillet et al., 2019). We evaluate the scalability of S-LoRA by serving up to two thousand LoRA adapters simultaneously and compare it with other strong baselines. We then perform ablation studies to verify the effectiveness of individual components.

7.1 Setup

Model. We test the Llama model series (Touvron et al., 2023a;b), one of the most popular open large language models. We consider 5 different model and adapter configurations, which are listed in Table 1 [1]. Our optimizations can be easily adapted to other transformer-based architectures as well, such as GPT-3 (Brown et al., 2020) and PaLM (Chowdhery et al., 2022; Anil et al., 2023).

Setting	Base model	Hidden size	Adapter ranks
S1	Llama-7B	4096	{8}
S2	Llama-7B	4096	{64, 32, 16, 8}
S4	Llama-13B	5120	{64, 32, 16}
S5	Llama-30B	7168	{32}
S6	Llama-70B	8192	{64}

Table 1. Model and adapter configurations.

Hardware. We conduct tests on various hardware settings, including a single NVIDIA A10G GPU (24GB), a single A100 GPU (40GB), a single A100 GPU (80GB), and multiple A100 GPUs (40GB/80GB). The host's main memory varies based on the GPU setup, ranging from 64 GB to 670 GB. We will show that S-LoRA can efficiently scale the number of adapters, limited only by the available main memory.

Baselines. We benchmark several variants of S-LoRA, HuggingFace PEFT (Mangrulkar et al., 2022), and vLLM (Kwon et al., 2023).

- "HuggingFace PEFT" is a library for training and running parameter-efficient fine-tuning models. It lacks advanced batching and memory management. We build a server using it that batches single adapter requests and switches adapter weights between batches.
- "vLLM m-packed" is a simple multi-model serving solution based on vLLM, a high-throughput serving system. Because vLLM does not support LoRA, we merge the LoRA weights into the base model and serve the multiple versions of the merged weights separately. To serve m LoRA adapters, we run m vLLM workers on a single GPU, where multiple workers are separate processes managed by NVIDIA MPS. We statistically allocate the GPU memory proportionally to the average request rate for each process.
- "S-LoRA" is S-LoRA with all the optimizations and it is using the first-come-first-serve scheduling strategy.
- "S-LoRA-no-unify-mem" is S-LoRA without the unified memory management.
- "S-LoRA-bmm" is S-LoRA without unified memory management and customized kernels. It copies the adapter weights to continuous memory space and performs batched matrix multiplication with padding.

[1]For Llama-70B, we used different architecture parameters than the official model and did not employ group-query attention.

Metrics. There are several metrics to measure the performance of serving systems, including latency and throughput. Following common practice, we report the *throughput, average request latency, average first token latency, and SLO attainment*. SLO attainment is defined as the percentage of requests that return the first token in 6 seconds. Additionally, we introduce a new metric termed *user satisfaction* (see Appendix B), which offers a more fine-grained analysis of the first token latency. Intuitively, a shorter first token latency gives a higher satisfaction. The satisfaction becomes 0 if the first token latency exceeds the SLO.

7.2 End-to-End Results on Synthetic Workloads

Workload trace. We generate synthetic workload traces using the Gamma process, which is commonly used in machine learning serving literature (Crankshaw et al., 2020; Li et al., 2023). Given n adapters, the requests for adapter i are modeled using a Gamma arrival process with a mean rate of λ_i and a coefficient of variance (CV) of cv. The mean rate, λ_i, adheres to a power-law distribution with an exponent α. The total request rate for all adapters is R requests per second. For the n adapters, we set their ranks based on the list provided in Table 1 with a round-robin method. Our tests cover various combinations of n, α, R, and cv. For every request, the input and output lengths are sampled from uniform distributions $U[I_l, I_u]$ and $U[O_l, O_u]$ respectively. The default duration of a trace is 5 minutes. To conduct comprehensive experiments, we first pick a set of default parameters for generating workloads, as shown in Table 2. We then vary one of the n, α, R, *and* cv to see how each factor affects the performance.

Table 2. Default parameters for generating the synthetic workloads. "7B @ A10G" means running a Llama-7B on a single A10G.

Setting	n	α	R	cv	$[I_l, I_u]$	$[O_l, O_u]$
7B @ A10G (24G)	200	1	2	1	[8, 512]	[8, 512]
7B @ A100 (80G)	200	1	10	1	[8, 512]	[8, 512]
13B @ A100 (40G)	200	1	2	1	[8, 512]	[8, 512]
13B @ A100 (80G)	400	1	6	1	[8, 512]	[8, 512]

Comparison with other systems. We compare S-LoRA with both vLLM-packed and HuggingFace PEFT for serving many LoRA adapters. The results are shown in Table 3. Remarkably, S-LoRA can serve 2,000 adapters simultane-

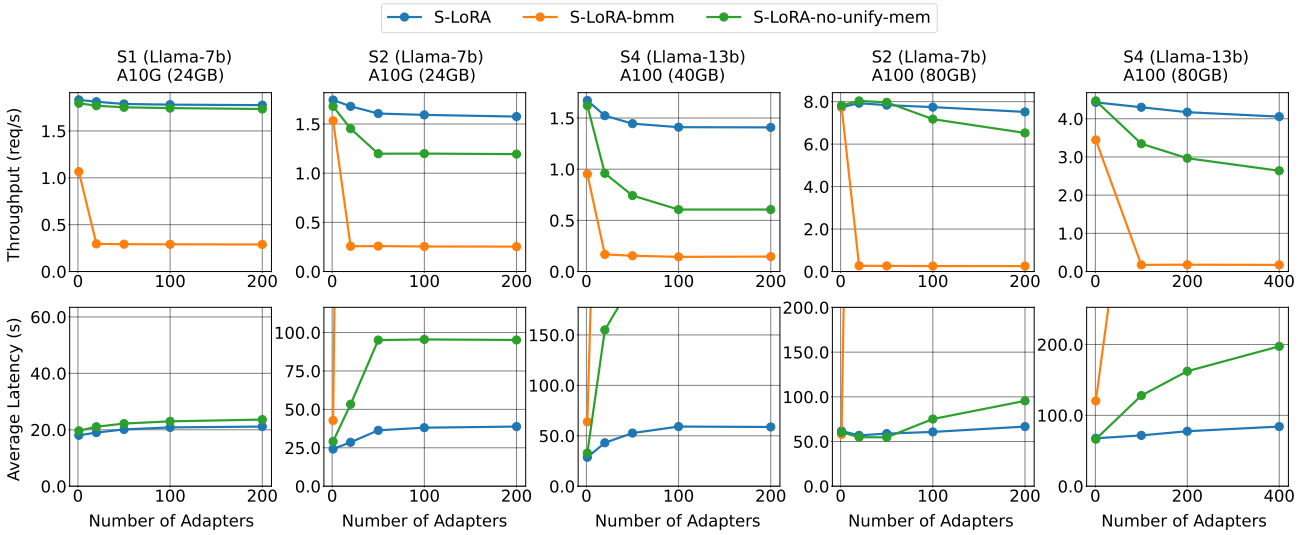

Figure 5. The throughput and average request latency of S-LoRA and its variants under different numbers of adapters. S-LoRA achieves significantly better performance and can scale to a large number of adapters. We run S-LoRA-bmm for a shorter duration since it has a significantly lower throughput. Some S-LoRA-bmm curves are omitted because it is out of the figures's scope.

Table 3. Throughput (req/s) comparison between S-LoRA, vLLM-packed, and PEFT. The hardware is a single A100 (80GB). We run PEFT for a shorter duration when $n = 100$. We do not evaluate PEFT for $n \geq 1000$, as its throughput is already very low for a small n. "OOM" denotes out-of-memory.

Model Setup	n	S-LoRA	vLLM-packed	PEFT
S1	5	8.05	2.04	0.88
	100	7.99	OOM	0.25
	1000	7.64	OOM	-
	2000	7.61	OOM	-
S2	5	7.48	2.04	0.74
	100	7.29	OOM	0.24
	1000	6.69	OOM	-
	2000	6.71	OOM	-
S4	2	4.49	3.83	0.54
	100	4.28	OOM	0.13
	1000	3.96	OOM	-

ously, maintaining minimal overhead for the added LoRA computation. In contrast, vLLM-packed needs to maintain multiple weight copies and can only serve fewer than 5 adapters due to the GPU memory constraint. The throughput of vLLM-packed is also much lower due to the missed batching opportunity. Although PEFT can swap adapters between batches, enabling it to handle a large number of adapters, its lack of advanced batching methods and memory management results in significantly worse performance. Overall, S-LoRA achieves a throughput up to 4x higher than vLLM-packed when serving a small number of adapters, and up to 30x higher than PEFT, while supporting a significantly larger number of adapters.

Comparing with own variants. Since no baseline system can efficiently scale to a large number of adapters, we now focus on comparing S-LoRA with its own variants. Figure 5 illustrates how they scale with the number of adapters. S-LoRA achieves noticeably higher throughput and lower latency compared to S-LoRA-bmm and S-LoRA-no-unify-mem. This implies that our memory pool and custom kernels are effective. When the number of adapters increases, the throughput of S-LoRA initially experiences a slight decline due to the overhead introduced by LoRA. However, once the number of adapters reaches a certain threshold (e.g., 100 in most experiments), the throughput of S-LoRA no longer decreases. This stability can be attributed to the fact that as the number of adapters grows, the number of activated adapters for the currently running batch remains unchanged, maintaining a constant overhead. Consequently, S-LoRA can scale to a much larger number of adapters without incurring additional overhead, constrained only by the available main memory.

Figure 6 demonstrates the variation in throughput, first token latency, and SLO attainment relative to the total request rate, revealing a pattern consistent with the aforementioned observations and underscoring the efficacy of our design.

7.3 End-to-End Results on Real Workloads

Real workload trace. We construct real-world serving traces by downsampling from the traces of LMSYS Chatbot Arena (Zheng et al., 2023b;a), a website that serves multiple LLMs. The raw log from Arena does not concern LoRA adapters; it focuses on different base models. Nonetheless, we treat the distribution of different base models as if they

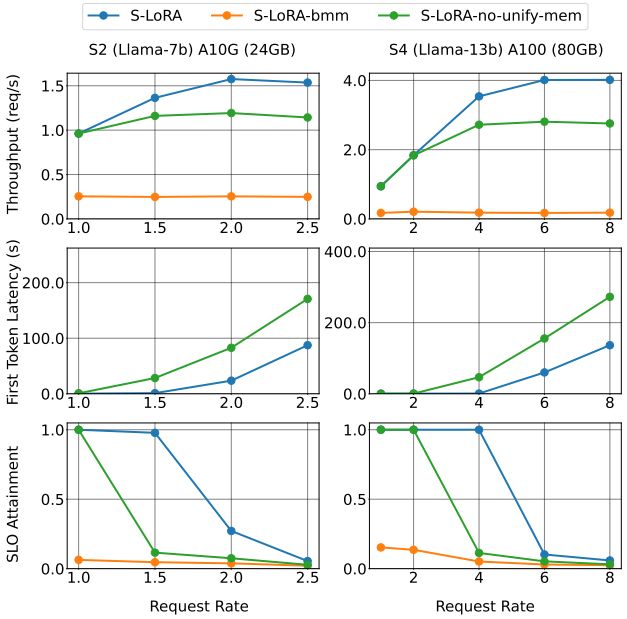

Figure 6. The throughput, first token latency, and SLO attainment of S-LoRA and its variants under different request rates. Note that in both settings the first token latency of S-LoRA-bmm is out of the figure's scope.

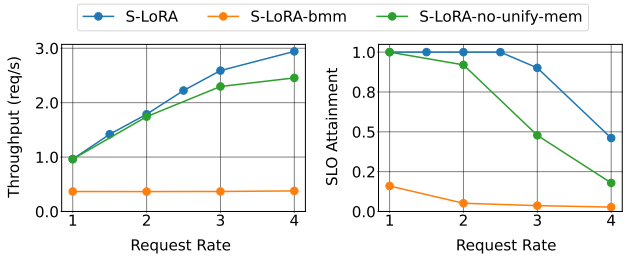

Figure 7. The throughput of S-LoRA and its variants on real workload traces with different request rates. The model and hardware configuration is S2 on an A10G (24GB).

were the distribution of different adapters of a single base model. The raw log can be sampled into traces that exhibit varying request rates, denoted as R, and durations, represented by D. To achieve this, we sample $R \cdot D$ requests from the raw log and rescale the time stamps to fit within the range of $[0, D]$. The number of models n corresponds to the number of adapters. Furthermore, we set the adapter ranks based on Table 1 with a round-robin method.

Since we are using a real workload trace, there are no parameters such as α, λ_i, or cv. For consistency, we set the duration to 5 minutes. We adjust the request rate R to study its impact on performance metrics. In the sampled trace, the average input length is 85 tokens, the average output length is 165 tokens, and the number of adapters is around 26.

Results. Figure 7 shows the throughput and attainment results, which show a similar pattern to the synthetic workloads. This means the strong performance S-LoRA holds for real world workloads.

7.4 Multi-GPU Tensor Parallelism

We test the scalability of our tensor parallelism strategy by running 1) Llama-30B on two A100 (40GB) and four A100 (40GB) GPUs with 10 to 100 adapters; and 2) Llama-70B on two A100 (80GB) and four A100 (80GB) GPUs with 10 adapters. We then report the serving throughputs.

As depicted in Figure 8, the disparity between S-LoRA with and without LoRA communication is small. This suggests that the added LoRA communication in our strategy has a very small overhead. The figure further reveals that the communication overhead due to LoRA is less than the computational overhead it introduces. Furthermore, when transitioning from 2 GPUs to 4 GPUs, the serving throughput increases by more than 2 times. This significant increase can be attributed to the fact that the system is predominantly memory-bound in this context. Adding more GPUs alleviates memory constraints, leading to superlinear scaling. In conclusion, the results verify both the minimal overhead and the scalability of our tensor parallelism strategy.

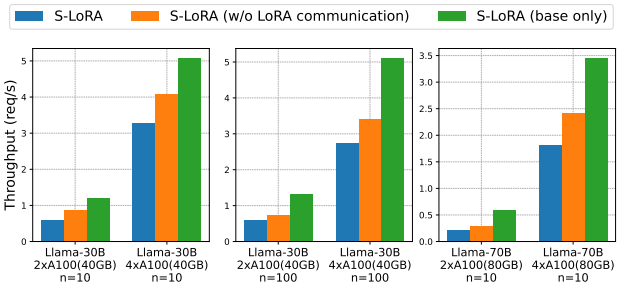

Figure 8. Throughput with tensor parallelism.

7.5 Ablation Study

Merging adapter weights versus computing on-the-fly. While S-LoRA does not merge adapter weights and computes LoRA matrices on-the-fly each time, we compare it with an alternative design that merges an adapter with the base model, denoted as $x(W + AB)$, as proposed in the LoRA paper. This approach involves: 1) Updating the base model with the current adapter weights before each new batch; and 2) Switching to a new adapter if there are too many waiting requests.[2] This method is efficient for a small number of adapters due to the reduced LoRA computation overhead.

Results in Figure 9 demonstrate that with just one adapter,

[2]This is different from PEFT. For example, it has continuous batching and PagedAttention, which are not enabled in PEFT.

the merging approach outperforms the on-the-fly computation owing to a one-time merging cost. However, its performance declines with more than 2 adapters, primarily because of the time-consuming switch between adapters. Such switching results in periods of GPU under-utilization. Furthermore, a smaller value of α causes requests to be distributed unevenly across adapters, which in turn reduces batch sizes and overall performance.

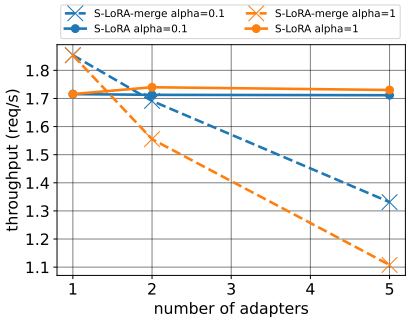

Figure 9. Ablation study comparing adapter merging and on-the-fly compute for S2 on A10G (24GB) with different α and number of adapters. The settings for the synthetic workloads are $R = 2$, $cv = 1$, $[I_t, I_u] = [8, 512]$, $[O_l, O_u] = [8, 512]$.

Early abort strategy experiments. We compared S-LoRA's early abort strategy to First Come First Serve (FCFS) and Last Come First Serve (LCFS) for optimizing user satisfaction and SLO attainment. As shown in Figure 10, S-LoRA-Abort outperforms both, especially as cv scales. FCFS is least effective, often processing requests that have already missed the SLO. LCFS, similar to a greedy algorithm that only prioritizes the newest requests, works well for small cv, but its performance drops with larger cv. S-LoRA-Abort excels as it avoids prioritizing only the newest requests, as detailed in Appendix B.

8 Related Work

Optimize LLM serving with system techniques. The significance of the transformer architecture has led to the development of many specialized serving systems for it. These systems use advanced batching mechanisms (Fang et al., 2021; Yu et al., 2022), memory optimizations (Sheng et al., 2023; Kwon et al., 2023), GPU kernel optimizations (Wang et al., 2021; Aminabadi et al., 2022; NVIDIA, 2023; Dao, 2023), model parallelism (Pope et al., 2022; Aminabadi et al., 2022), parameter sharing (Zhou et al., 2022), and speculative execution (Stern et al., 2018; Miao et al., 2023) for efficient serving. Among them, PetS (Zhou et al., 2022) is most relevant to ours. However, PetS only considers the serving for small encoder-only BERT models. It does not consider generative inference, a very large number of adapters or large models go beyond a single GPU, so it does not address the problems in our settings.

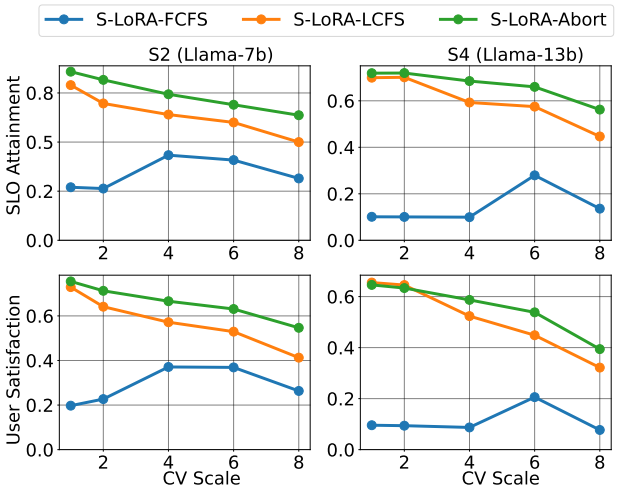

Figure 10. Ablation study for early abort scheduling strategy on A10G-24G (S1) and A100-80G (S4). Other settings follow the description in Table 2.

In concurrent work, Punica (Chen et al., 2023) explored the concept of decomposed computation for the base model and adapters. Some of our CUDA kernels were developed based on the implementation presented in a previous blog post of Punica, with additional support for batching different ranks and non-contiguous memory. Analyzing kernel performance is not the focus of this paper, but it is discussed in Punica. Our work differs from Punica in our novel memory management and tensor parallelism techniques, which have not been covered in any previous work.

Optimize LLM serving with algorithm techniques. In addition to system-level improvements, inference efficiency can be enhanced using algorithm techniques like quantization (Yao et al., 2022; Dettmers et al., 2022; Frantar et al., 2022; Xiao et al., 2023; Lin et al., 2023), sparsification (Frantar & Alistarh, 2023; Zhang et al., 2023b) and model architecture improvements (Shazeer, 2019). These approaches can reduce memory consumption and accelerate the computation, with a minor compromise in model quality. They are complementary to the techniques in this paper.

Parameter-efficient fine-tuning. Recent work has developed methods for parameter-efficient fine-tuning of large pre-trained language models. These methods show fine-tuning is possible with only a small fraction of tuned parameters. The state-of-the-art methods include LoRA (Hu et al., 2021), Prefix-tuning (Li & Liang, 2021), P-Tuning (Liu et al., 2021), Prompt tuning (Liu et al., 2023; Lester et al., 2021), AdaLoRA (Zhang et al., 2022), and $(IA)^3$ (Liu et al., 2022). While our paper focuses on LoRA due to its wide adoption, most techniques can be easily applied to other parameter-efficient fine-tuning methods as well.

General purpose model serving systems. Over the years,

the domain of general model serving has seen significant advancements, Notable systems from earlier research include Clipper (Crankshaw et al., 2017), TensorFlow Serving (Olston et al., 2017), Nexus (Shen et al., 2019), InferLine (Crankshaw et al., 2020), and Clockwork (Gujarati et al., 2020). These systems delve into topics such as batching, caching, and model placement, catering to both individual and multiple model deployments. In more recent developments, DVABatch (Cui et al., 2022), REEF (Han et al., 2022), Shepherd (Zhang et al., 2023a) and AlpaServe (Li et al., 2023) have explored the ideas of multi-entry multi-exit batching, preemption, and statistical multiplexing with model parallelism. Although these systems have made significant contributions, they overlook the auto-regressive characteristics and parameter-efficient adapters in LLM serving, leading to potential optimization gaps.

9 Conclusion

We present S-LoRA, a system capable of serving thousands of LoRA adapters from a single machine with much higher throughput compared to existing systems. S-LoRA is made possible by our innovative design of the unified memory pool, tensor parallelism strategy, adapter batching, and CUDA kernels. S-LoRA enables large-scale, customized fine-tuning services essential for deploying models tailored to diverse requirements. Future extensions of S-LoRA will encompass support for additional adapter methods, enhanced fused kernels, and the use of multiple CUDA streams to parallelize base model and LoRA computations.

Acknowledgment

This research was supported by gifts from Anyscale, Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, and VMware. Ying is partly supported by the Stanford Center for Automated Reasoning. We thank Clark Barrett for academic advising and funding support. We also thank Yonghao Zhuang and Lisa Dunlap for their helpful discussions and feedback.

References

Alayrac, J.-B., Donahue, J., Luc, P., Miech, A., Barr, I., Hasson, Y., Lenc, K., Mensch, A., Millican, K., Reynolds, M., et al. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736, 2022.

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 Wolf, F., Shende, S., Culhane, C., Alam, S. R., and Jagode, H. (eds.), *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022.

Anil, R., Dai, A. M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., Shakeri, S., Taropa, E., Bailey, P., Chen, Z., et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Chen, L. Potentials of multitenancy fine-tuned llm serving. https://le.qun.ch/en/blog/2023/09/11/multi-lora-potentials/, 2023.

Chen, L., Ye, Z., Wu, Y., Zhuo, D., Ceze, L., and Krishnamurthy, A. Punica: Multi-tenant lora serving, 2023.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

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, 2017.

Crankshaw, D., Sela, G.-E., Mo, X., Zumar, C., Stoica, I., Gonzalez, J., and Tumanov, A. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 477–491, 2020.

Cui, W., Zhao, H., Chen, Q., Wei, H., Li, Z., Zeng, D., Li, C., and Guo, M. Dvabatch: Diversity-aware multi-entry multi-exit batching for efficient processing of dnn services on gpus. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 183–198, 2022.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Llm.int8(): 8-bit matrix multiplication for transformers at scale. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022.

Fang, J., Yu, Y., Zhao, C., and Zhou, J. Turbotransformers: an efficient gpu serving system for transformer models.

In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 389–402, 2021.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270, 2022.

Frantar, E. and Alistarh, D. Massive language models can be accurately pruned in one-shot. *arXiv preprint arXiv:2301.00774*, 2023.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

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, 2020.

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, 2022.

Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pp. 2790–2799. PMLR, 2019.

Hu, E. J., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2021.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Jamin, S., Shenker, S., Zhang, L., and Clark, D. D. An admission control algorithm for predictive real-time service. In *Network and Operating System Support for Digital Audio and Video: Third International Workshop La Jolla, California, USA, November 12–13, 1992 Proceedings 3*, pp. 347–356. Springer, 1993.

Kenton, J. D. M.-W. C. and Toutanova, L. K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pp. 4171–4186, 2019.

Korthikanti, V. A., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 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*, pp. 611–626. ACM, 2023.

Lester, B., Al-Rfou, R., and Constant, N. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3045–3059, 2021.

Li, X. L. and Liang, P. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 4582–4597, 2021.

Li, Z., Zheng, L., Zhong, Y., Liu, V., Sheng, Y., Jin, X., Huang, Y., Chen, Z., Zhang, H., Gonzalez, J. E., et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 663–679, 2023.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., and Han, S. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

Liu, H., Tam, D., Muqeeth, M., Mohta, J., Huang, T., Bansal, M., and Raffel, C. A. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35: 1950–1965, 2022.

Liu, X., Ji, K., Fu, Y., Tam, W. L., Du, Z., Yang, Z., and Tang, J. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *arXiv preprint arXiv:2110.07602*, 2021.

Liu, X., Zheng, Y., Du, Z., Ding, M., Qian, Y., Yang, Z., and Tang, J. Gpt understands, too. *AI Open*, 2023. ISSN 2666-6510. doi: https://doi.org/10.1016/j.aiopen.2023.08.012. URL https://www.sciencedirect.com/science/article/pii/S2666651023000141.

Mangrulkar, S., Gugger, S., Debut, L., Belkada, Y., Paul, S., and Bossan, B. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Wang, Z., Wong, R. Y. Y., Chen, Z., Arfeen, D., Abhyankar, R., and Jia, Z. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 2023.

ModelTC. Lightllm: Python-based llm inference and serving framework. https://github.com/ModelTC/lightllm, 2023. GitHub repository.

Naghshineh, M. and Schwartz, M. Distributed call admission control in mobile/wireless networks. *IEEE Journal on Selected Areas in Communications*, 14(4):711–717, 1996.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

NVIDIA. Cutlass gemm grouped. https://github.com/NVIDIA/cutlass/blob/main/examples/24_gemm_grouped/gemm_grouped.cu.

NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2023.

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. *arXiv preprint arXiv:1712.06139*, 2017.

OpenAI. Gpt-4 technical report, 2023.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022.

Shazeer, N. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

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*, pp. 322–337, 2019.

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 *International Conference on Machine Learning, ICML 2023*, volume 202 of *Proceedings of Machine Learning Research*, pp. 31094–31116. PMLR, 2023.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Stern, M., Shazeer, N., and Uszkoreit, J. Blockwise parallel decoding for deep autoregressive models. *Advances in Neural Information Processing Systems*, 31, 2018.

Tillet, P., Kung, H.-T., and Cox, D. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Vin, H., Goyal, P., and Goyal, A. A statistical admission control algorithm for multimedia servers. In *Proceedings of the second ACM international conference on Multimedia*, pp. 33–40, 1994.

Wang, X., Xiong, Y., Wei, Y., Wang, M., and Li, L. Lightseq: A high performance inference library for transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*, pp. 113–120, 2021.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning, ICML 2023*, volume 202 of *Proceedings of Machine Learning Research*, pp. 38087–38099. PMLR, 2023.

Yao, Z., Aminabadi, R. Y., Zhang, M., Wu, X., Li, C., and He, Y. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022.

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, 2022.

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 2023a. USENIX Association. ISBN 978-1-939133-33-5. URL https://www.usenix.org/conference/nsdi23/presentation/zhang-hong.

Zhang, Q., Chen, M., Bukharin, A., He, P., Cheng, Y., Chen, W., and Zhao, T. Adaptive budget allocation for parameter-efficient fine-tuning. In *The Eleventh International Conference on Learning Representations*, 2022.

Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., et al. H _2 o: Heavy-hitter oracle for efficient generative inference of large language models. *arXiv preprint arXiv:2306.14048*, 2023b.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, 2022.

Zheng, L., Chiang, W.-L., Sheng, Y., Li, T., Zhuang, S., Wu, Z., Zhuang, Y., Li, Z., Lin, Z., Xing, E., et al. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *arXiv preprint arXiv:2309.11998*, 2023a.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Conference on Neural Information Processing Systems, Datasets and Benchmarks Track*, 2023b.

Zhou, Z., Wei, X., Zhang, J., and Sun, G. {PetS}: A unified framework for {Parameter-Efficient} transformers serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 489–504, 2022.

A Additional Experiment Results

A.1 Analysis of PEFT

In our evaluation of PEFT, several key observations were discerned. First, the lack of KV cache support makes the maximal batch size of PEFT much smaller compared to S-LoRA. For instance, in A10G S1, S-LoRA can accommodate a maximal batch size of 30, while PEFT can only accommodate a maximal batch size of 6. Secondly, the lack of continuous batching support makes shorter requests wait for longer requests in a batch. These two factors together result in the low throughput of PEFT even when there is only one adapter. When there are more adapters, the lack of batching support across different adapters makes the throughput even lower, resulting in only 0.17 request/second performance in the largest number of adapters we test. As another result, the average latency explodes because the request rate is far beyond the maximal capacity of the PEFT system. In Table 5, we show that even in the lowest request rate we test, PEFT fails to process with a low latency.

num adapters	throughput	avg. latency	avg. attainment
1	0.26	1021.86	0.0
20	0.23	1178.52	0.0
50	0.22	1293.97	0.0
100	0.20	1421.16	0.0
200	0.17	1609.50	0.0

Table 4. PEFT results on the synthetic workload S1 against number of adapters.

req rate	throughput	avg. latency	avg. attainment
1	0.11	1165.46	0.0
1.5	0.13	1398.56	0.0
2	0.17	1614.37	0.0
2.5	0.18	1904.73	0.0

Table 5. PEFT results on the synthetic workload S1 against request rate.

A.2 Experiments for adapter clustering.

We implement a straightforward adapter clustering algorithm. Let parameter d be the number of adapters in a batch. In addition to the FCFS order (or early abort order if turned on), if the number of adapters reaches d, we will prioritize the requests that have their adapter already in the batch. But if the requests from the d adapters cannot fill all the space for a running batch, we allow other requests to be added. We run some additional experiments to study how the number of clusters impacts throughput and SLO attainment. We call d as the number of clusters in the figure. As shown in Figure 11 and Figure 12, the impact is not significant but observable, especially for larger α and cv. Generally, a small d can result in better performance. The small fluctuation for small d's may be because of the scheduler overhead and random noise.

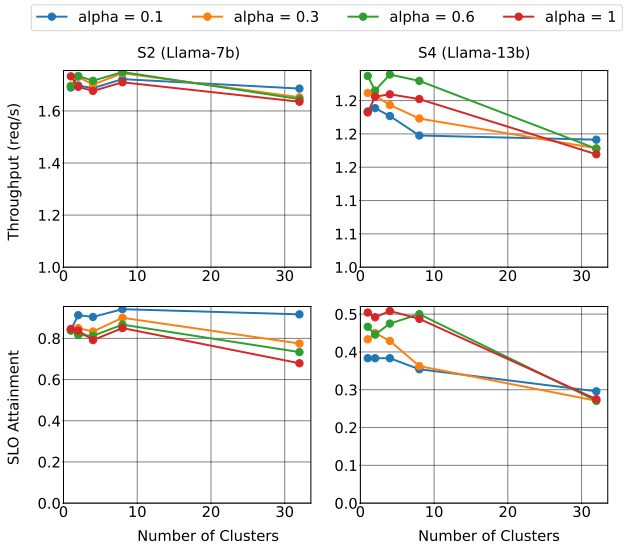

Figure 11. Ablation study for different number of clusters on A100 (40GB) with different α. The settings for the synthetic workload trace are $n = 32, \alpha = [0.1, 0.3, 0.6, 1]$, $R = 2$, $cv = 1$, $[I_t, I_u] = [8, 512]$, $[O_l, O_u] = [8, 512]$

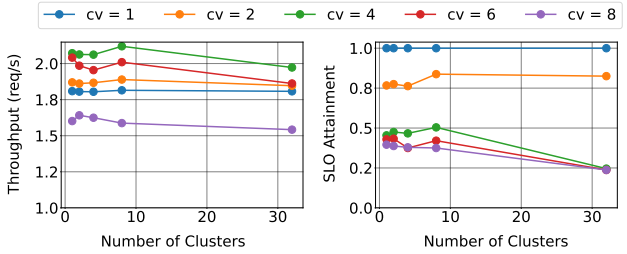

Figure 12. Ablation study for different number of clusters on S2 (Llama-7b) A100 (80GB) with different cv. The settings for the synthetic workload trace are $n = 32, \alpha = 1$, $R = 2$, $cv = [1, 2, 4, 6, 8]$, $[I_t, I_u] = [8, 512]$, $[O_l, O_u] = [8, 512]$

B Admission Control in S-LoRA

Traditional admission control usually assumes a hard threshold for the delay in the service (Jamin et al., 1993; Vin et al., 1994; Naghshineh & Schwartz, 1996), and controls the total number of violations of delay. Here for LoRA serving, we assume a soft threshold characterized by the user's reward function. For illustration purposes, let the arrival time of the requests be integers, and assume that we process one query in each time period of length 1. Let $Q = \{q_1, q_2, \cdots q_n\}$ be the request queue in the ascending order of the arrival time, and l be the desired number of served requests. We quantify the user's satisfaction with a

reward function $r : \mathbb{R}^+ \mapsto [0, 1]$ that maps the first token latency of a request to a scalar in between $[0, 1]$, where 0 represents the user losing patience and giving up the query, and 1 represents the user is completely satisfied with the latency. Let t_i be the latency of serving the request q_i in the queue Q. Then we aim to solve the following constrained optimization:

$$\max \sum_{i=1}^{n} r(t_i) \quad \text{s.t. } \mathbb{1}(r(t_i) > 0) = l. \tag{3}$$

We show that when the derivative of reward is non-increasing, the optimal solution to the above constrained optimization problem is to serve the most recent l elements $q_{n-l+1}, q_{n-l+2}, \cdots, q_n$ in order.

Theorem B.1. *Assume that $r'(t) \leq 0$ for any $t \in \mathbb{R}^+$. The optimal solution to Equation (3) is to serve the most recent l elements $q_{n-l+1}, q_{n-l+2}, \cdots, q_n$ in order.*

The proof is deferred to Appendix B.1. In practice, for a given request queue, we can estimate the largest possible number of requests to be served in SLO as l. Then we take the most recent l elements for serving. Such an l can be approximated by simulating a First-Come-First-Serve (FCFS) strategy, which is optimized to serve requests as many as possible.

In S-LoRA, the scenario is more complicated because of the heterogeneity and unpredictability of the sequence length. As an approximation, we implement a heuristic as follows. The high-level scheduling is that we will fetch a minibatch of new requests to be added into the running batch every several decode step. From the history, we use the moving average to estimate a current request rate R_1 measured in how many requests will be added to the waiting queue per period of fetching new requests. We also use the moving average to estimate the number of new requests R_2 that can be added to the running batch for a period. Let rt_i be the coming time of request r_i, ct be the current time, tl_{max} be the maximum allowed first token latency to meet the SLO and $l_{prefill}$ be the maximum prefill latency for a minibatch in history. Each time we generate the new minibatch, we will first abort the requests $R = \{r_k \mid ct - rt_k + l_{prefill} > tl_{max}\}$. Requests in R are highly likely to miss the SLO even if they get scheduled immediately due to the high prefill latency. Then if $R_1 > R_2$, which means the system is temporarily overloaded, we will fetch the newest requests into the minibatch. If $R_1 \leq R_2$, the waiting queue will be shortened if the trend continues. In this case, we will choose from the earliest.

B.1 Proof of Theorem B.1

We first prove that for any admission control strategy that serves l elements, one can always find another admission control strategy that serves the most recent l elements with a larger cumulative reward.

Assume that we serve l elements $q_{s_1}, q_{s_2}, \cdots, q_{s_l}$ in the l timesteps. Assume without loss of generality that q_{s_1} is not among the most recent l elements, and assume that the k-th element is not served with $k \in [n - l, n]$. By definition we know that $s_1 < k$. Now at the time of serving q_{s_1}, we serve q_k rather than q_{s_1}, and keep the rest of the choices in other time steps same. In this case, the number of served queries remains the same. On the other hand, we know that the latency satisfies $t_{s_1} > t_k$ since the k-th element is more recent. This gives that

$$r(t_{s_1}) < r(t_k).$$

Since the reward for other elements does not change, the total reward is increased while the constraint is still satisfied. By repeating the operations until all the elements served are the most recent l elements, we prove that claim.

Next, we prove that serving the most recent l elements in order of $q_{n-l+1}, q_{n-l+2}, \cdots, q_n$ is optimal. For any $i, j \in [n - l + 1, n]$, we assume that $i < j$ and j is first served at time t_1 while i is served at time t_2 with $t_1 < t_2$. Let t_i^a, t_j^a be the arrival time of i, j. The reward for serving i, j in this case becomes

$$r(t_2 - t_i^a) + r(t_1 - t_j^a).$$

Now we show that by swapping the time of serving i, j, the reward does not decrease. This is equivalent to showing that

$$r(t_1 - t_i^a) + r(t_2 - t_j^a) \geq r(t_2 - t_i^a) + r(t_1 - t_j^a).$$

Rearranging the above equation, we know that it is equivalent to

$$\frac{r(t_1 - t_i^a) - r(t_2 - t_i^a)}{t_1 - t_2} \leq \frac{r(t_1 - t_j^a) - r(t_2 - t_j^a)}{t_1 - t_2}.$$

This is true due to the concavity of the reward function, thus finishing the proof.