arXiv:2401.11240v1 [cs.DC] 20 Jan 2024

CARASERVE: CPU-Assisted and Rank-Aware LoRA Serving for Generative LLM Inference

Suyi Li*, Hanfeng Lu*, Tianyuan Wu, Minchen Yu◇, Qizhen Weng‡, Xusheng Chen†, Yizhou Shan†, Binhang Yuan, Wei Wang

HKUST, ◇CUHK-Shenzhen, ‡Shanghai AI Laboratory, †Huawei Cloud

Abstract

Pre-trained large language models (LLMs) often need specialization for domain-specific tasks. Low-Rank Adaptation (LoRA) is a popular approach that adapts a base model to multiple tasks by adding lightweight trainable adapters. In this paper, we present CARASERVE, a system that efficiently serves many LoRA adapters derived from a common base model. CARASERVE maintains the base model on GPUs and dynamically loads activated LoRA adapters from main memory. As GPU loading results in a *cold-start* that substantially delays token generation, CARASERVE employs a *CPU-assisted* approach. It early starts the activated adapters on CPUs for prefilling as they are being loaded onto GPUs; after loading completes, it then switches to the GPUs for generative LoRA inference. CARASERVE develops a highly optimized synchronization mechanism to efficiently coordinate LoRA computation on the CPU and GPU. Moreover, CARASERVE employs a *rank-aware* scheduling algorithm to optimally schedule heterogeneous LoRA requests for maximum service-level objective (SLO) attainment. We have implemented CARASERVE and evaluated it against state-of-the-art LoRA serving systems. Our results demonstrate that CARASERVE can speed up the average request serving latency by up to 1.4× and achieve an SLO attainment of up to 99%.

1 Introduction

Large language models (LLMs) are making significant strides in generative AI [28, 34], enabling a variety of novel applications across numerous domains. Deploying LLMs for domain-specific tasks requires specialization [5, 17], which involves adapting a pre-trained base model to different downstream tasks. Low-Rank Adaptation [2, 5, 9] (LoRA) has emerged as a popular parameter-efficient fine-tuning (PEFT) approach. It preserves the base model's parameters and adds trainable rank decomposition matrices to each Transformer layer. This method significantly reduces the number of trainable parameters, allowing the creation of numerous lightweight LoRA adapters from a single base model. As LoRA gains popularity in LLM deployment, efficiently serving them in a multi-tenant cloud becomes critically important [1, 23].

*Equal contribution

	GPU-Efficient	Cold-Start-Free	SLO-Aware
HF-PEFT [14]	✗	✓	✗
S-LoRA [23]	✓	✗	✗
Punica [1]	✓	✗	✗
CARASERVE	✓	✓	✓

Table 1: Summarization of LoRA serving systems.

However, developing a system for efficient LoRA serving presents non-trivial challenges. One straightforward solution is to *merge* the weights of a LoRA adapter into the parameters of the base model, resulting in an independent, specialized LLM instance of a full size (e.g., HF-PEFT [14]). This approach, though easy to implement, is expensive as it requires duplicating the base model for individual LoRA instances, consuming a substantial amount of GPU memory. Recently, pioneering attempts have been made to enable *base model multiplexing* between LoRA adapters [1, 23], in which the system maintains a shared copy of the base LLM on the GPU and loads LoRA adapters from main memory as requests arrive. Although this approach is GPU-efficient, it results in a severe *cold-start* problem when a requested LoRA adapter is not on GPU and must be fetched from main memory. Depending on the adapter size, a single cold-start can take tens of milliseconds. This delay affects not only the *time-to-first-token* of the newly arrived request but also the *decoding* process of other ongoing requests when *continuous batching* [10, 11, 16, 31] is in use, resulting in an average of 25% latency increase in inference serving in our experiments (§2.3).

We believe a desirable LoRA serving system should exploit base model multiplexing for *GPU-efficient inference*, without incurring high cold-start overhead (*cold-start-free*). Additionally, as a multi-tenant system, it should prioritize meeting users' service-level objectives in latency (*SLO-aware*) by judiciously scheduling their inference requests to heterogeneous LoRA models with varying ranks. Unfortunately, current systems fail to fulfill these requirements (see the summarization in Table 1). To bridge this gap, we present CARASERVE (CPU-assisted, Rank-aware Serve), a multi-tenant LoRA serving system that achieves all three design goals concurrently. We highlight the design approaches and key techniques of CARASERVE as follows:

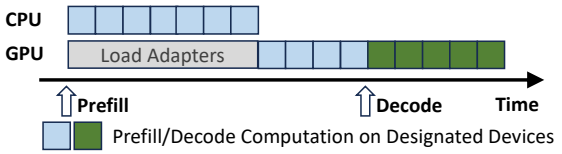

Figure 1: Illustration of CPU-assisted LoRA serving.

CPU-assisted LoRA serving. Similar to the existing LLM-multiplexing solutions [1, 23], CARASERVE maintains the base LLM on GPUs and all LoRA adapters in main memory, which are dynamically loaded onto the GPU as new requests arrive. Yet, instead of waiting for the adapter loading to complete, CARASERVE *concurrently* runs the adapter on CPU to early-start the *prefill* phase. Once the adapter is fully loaded, CARASERVE switches to GPU computation to resume the prefill phase, if not finished, and then proceed to the subsequent decoding phase (Fig. 1), alongside other ongoing requests using continuous batching [10, 11, 16, 31]. This *CPU-assisted* approach effectively mitigates the cold-start overhead, substantially improving decoding efficiency.

Nevertheless, implementing CPU-assisted LoRA serving poses several challenges. LLMs are constructed using the Transformer [29] architecture, which consists of multiple attention layers. During inference, the computed output of the base LLM needs to be synchronized with that of the LoRA models at each layer. Since these computations are split between the CPU and GPU, efficient layer-wise synchronization between the two devices is crucial. Additionally, the frequent triggering of LoRA computations (e.g., 32 times per decoding iteration in Llama2-7B [28]) leads to high invocation overheads, such as inter-process communication (IPC) and data transfer, which can significantly increase inference latency by 79.4%. Moreover, offloading the heavy prefill computation to the CPU may create a new bottleneck due to its limited parallelism compared with GPU.

We address these challenges with a series of techniques. To efficiently coordinate on-GPU LLM computation and on-CPU LoRA computation, we develop a specialized CUDA operator that optimally pipelines the two computations by means of asynchronous memory copy and signaling. Additionally, we employ shared memory to enable fast data exchange between the base LLM process and multiple CPU LoRA processes, eliminating the need for data copying and serialization. This reduces the LoRA invocation overhead to less than 1 ms. Furthermore, we devise a profiling-guided parallelization scheme to scale out LoRA computations across multiple CPUs to eliminate the potential bottleneck. Putting it altogether, CARASERVE can reduce the prefill latency by 57.9%.

Rank-aware request scheduling. In multi-tenant LoRA serving, users often request to utilize heterogeneous adapters with different ranks, which can be batched together to multiplex the base LLM [1, 23]. However, we observe significant performance variations in decoding when batching different sets of heterogeneous LoRA adapters (§2.3). This highlights the need for intelligent request scheduling that takes into account the rank heterogeneity and its impact on decoding. To this end, we establish a *performance model* through extensive system profiling, which can be used to accurately predict the decoding latency for a specific batch of LoRA adapters. Leveraging this information, we design a *rank-aware scheduling algorithm* to enhance cluster-wide performance and meet users' latency SLOs. Specifically, when a new request arrives, the scheduler evaluates all inference servers that possess the required LoRA adapters and calculates a cost score for each server using the performance model. This score measures the additional latency cost and SLO violation on the current ongoing requests if the new request were to be accommodated in that server. The scheduler then selects the server with the minimum cost score and routes the request to it accordingly.

We have implemented CARASERVE as a pluggable LLM serving module in LightLLM [16] and evaluated its performance using Llama2-7B/30B/70B [28] with requests generated from synthetic and real-world traces. Our evaluation highlights that CARASERVE outperforms S-LoRA [23], the state-of-the-art solution, by accelerating the average serving latency of inference requests by up to 1.4×. We also evaluated the rank-aware scheduling algorithm through testbed experiments and large-scale simulations. Compared to popular scheduling policies, including the one used in the existing adapter serving system [1], CARASERVE reduces the average time per token by up to 36.4% and achieves an SLO attainment of 99%.

We will release CARASERVE as an open-source software after the double-blind review process.

2 Background and Motivation

In this section, we give a primer to LLM inference and low-rank adaptation (LoRA). We also discuss the key challenges that arise when serving LoRA models in a multi-tenant cloud.

2.1 LLM Inference

Generative LLM inference computation. LLM inference is a process that involves generating a sequence of output tokens in response to an input prompt, which is a list of tokens. This process consists of two phases: *prefill* and *decoding*. During the prefill phase, the input sequence is used to generate the key-value cache (KV cache) for each transformer layer; the decoding phase then uses the previous KV cache to generate new tokens step-by-step and update the KV cache accordingly. The computation of one transformer layer can be summarized as follows. Denote the batch size by B, the prompt sequence length by L, the hidden dimension of the transformer

by H, and the intermediate size by H'. We have weight matrices of the i-th transformer layer: $\mathbf{W}_K^i, \mathbf{W}_Q^i, \mathbf{W}_V^i, \mathbf{W}_O^i \in \mathbb{R}^{H\times H}$, $\mathbf{W}_1^i \in \mathbb{R}^{H\times H'}$, and $\mathbf{W}_2^i \in \mathbb{R}^{H'\times H}$. During the prefill phase, let $\mathbf{x}^i$ be the input of the i-th transformer layer, and the `key`, `value`, `query`, and `output` of the attention layer respectively specified as $\mathbf{x}_K^i, \mathbf{x}_V^i, \mathbf{x}_Q^i, \mathbf{x}_{\text{Out}}^i \in \mathbb{R}^{B\times L\times H}$. The computation of the cached `key`, `value` is given by $\mathbf{x}_K^i = \mathbf{x}^i \cdot \mathbf{W}_K^i$ and $\mathbf{x}_V^i = \mathbf{x}^i \cdot \mathbf{W}_V^i$. The remaining computation in this transformer layer is given by

$$\mathbf{x}_Q^i = \mathbf{x}^i \cdot \mathbf{W}_Q^i,$$
$$\mathbf{x}_{\text{Out}}^i = f_{\text{softmax}}\left(\frac{\mathbf{x}_Q^i \mathbf{x}_K^{i\,T}}{\sqrt{H}}\right) \cdot \mathbf{x}_V^i \cdot \mathbf{W}_O^i + \mathbf{x}^i,$$
$$\mathbf{x}^{i+1} = f_{\text{relu}}\left(\mathbf{x}_{\text{Out}}^i \cdot \mathbf{W}_1^i\right) \cdot \mathbf{W}_2^i + \mathbf{x}_{\text{Out}}^i.$$

During the decoding phase, let $\mathbf{t}^i \in \mathbb{R}^{B\times 1\times H}$ be the embedding of the current generated token in the i-th layer. The inference computation involves i) updating the KV cache, i.e., $\mathbf{x}_K^i \leftarrow f_{\text{concat}}\left(\mathbf{x}_K^i, \mathbf{t}^i \cdot \mathbf{W}_K^k\right)$, $\mathbf{x}_V^i \leftarrow f_{\text{concat}}\left(\mathbf{x}_V^i, \mathbf{t}^i \cdot \mathbf{W}_V^i\right)$, and ii) computing the output of the current layer:

$$\mathbf{t}_Q^i = \mathbf{t}^i \cdot \mathbf{W}_Q^i,$$
$$\mathbf{t}_{\text{Out}}^i = f_{\text{softmax}}\left(\frac{\mathbf{t}_Q^i \mathbf{x}_K^{i\,T}}{\sqrt{H}}\right) \cdot \mathbf{x}_V^i \cdot \mathbf{W}_O^i + \mathbf{t}^i,$$
$$\mathbf{t}^{i+1} = f_{\text{relu}}\left(\mathbf{t}_{\text{Out}}^i \cdot \mathbf{W}_1^i\right) \cdot \mathbf{W}_2^i + \mathbf{t}_{\text{Out}}^i.$$

The decoding phase continues until a specified condition is met, such as emitting an end-of-sequence (`<eos>`) token or reaching a desired output sequence length.

LLM adaption. Adapting LLMs in a parameter-efficient manner is a popular approach to enhancing their performance for domain-specific tasks or customizing the model inference results to align with human intents [17, 18]. One notable approach is Low-Rank Adaptation or LoRA [9], which introduces an *adapter* to modify the intermediate LLM inference results while keeping the original LLM parameters unchanged. Specifically, given a pre-trained weight matrix $\mathbf{W} \in \mathbb{R}^{H_1\times H_2}$, an adapter consists of two low-rank matrices $\mathbf{A} \in \mathbb{R}^{H_1\times r}$ and $\mathbf{B} \in \mathbb{R}^{r\times H_2}$, where r is the LoRA rank. LoRA adapts this weight matrix to $\mathbf{W}' = \mathbf{W} + \mathbf{AB}$. Let $\mathbf{y}$ be the original output of this layer given by $\mathbf{y} = \mathbf{xW}$. With LoRA adaption, the updated computation becomes

$$\mathbf{y}' = \mathbf{xW} + \mathbf{xAB} = \mathbf{xW}'. \tag{1}$$

The LoRA adapter is highly efficient in terms of parameter space because the rank $r \times (H_1 + H_2) \ll H_1 \times H_2$. Therefore, LoRA adaption is widely applied in the attention modules of transformer-based LLMs [9, 23]. When deploying LoRA-adapted models for inference, the computation load required by the LoRA adapter (**xAB**) is orders of magnitude smaller than that of the original weights **xW** in terms of floating-point operations, if we compute these two parts separately.

computation overheads

2.2 Multi-Tenant LoRA Serving

The need of LLM-multiplexing. A naive way to serve a LoRA adapter [9] is to *merge* its weights into the weights of the base LLM, which introduces no additional computational overhead when deploying the adapted model for inference. However, this approach does not scale to multi-tenant LoRA serving: because one base model can only merge with one LoRA adapter at a time, serving n different LoRA models requires duplicating n copies of the base LLM, wasting GPU memory and missing opportunities for batch inference [11].

In practice, many LoRA models are developed based on common LLM series (e.g., Llama2 [28]), and multiple LoRA models originating from the same LLM can multiplex that LLM for GPU-efficient inference. This can be achieved by computing LoRA adaption **xAB** on the fly and adding this result back to the intermediate results **xW** before subsequent computations. As described in §2.1, the computation of **xAB** is lightweight, and multiple LoRA computations can be batched during inference.

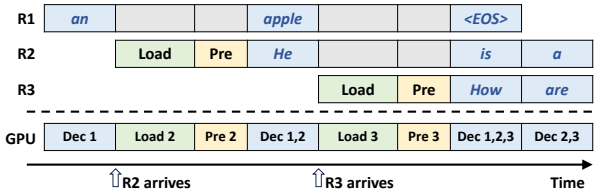

Figure 2: Continuous batching in which the decoding phase (Dec) is preempted to perform prompt processing upon a request arrival, which involves loading the requested LoRA adapter (Load) and prefilling (Pre).

Continuous batching. Existing LLM serving systems employ a *continuous batching* strategy optimized for LLM's iterative auto-regressive generation process [10, 11, 16, 31]. Continuous batching operates at the iteration level, where completed requests are immediately removed from the running batch after each iteration to make room for new requests to join. This allows a new request to be incorporated in just one iteration without waiting for the entire batch inference to complete. Continuous batching significantly improves the token generation throughput while minimizing the request queuing delays. Fig. 2 illustrates this batching process used in existing systems [10, 11, 16], where the decoding and prefill phases interleave as new requests arrive. Upon a request's arrival, the decoding phase (Dec) is preempted to perform prompt processing, which involves loading the requested LoRA adapter (Load) and prefilling (Pre). Once completed, the new requests join the running batch, and the system combines them together to continue the decoding process.

preemption for prefill -> decode

2.3 Challenges

However, simply enabling LLM-multiplexing and continuous batching is insufficient to achieve optimal performance for multi-tenant LoRA serving, as it results in two challenges.

C1: High cold-start overhead. To save GPU memory, exist-

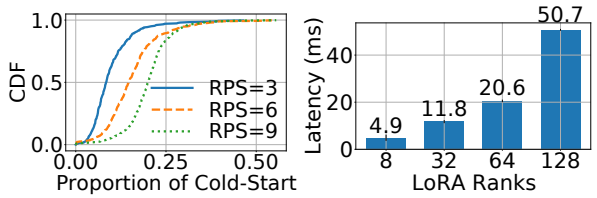

Figure 3: Left: The distribution of cold-start overhead during the entire token generation of each request. Right: The cold-start latency of loading a single LoRA adapter of different rank onto GPU. The adapter applies to the $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v$ of a Llama2-7B on an A10 GPU instance.

ing systems only cache the base LLM on GPU while keeping all its LoRA adapters in host memory [1, 23]. When a new request arrives, the system fetches the corresponding adapter from the host to the GPU, leading to an *adapter loading* phase that must complete before the prefill phase begins (Fig. 2). This results in a severe *cold-start* problem, where loading an adapter from the host to a GPU can take between a few to tens of milliseconds, depending on the adapter size (Fig. 3-Right). Cold-start degrades the service responsiveness, measured by time-to-first-token [4, 23]. Moreover, under continuous batching, each time a new request arrives, the decoding phase of in-flight requests is blocked until the new arrival's prefill phase completes (Fig. 2). As new requests keep arriving, their cold-start overhead *cumulatively delays* the token generation of an in-flight request (as shown in Fig. 2, where R1 experiences two cold-starts due to the arrivals of R2 and R3). We empirically validate this issue by multiplexing a Llama2-7B model with a group of 512 LoRA adapters (rank=64). These adapters have skewed popularity (Fig. 12) following the Microsoft Azure Function (MAF) trace [21]. We configured Poisson request arrivals with various aggregate loads. Fig. 3-Left shows the proportion distribution of the cold-start overhead, which, on average, accounts for 10%, 16%, and 20% of the entire request serving time when the aggregate load is 3, 6, and 9 requests per second, respectively.

To avoid cold-start, a simple approach is to pre-cache all LoRA models in GPU. However, this approach is expensive: a single rank-64 adapter that adapts three attention weights $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$ of a Llama2-7B model [28] demands approximately 100 MiB, equivalent to the size of a KV cache of 200 tokens. S-LoRA [23] suggests using predictive pre-fetching, yet without providing details. Given that inference requests to individual models are highly bursty [8, 33], frequent mispredictions and cold-starts are expected. Punica [1] uses asynchronous loading to avoid blocking subsequent decoding iterations. However, new requests still need to undergo the adapter loading phase, leading to the extended time-to-first-token [1].

C2: Request scheduling for heterogeneous LoRA serving. In multi-tenant LoRA serving, users often request to use heterogeneous LoRA adapters with varying ranks [23]. These heterogeneous adapters can be batched together to multiplex

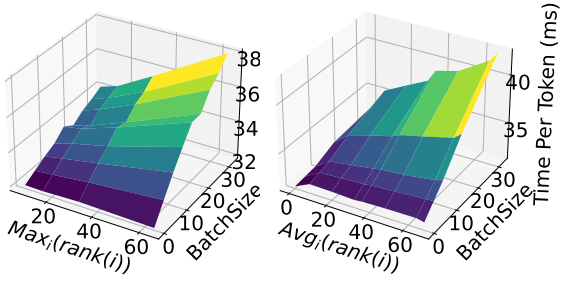

Figure 4: The varying decoding latency of batching heterogeneous LoRA adapters. Left: The performance of Punica's BGMV [1] is determined by the batch size and the *maximum rank*. Right: The performance of S-LoRA's MBGMV [23] depends on the batch size and the *average rank* in the batch.

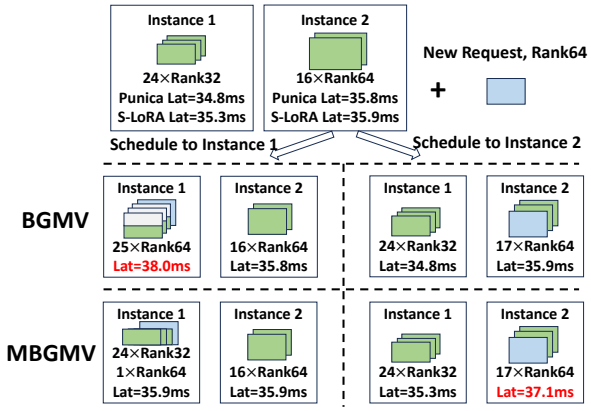

Figure 5: An example of rank-aware LoRA scheduling with a decoding latency SLO of 36 ms. With Punica's BGMV, scheduling the new request to Instance 2 meets the SLO; with S-LoRA's MBGMV, scheduling it to Instance 1 preserves the SLO.

one base LLM using specialized kernel implementations, such as the Batched Gather Matrix-Vector Multiplication (BGMV) kernel in Punica [1] or the Multi-size Batched Gather Matrix-Vector Multiplication (MBGMV) kernel in S-LoRA [23]. Specifically, when batching a set of heterogeneous LoRA adapters, BGMV pads adapters of smaller ranks to the highest rank to perform batch operations, while MBGMV does not use padding [23]. As a result, BGMV's performance is determined by the maximum rank in the batch, whereas MBGMV's performance depends on the average rank. We measure the decoding latency of batch serving heterogeneous LoRA adapters using these two kernels with various batch configurations, and the results are depicted in Fig. 4. We observe significant performance variations when batching different sets of heterogeneous adapters. This highlights the need for *intelligent request scheduling* that takes into account the rank heterogeneity and the batching performance of a specific kernel implementation.

To illustrate this point, we refer to a toy example shown in Fig. 5. In this example, Instance 1 is handling 24 requests with LoRA rank=32, while Instance 2 is running 16 requests with

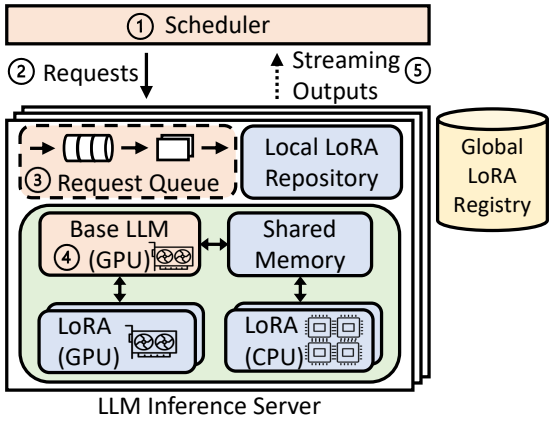

Figure 6: An architecture overview of CARASERVE.

rank=64. Using Punica's BGMV kernel, the decoding latencies for Instances 1 and 2 are 34.8 ms and 35.8 ms, respectively. With S-LoRA's MBGMV, the latencies are 35.3 ms for Instance 1 and 35.9 ms for Instance 2. Assume a decoding latency SLO of 36 ms, and we need to determine the optimal schedule for a new incoming request with rank=64. With the BGMV kernel, assigning this new request to Instance 2 would meet the SLO, while sending it to Instance 1 would increase the maximum rank of the batched requests to 64, resulting in an SLO violation due to the processing of 25 higher-rank requests on Instance 1. Things become different when it comes to S-LoRA's MBGMV kernel, as the latency is proportional to the total LoRA ranks within a batch. Since Instance 2 already has a higher sum of batch ranks, its latency is higher than that of Instance 1. Therefore, scheduling the new request to Instance 1 preserves the SLO, while routing it to Instance 2 would lead to an SLO violation.

Despite the significant impact of request scheduling, existing LoRA serving systems [1, 23] provide no optimization to it, resulting in significant delays that violate SLOs (§7.5).

3 CARASERVE Overview

In this section, we provide a high-level overview of CARASERVE, a LoRA serving system that efficiently tackles the two challenges mentioned earlier. CARASERVE uses a *CPU-assisted* approach to hide the long cold-start latency. It uses CPUs to simultaneously execute the requested LoRA adapter while loading it onto the GPU, effectively overlapping the adapter loading (cold-start overhead) with the *prefill* computation (§4). CARASERVE also optimizes the scheduling of inference requests to heterogeneous LoRA adapters using a *rank-aware* scheduling algorithm, significantly enhancing cluster performance and SLO compliance (§5). Fig. 6 illustrates the system architecture, which consists of a cluster of LLM inference servers, a scheduler, and a global LoRA registry.

LLM inference server. Each LLM inference server maintains a long-running service of the base LLM on the GPU. It also stores a set of heterogeneous LoRA adapters in an in-memory *local LoRA repository*. During inference, the server coordinates LoRA computations on the CPU and GPU to avoid cold-start. Specifically, it adapts the BGMV kernel from [1] to perform LoRA computation efficiently on the GPU. For CPU-based LoRA execution, it utilizes three techniques to enhance its efficiency: asynchronous invocation, shared memory, and profiling-guided parallelization, which we elaborate in §4.

Scheduler. The scheduler receives user requests and routes them to the appropriate servers to meet the SLOs. To guide the scheduling decision, it uses a performance model to predict the latency cost by jointly considering the rank heterogeneity of the serving batch and the underlying kernel implementation, which we explain in §5.

Global LoRA registry. The global LoRA registry stores the metadata of all LoRA adapters, such as the LoRA ranks, the path to their weights file, etc.

Workflow. As illustrated in Fig. 6, new requests arrive at the scheduler (①), which uses the rank-aware scheduling algorithm described in §5 to route them to appropriate inference servers (②). Following the continuous batching strategy [31], the LLM inference server fetches requests from the request queue (③) and provides generative inference services using the corresponding LoRA adapters (④). New tokens generated by the LLM are then streamed back to the users (⑤).

4 CPU-Assisted LoRA Serving

In this section, we present the design and implementation of CPU-assisted LoRA serving. We begin by describing LoRA computation on GPU and CPU and discussing the challenges of efficiently combining the two executions to address the cold-start problem (§4.1). We then present three optimization techniques that address these challenges (§4.2).

4.1 LoRA Computation on GPU and CPU

A parameter-efficient adapter, LoRA requires lightweight computation and can run on either GPU or CPU.

GPU LoRA. As the base LLM is "pinned" on GPU, running LoRA adapters on the same device saves the communication overhead and is usually more efficient than running them on CPU. To maximize the token throughput, LoRA computations (i.e., **xAB** in Eq. (1)) are batched in each attention layer during base LLM inference. This can be achieved with a specialized CUDA operator [1, 23]. In CARASERVE, we adapt the Batched Gather Matrix-Vector Multiplication (BGMV) operator [1], which parallelizes the LoRA weight gathering and computation for efficient execution. The LoRA output is then added to the base output in the self-attention computation,

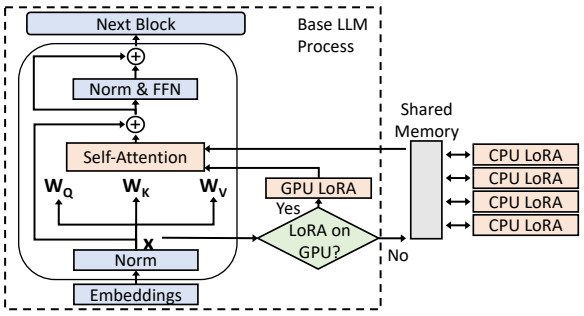

Figure 7: Illustration of coordinated LoRA computation on GPU and CPU per transformer block's attention layer.

following in Eq. (1). For an efficient implementation, we incorporate the operators of GPU LoRA computation into the base LLM inference process, as shown in Fig. 7.

CPU LoRA. LoRA computation can also be executed using the CPU, which requires *layer-wise synchronization* with the base LLM inference running on the GPU. Specifically, at each attention layer, the base inference process transfers the input tensor **x** in Eq. (1) from the GPU device memory to the host memory (Fig. 7). The CPU LoRA process then performs computation and transfers the result **xAB** back to the GPU device. In the meantime, the base inference process proceeds to compute **xW**, which is finally adapted with the received LoRA output following Eq. (1). Although CPU LoRA requires synchronization, it can start immediately because the LoRA weights are already in memory. We hence utilize it to address the cold-start problem that arises in GPU LoRA (**C1** in §2.3).

Mitigating GPU cold-start with CPU assistance. As illustrated in Fig. 1, when a new request arrives and the corresponding adapter is not available on the GPU, the server fetches it from host memory and, in the meantime, starts its *prefill* computation using the CPU. Once the adapter is fully loaded, the GPU LoRA takes over, finishing the remaining prefill computation not done by the CPU, if any, and the subsequent decoding process. Fig. 7 illustrates how CPU and GPU LoRA computations are coordinated in our design, where we run CPU LoRA adapters as isolated, concurrent processes for resource/failure isolation and improved performance.

Challenges. Though hosting LoRA computation in isolated CPU processes effectively addresses the *cold-start* problem, it poses three challenges to system implementation. First, running LoRA in CPU processes requires layer-wise synchronization between the GPU-based LLM inference to ensure data validity. Second, frequent triggering of LoRA computation in each attention layer leads to high invocation overhead, such as inter-process data transfer. Third, using CPU to compute adaptation can be slow given its limited parallelization capability, especially when the input prompt is long.

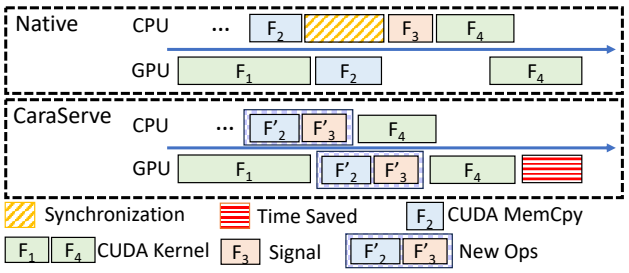

Figure 8: Execution timeline of Native LoRA Invocation and LoRA Invocation with CARASERVE's operator in base LLM process. CPU LoRA is ignored for simplicity.

4.2 Efficient GPU-CPU LoRA Coordination

In this subsection, we tackle the system challenges mentioned earlier with three optimization techniques.

Sync-free CPU LoRA invocation. Most LLM serving systems achieve low latency through asynchronous GPU computation in PyTorch-like frameworks [10, 11, 16, 23, 28]. However, adapter serving requires careful coordination between base LLM inference running on GPU and LoRA invocation running on CPU to ensure correctness and good performance.

In native PyTorch, having the base LLM process invoke CPU LoRA requires explicit synchronization, which *blocks* subsequent kernels from launching. To illustrate this problem, we refer to Fig. 8-Top, which depicts the native PyTorch invocation timeline from the base LLM process's perspective.* The CUDA kernel F_1 computes the input matrix **x**. In the meantime, the base LLM process issues F_2, a CUDA `MemCpy` kernel, to transfer the input matrix to the host memory for CPU LoRA's access. Once the data transfer completes, the base process uses a *signaling operator* F_3 to notify CPU LoRA processes to compute **xAB**. It then launches the next CUDA kernel F_4 following F_1. This implementation requires explicit synchronization (shown as a yellow block with slashes) to ensure that the memory copy (F_2) completes before the signaling (F_3). However, this synchronization blocks the subsequent F_4 from launching, resulting in significant inference delay and GPU underutilization.

To address this issue, we introduce a customized operator that eliminates explicit synchronization by fusing an asynchronous `MemCpy` kernel with a signaling kernel. As shown in Fig. 8-Bottom, instead of relying on synchronization, we fuse F_2 and F_3 into an *asynchronous* CUDA kernel $[F_2', F_3']$, where F_2' performs asynchronous `MemCpy` and F_3' asynchronously signals the intended CPU LoRA processes through shared memory. As a result, the fused kernel $[F_2', F_3']$ can be added to the GPU device queue without waiting for the completion of F_1. Note that data validity is preserved in this case because CUDA device queue follows a sequential, strict first-in-first-

*Note that CPU LoRA processes (i.e., CPU calculation for **xAB**) are not depicted in Fig. 8 because they are identical in both implementations.

out execution ordering. Since the new operator requires no explicit synchronization, subsequent base model kernels, such as F_4, can launch without being blocked, eliminating unnecessary synchronization overhead. Our experiments in §7.4 demonstrate that our kernel can reduce the latency of each prefill iteration by 16% compared with PyTorch's native implementation.

Shared memory data transfer. Transferring data and signals between the base LLM process and the isolated CPU LoRA processes requires inter-process communication (IPC). This is a one-to-N communication involving one base LLM inference process and multiple CPU LoRA processes. (We explain why multiple CPU LoRA processes later.) We utilize shared memory for fast inter-process data transfer, eliminating the need for data copying and serialization (Fig. 7). After the base LLM process executes our customized operator (see Fig. 8), the CPU LoRA processes will soon be signaled to start reading the input matrix **x** from the shared memory and perform the computation **xAB**. They then write **xAB** back to the shared memory and notify the LLM inference process to incorporate the adaptation results (Eq. (1)). Micro-benchmark evaluations (§7.4) demonstrate that the use of shared memory reduces data transfer overhead to less than 1 ms (Fig. 17), substantially outperforming the message passing IPC employed by existing LLM frameworks [16].

Profiling-guided LoRA parallelization. Given that the CPU has lower computing power and limited parallelization capability compared to the GPU, performing LoRA adaptation using a single CPU is not scalable. Therefore, we propose a profiling-guided parallelization scheme to accelerate LoRA adaptation using multiple CPU cores. As discussed in §2.1, the adaptation computation is **xAB**, where $\mathbf{x} \in \mathbb{R}^{B \times L \times H}$ is the input matrix for B requests with L tokens, totaling $B \times L$ tokens. We first profile the performance achieved by a single core under varying workloads (Fig. 18-Left) and set the maximum workload for a single CPU, which is the maximum number of tokens a CPU core can handle for computation. For example, if one core can handle c tokens, we allocate $\lceil \frac{L}{c} \rceil$ cores for computing the adaptation results of each request with weight matrix **W**. Each core is dedicated to an isolated CPU process to avoid interference. Specifically, the CPU process reads a slice of **x** from the shared memory region, performs the computation, writes the results back to the shared memory, and notifies the base LLM process accordingly. Compared to PyTorch's native multi-threading module [7], this approach achieves 1.7× speedup when using 8 CPUs for the same workload (Fig. 18-Right).

Putting it altogether, our design, as demonstrated in §7.2, can accelerate the request serving by 1.4× on average.

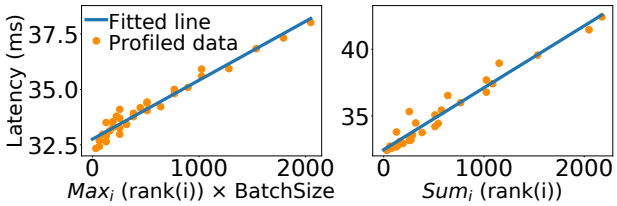

Figure 9: Performance models for BGMV (Left) and MBGMV (Right) kernels. Both linear regression models achieve a high coefficient of determination (R^2) of 0.96.

5 Rank-Aware Scheduling

In a multi-tenant LoRA serving system, user requests can trigger the use of heterogeneous LoRA adapters with varying ranks. As discussed in §2.3, the heterogeneity in adapter ranks directly affects the performance of multi-tenant LoRA serving systems. Therefore, the scheduling strategy for handling these requests is crucial for enhancing system efficiency (**C2**): a suboptimal strategy can drive the adapter heterogeneity in a server to a non-ideal setting that slows down token generation for both new and ongoing requests. To address this, an effective scheduler needs to be aware of the heterogeneity-performance model, and make optimal scheduling decisions to achieve high SLO attainment.

Performance modeling. The goal of performance modeling is to establish a correlation between rank heterogeneity in a batch of LoRA requests and its impact on serving performance. This enables the scheduler to make informed scheduling decisions to meet SLOs. Under continuous batching (§2.2), when new requests are routed to a server, the server's running batch size increases, and the batch's rank heterogeneity changes as well. To efficiently serve a batch of LoRA requests, existing works [1, 23] provide two CUDA kernels for computing the adaption **xAB**: the padding-based BGMV and padding-free MBGMV (§2). We characterize these kernels using NVIDIA Nsight Compute [6] and observe that both kernels consume over 70% of the GPU memory bandwidth, suggesting that their performance is bounded by the GPU memory bandwidth.

Based on the characterization of kernels, we develop generic performance models to predict the *prefill* and *decoding* latency of a specific batch of heterogeneous adapters. These models are created through lightweight serving performance profiling, involving varying batch sizes and heterogeneous adapters on a specific GPU. We present the performance models tailored for both BGMV [1] and MBGMV [23]. For the padding-based BGMV kernel, where lower-ranked LoRAs require padding to match the highest rank for the BGMV operation, we observe that the serving performance of decoding latency is almost linear to the product of batch size and the *maximum* rank encountered in the batch (see Fig.9-Left). On the other hand, S-LoRA's MBGMV [23] modifies the BGMV kernel to eliminate padding, improving performance with highly

heterogeneous LoRA ranks but introducing additional performance overhead for computing homogeneous ranks. Through profiling, we find that under MBGMV, the serving performance scales linearly with the *sum* of LoRA ranks in a batch of heterogeneous adapters (Fig. 9-Right). Denoting the adapter rank of request i as $rank(i)$, we present performance models for these two kernels on a batch of requests $\mathcal{S}$ as two linear functions with parameters α and β, inspired by [13]:

$$\text{PERF}_{\text{BGMV}}(\mathcal{S}) = \alpha_B \cdot |\mathcal{S}| \cdot \text{Max}_{i\in\mathcal{S}} rank(i) + \beta_B$$
$$\text{PERF}_{\text{MBGMV}}(\mathcal{S}) = \alpha_M \cdot \text{Sum}_{i\in\mathcal{S}} rank(i) + \beta_M$$

As depicted in Fig. 9, our linear performance models accurately fit the profiled data. Both models achieve a high coefficient of determination (R^2) of 0.96, in that $R^2 = 1$ indicates a perfect fit of the linear model to the data.

Scheduling policy. Using the established performance models, we develop a rank-aware scheduling algorithm (Algo. 1) for heterogeneous LoRA requests. Upon receiving a new request, the scheduler gathers information about ongoing requests from all available LLM inference servers. The scheduler identifies potential candidate servers by matching the base LLM, adapter, and GPU memory availability. If multiple candidates are found, the scheduler calculates a total cost score for each candidate server based on the performance model. This cost score measures the impact of the new requests on the performance of the server's ongoing requests. If serving the new request would cause a violation of the SLO, the cost score is assigned a large penalty. The scheduler then selects the server with the minimum cost score to handle the new request. In our evaluation (§7.5), this rank-aware scheduling algorithm achieves a high SLO attainment of up to 99%, substantially outperforming other baseline strategies.

Algorithm 1: Rank-aware Scheduling Policy

Input: Performance models for *Prefill* and *Decoding*: $PrePerf(\cdot)$, $DecPerf(\cdot)$; average response length: avg_resp_len

```
while True do
    Request i arrives;
    candidates ← available LLM inference servers
    for instance in candidates do
        running_batch, queue = instance.GetStats()
        cost = CalcCost(i, running_batch, queue)
        requests = len(running_batch) + len(queue)
        instance.total_cost = cost * requests
    end
    best = min(candidates, key=lambda x: x.total_cost)
    best.serve(i)
end
Function CalcCost(req, running_batch, queue):
    exists = running_batch + queue
    # calculate additional prefilling time
    Δ_prefill = PrePerf(queue + req) - PrePerf(queue)
    # calculate additional decoding time per token
    Δ_decode = DecPerf(exists + req) - DecPerf(exists)
    cost_score = (Δ_prefill / avg_resp_len) + Δ_decode
    if DecPerf(exists + req) > SLO then
        cost_score += penalty_score
    end
    return cost_score
```

6 Implementation

LLM inference server. We implemented CARASERVE's LLM Inference Server on top of LightLLM [16], an LLM serving framework based on PyTorch [19] and Triton [27]. Specifically, we extended its Llama2 inference module to incorporate our LoRA adapters. This allows for easy integration with different LLMs and other popular LLM inference frameworks such as vLLM [11]. We implemented GPU LoRA adapters by adapting the BGMV kernels in Punica [1]. Regarding CPU LoRA, we implemented a custom CUDA kernel (described in §4.2) as a PyTorch Extension using PyBind11, and built CPU LoRA on top of PyTorch. Each CPU LoRA adapter runs as an isolated process, binding to one CPU core using the numactl command. To enable efficient batch inference, we utilize the request queue in LightLLM, which facilitates the continuous batching mechanism [11, 31].

Support model parallelism. We employ tensor parallel techniques [25] to support base LLMs that require multiple GPU devices. Tensor parallelism involves partitioning a weight matrix into multiple chunks along a specific dimension. Each GPU device holds only one chunk of the entire weight matrix and performs a portion of the computation in parallel [12]. Tensor parallelism may require communication between the participating GPU devices for output merging. To enable tensor parallelism for LoRA computation, CARASERVE partitions the LoRA adapter weights (**B** in Eq.(1)) using the same strategy as that of the base LLMs. It performs the computation and incorporates the adaptation results into the inference intermediates in-place, causing no extra communication overhead.

Scheduler & global LoRA registry. In our prototype, we implemented the scheduler using Python Flask. It serves as the frontend that receives requests and routes them to LLM inference servers based on Algo. 1. For the global LoRA registry, we utilized SQLite in our prototype.

7 Evaluation

We evaluate CARASERVE using both synthetic and scaled production workloads [21] in terms of the LLM inference server's serving efficiency (§4) and the scheduler performance across multiple servers (§5). Our evaluation highlights include:

- CARASERVE achieves efficient multi-tenant LoRA serving on both synthetic and real-world workloads, outperforming strong state-of-the-art baselines, e.g., S-LoRA [23] (§7.2).
- CARASERVE is compatible with model parallelism to support LLMs that require multiple GPUs (§7.3).

Base Model	Hidden Size	Layers	GPU Config.
Llama2-7B	4096	32	A10 (24G)
Llama2-13B	5120	40	2 × A10 (24G)
Llama2-70B	8192	80	4 × A100 (80G)

Table 2: Model and GPU configurations.

- CARASERVE's optimizations in CPU LoRA execution are effectively illustrated by various micro-benchmarks (§7.4).
- CARASERVE's scheduler achieves high SLO attainment and improves the performance as a cloud service (§7.5).

7.1 Experimental Setup

Model and server configurations. We adopt Llama2 [28] models with 7B, 13B and 70B parameters for evaluation (details in Tab. 2), where LoRA adapters are applied to $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ (§2) in LLM's attention layers following the standard settings [5, 9, 23][*].

Metrics. We use the following metrics in evaluation, which are considered essential in user-facing LLM serving [4, 23].

- ***Time to first token.*** It measures how quickly users start getting the model's output after entering their prompts. Low waiting times for a response are essential in real-time interactions. This metric reflects the time required to process the prompt and then generate the first output token.
- ***Time per token.*** It measures the time on average to generate an output token for each user. This metric corresponds with the perceived "speed" of the model.
- ***Request latency.*** It measures the overall time it takes for the model to generate the full response for a request.

Baselines. We consider the following baselines.

- CACHED represents an Oracle method where all required LoRA adapters are pre-cached in unlimited GPU memory. It has no adapter loading overhead, thus achieving performance upper bound.
- ONDMD loads LoRA adapters on demand. It will suffer from the *cold-start* overhead if the required LoRA adapters are not on GPUs.
- S-LORA [23] represents a state-of-the-art multi-tenant LoRA serving framework, which is also built on top of LightLLM [16]. It loads LoRA adapters on demand and uses an adapted CUDA kernel for GPU LoRA computation.

Note that we equip baselines other than S-LoRA with the `BGMV` kernel [1] to perform GPU LoRA computation for a fair comparison in the single GPU case.

Workloads. We use both synthetic and scaled production workloads in our evaluation.

[*]Following the setting in [1,23], we use dummy weights for LoRA models, which do not affect system performance.

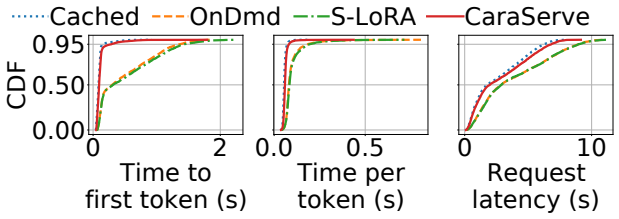

Figure 10: End-to-end results with Llama2-7B.

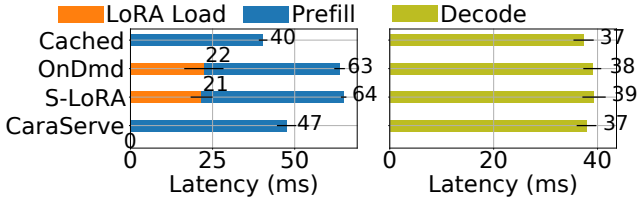

Figure 11: Prefill and decoding latency at LLM inference server. CARASERVE hides the LoRA adapter loading overhead by overlapping loading and CPU computation.

- ***Synthetic workload.*** The aggregate request traffic to an LLM server follows Poisson processes with varying intensities, widely used in approximating simulated invocations [1, 33]. Similar to [1], each request targets a distinct adapter and hence undergoes the adapter loading phase.
- ***Scaled production workload.*** We use the MAF trace [21] to generate a scaled production workload widely used to emulate model serving workloads [8, 15, 20, 33]. The trace contains invocation patterns of different functions, and we regard each function as one LoRA adapter. We randomly group the LoRA adapters. Each LLM inference server hosts a group of adapters and receives the aggregated request traffic from all the LoRA adapters it hosts. Within a group, adapters have varying probabilities of being invoked, proportional to their invocation frequency in the original trace. Fig. 12 shows the invocation probability density function.

For both workloads, we set each request's input prompt and output length according to the Alpaca dataset [11, 26], which contains input and output texts of real LLM services. Like S-LoRA [23], we run each workload for 5 minutes.

7.2 End-to-End Performance on a Single GPU

We first evaluate CARASERVE on the synthetic and scaled production workloads on an A10 GPU serving Llama2-7B.

Synthetic workloads. We generate traces using a Poisson process with an aggregate $RPS = 9$ and set the LoRA adapter rank to 64. We measure the performance of each baseline using the metrics discussed in §7.1. Fig. 10 plots the CDFs of time metrics, demonstrating that CARASERVE can rival CACHED and outperform ONDMD/S-LORA.

Compared to the CACHED baseline, ONDMD/S-LORA introduce prohibitively high overhead, increasing time to

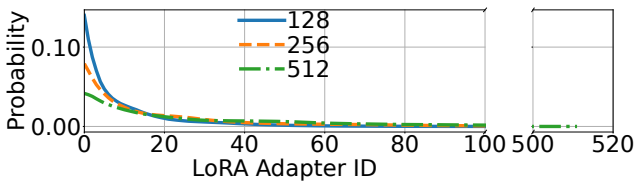

Figure 12: LoRA Invocation Probability Mass function. X-axis: ID sorted by invocation probability in descending order.

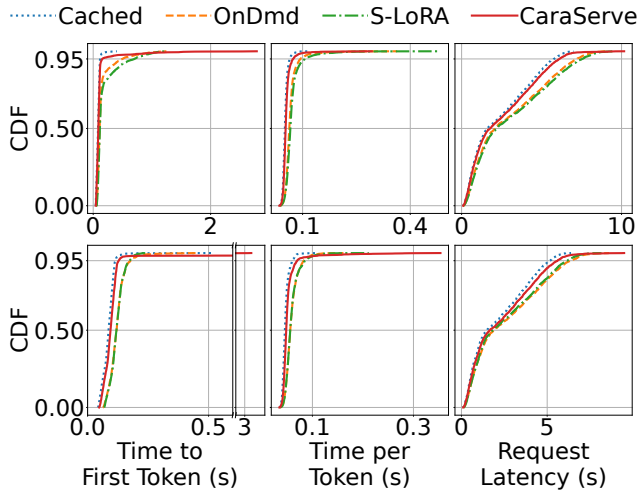

Figure 13: Sensitivity analysis of different Ranks and Traces. Top: $RPS = 9, rank = 32$; Bottom: $RPS = 6, rank = 64$.

first token by 412%/451%, time per token by 71%/78%, and request latency by 50%/50% on average. However, CARASERVE rivals the performance of CACHED by introducing tolerable overheads. On average, CARASERVE reduces the time to the first token latency overhead to 22%, time per token overhead to 11%, and the end-to-end request latency overhead to 9%. Fig. 11 explains CARASERVE's advantage from the LLM inference server's side. We can see that the latency of each *decoding* iteration is similar across all baselines, while ONDMD/S-LORA have a long *prefill* iteration due to the adapter loading overhead. On the other hand, CARASERVE leverages the CPU-assisted design (§4) to avoid the adapter loading overhead in *prefill* iteration.

Sensitivity Analysis. Two factors affect the benefits achieved by CARASERVE (§2). The first is LoRA rank — smaller rank leads to shorter loading latency. We evaluate each baseline with adapter $rank = 32$ and aggregate $RPS = 9$. Fig. 13-Top shows that although smaller LoRA ranks decrease overhead, ONDMD/S-LORA introduces a considerable amount of overhead compared to the CACHED: 88%/126% for time to first token, 28%/36% for time per token, and 25%/31% for request latency on average. CARASERVE outperforms by introducing minimal overhead: 36%, 5%, 6% for the three metrics respectively. The second factor is the workload, which determines the frequency of LoRA loading. Higher request traffic results in increasing *prefill* phases

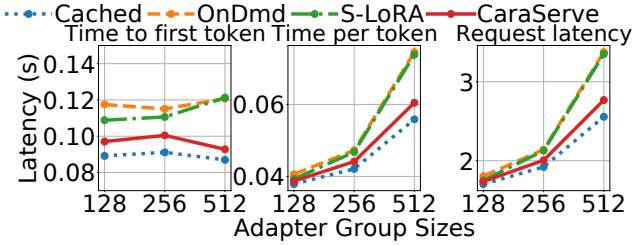

Figure 14: Baseline performance with varying number of LoRA adapters under MAF workloads.

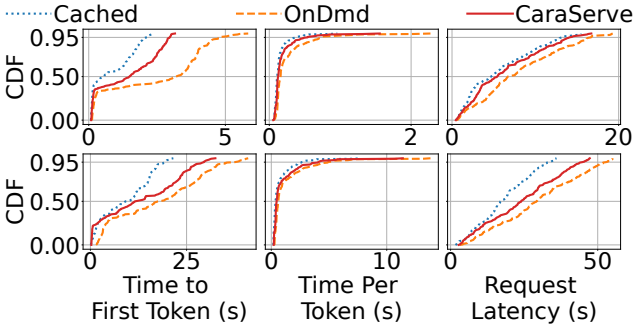

Figure 15: Evaluation on Llama2-13B (Top) and Llama2-70B (Bottom) models with $RPS = 6, rank = 64$.

and adapter loading (§2.3). We evaluate each baseline with a lighter traffic with aggregate $RPS = 6$ and the $rank = 64$. Similar to reducing LoRA rank, reducing workload decreases the overheads of ONDMD/S-LORA to 42%/41%, 25%/25%, 24%/20% for the three metrics respectively (Fig. 13-Bottom). CARASERVE maintains superior with minimal overhead: 1%, 10%, 9% for the three metrics, respectively.

Scaled production workloads. We next evaluate CARASERVE on a production workload based on the MAF trace [21]. Fig. 12 illustrates the skewed distribution of function popularity. We evaluate each baseline with an increasing number of LoRAs and their workloads in a single LLM inference server. More LoRA adapters mean heavier request loads, and each new request is more likely to invoke a new LoRA adapter that needs to be loaded onto GPU on demand (Fig. 12). The average aggregate *RPS* for 128/256/512 adapters is 1.5/3.6/7.7, respectively, scaled from the original trace.

We measure each request's serving performance using the metrics defined in §7.1. Fig. 14 presents the results. When 128 LoRA adapters are in a single server, the impact of *cold-start* is negligible because the invocation traffic is low, and most new requests do not require adapter loading. Compared to CACHED, ONDMD/S-LORA/CARASERVE increase time to first token by 31%/22%/9%, time per token by 8%/3%/3%, and request latency by 6%/3%/2% on average.

However, as the number of LoRA adapters increases to 512, adapter loading introduces prohibitively high overhead, hindering a system from scaling to host a large number of LoRA adapters. In comparison to the CACHED baseline,

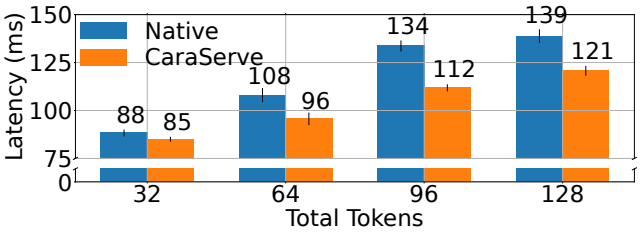

Figure 16: Prefill performance of different kernels on Llama2-7B model. Native: PyTorch default kernels. CARASERVE: Implementation with our optimized kernels (§4.2).

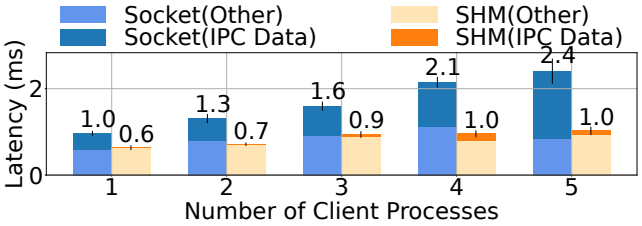

Figure 17: CPU LoRA computation time. Each process receives data of 16 tokens. *Socket*: Domain socket for inter-process communication (IPC). *SHM*: Shared memory for IPC. *IPC Data*: Time for transfering data to another process via IPC. *Other*: Time for all other operations.

ONDMD/S-LORA/CARASERVE increase first token latency by 39%/39%/7%, time per token by 34%/32%/7%, and request latency by 31%/31%/8% on average. These results suggest that the *cold-start* issue prevents ONDMD/S-LORA from scaling to accommodate a large number of LoRA adapters. Nevertheless, CARASERVE performs better than its competitors by rivaling the performance of the CACHED baseline.

7.3 End-to-End Performance on Multi-GPUs

We evaluate each baseline with Llama2-13B and Llama2-70B with two A10 GPUs and four A100 GPUs, respectively. We compare CARASERVE with CACHED and ONDMD since existing works [1,23] have not released their code in multi-GPU settings. For the Llama2-70B model, we adopt the `torch.bmm` operator instead of the `BGMV` kernel from Punica [1], since it does not support the key/value matrix shape of the Llama2-70B model. For both models, we use a synthetic Poisson arrival rate with $RPS = 6$ and prompts from the Alpaca dataset [26].

Fig. 15 plots the CDFs of requests' serving performance regarding the three metrics. CARASERVE gains a much better performance than the on-demand loading methods. On average, CARASERVE achieves a 20.2%/18.5% speedup on the end-to-end request latency for Llama2-13B and Llama2-70B models. Compared with ONDMD, CARASERVE reduces its cold-start overhead by over 50%.

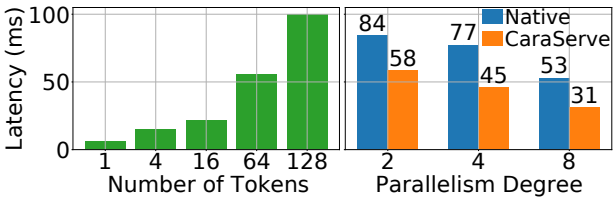

Figure 18: Left: CPU computation time of **xAB** in the *prefill* phase for prompts of different length. Right: Comparison of CPU computation time for 128 tokens with different CPU parallelism. Native: PyTorch native multi-threading [7].

7.4 Microbenchmark Evaluation

In this section, we evaluate CARASERVE's optimizations on CPU LoRA computation (§4.2) at a micro-benchmark level.

Sync-free CPU LoRA invocation. To analyze the performance of our optimized CPU LoRA invocation kernel, we use the Llama2-7B model on one A10 GPU to measure the *prefill* latency of the PyTorch's native implementation and our optimized kernels. As Fig. 16 shows, our customized kernel performs better than the default PyTorch kernel. As the total number of tokens in *prefill* phase increases, CARASERVE's kernel gains up to a 16% performance increase.

Shared memory data transfer. We compare the latency of computing CPU LoRA with different IPC methods: shared memory and UNIX domain socket. We measure the time it takes to perform LoRA computation and the data round trip cost. We increase the number of receiver processes to represent the increase in the number of CPU LoRA processes (§4.2). Fig. 17 shows that as the number of receiver processes increases, the domain socket-based approach suffers from linear time increase in initialization and serialization overhead, whereas the shared memory-based approach obtains near-constant performance.

Multi-CPU computation. We first profile the LoRA computation performance during a *prefill* phase with a single CPU. We profile it with different workloads (number of tokens to process). We run the profiling on a Llama2-7B model with a single A10 card. As shown in Fig. 18-Left, the CPU has limited parallelism and does not scale to fit high workloads. Fig. 18-Right illustrates the performance of *prefilling* a prompt of 128 tokens with CARASERVE's multi-CPU design (§4.2) or the native multi-core utilization of PyTorch multi-threading module [7]. We can see that CARASERVE's design achieves up to 1.7× speedup.

7.5 Scheduler

In this section, we evaluate the effectiveness of our scheduling policy (§5), which achieves a higher SLO attainment.

Baselines. Upon the arrival of new requests, we consider the following scheduling baselines for comparison:

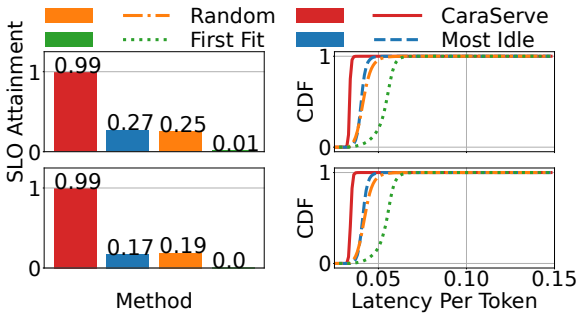

Figure 19: [**Simulation**] Scheduler performance with S-LoRA's MBGMV and CARASERVE's BGMV backend on 60 instances. Top: SLO attainment and time per token CDF with MBGMV. Bottom: Same metrics for BGMV.

- MOSTIDLE scheduler selects the inference server that has the least workload.
- FIRSTFIT scheduler picks a server following the first-fit bin-packing strategy, which is also adopted by Punica [1].
- RANDOM scheduler randomly picks an inference server.

Setup. Following [33], we run experiments in two settings: a large-scale simulation and an 8-instance real-world testbed.

Large-scale simulation. We first evaluate the scheduler's performance through simulation, where we obtain the *prefill* and *decoding* latency of the simulator by profiling. We include all 40,000 functions from the MAF trace [21], with aggregated *RPS* $\approx$ 340, and use 60 simulated servers. We set the SLO regarding time per token, as it corresponds to the perceived "speed" of the inference service. The SLO is set to 1.5$\times$ higher than that achieved by the HF-PEFT solution (§1). Fig. 19-Top shows that with S-LoRA's MBGMV, CARASERVE's scheduler achieves an SLO attainment of 99% and speeds up the average time per token by 16.1/18.8/36.4% compared to the MOSTIDLE/RANDOM/FIRSTFIT. Fig. 19-Bottom shows the performance with Punica's BGMV kernel, which is also adopted in CARASERVE (§4.1). Our scheduler has 99% SLO attainment and accelerates time per token up to 36.0%.

Testbed. Next, we evaluate the scheduler in a small-scale testbed, which has 8$\times$A10 GPUs to support 8 Llama2-7B models. Due to the limited number of available CPUs, we use CACHED (§7.1) as the LoRA serving backend, as our CPU-assisted design can rival its performance in various settings (§7.2, §7.3). We randomly sample 1,200 requests with an aggregated *RPS* $\approx$ 60 from the MAF trace [21]. The SLO is also set regarding time per token, which is 1.5$\times$ higher than that achieved by the HF-PEFT solution. As illustrated in Fig. 20, CARASERVE outperforms other baselines by achieving the highest SLO attainment of 80%.

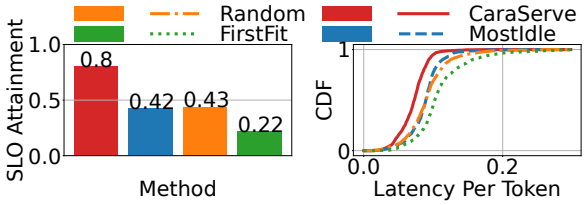

Figure 20: [**Testbed**] Scheduler performance on 8 instances (BGMV). Left: SLO attainment; Right: Time per token CDF.

8 Related Work and Discussion

LLM inference. Optimizing LLM inference is the target of recent studies. Orca [31] proposed iteration-level continuous batching to improve the throughput of LLM serving. Further, vLLM [11] addressed the issue of the GPU memory fragmentation resulting from LLM's KV Cache, improving serving throughput by high GPU efficiency. FlexGen [24] supported LLMs with limited GPU memory, maximizing serving throughput by efficiently storing, accessing, and quantizing tensors. SpotServe [15] leveraged preemptible GPU instances on clouds to reduce the serving cost. CARASERVE is compatible with these optimizations, has already supported continuous batching, and has employed optimized GPU memory management mechanism [16] to mitigate fragmentation.

Multi-tenant LoRA serving. Multi-tenant LoRA serving has recently gained attention in the research community. Punica [1] and S-LoRA [23] are pioneering works targeting multi-tenant LoRA serving. They have designed optimized CUDA kernels for GPU LoRA computation and leveraged existing GPU memory management mechanisms [11, 16] to minimize memory fragmentation. These designs are portable to CARASERVE. However, they overlooked the challenges of *cold-start* and *heterogeneity-aware scheduler* (§2). Besides, PetS [35] proposed a unified framework to serve adapters of different types with LLMs. However, it only considered the discriminative language models, which lack an iterative decoding process and continuous batching.

Multi-model inference serving. A series of works have developed systems for multi-model inference serving, including Clipper [3], MArk [32], Nexus [22], INFaaS [20], Clockwork [8], Shepherd [33], and AlpaServe [12]. These works optimize batching, caching, model placement, and cost-efficiency in serving multiple models in a cluster. However, they are not specially designed to serve generative LLMs and heterogeneous LoRA adapters, leading to optimization gaps.

Discussion. CARASERVE's design is not limited to a particular framework and supports various LLM types. Nevertheless, the scalability of CPU-assisted LoRA serving is limited by the number of available CPUs in the host. Typically, GPU servers designed for LLM serving have abundant CPU cores and host memory. For example, the g5.48xlarge instance provided by

AWS has 192 vCPU cores. Such server configurations are also widely used in production clusters [30], where many GPU instances have one A10 GPU and 128 vCPU cores. In future work, we plan to leverage resource disaggregation to address the scalability issue.

9 Conclusion

This paper presents CARASERVE, a multi-tenant LoRA serving system that is GPU-efficient, cold-start-free, and SLO-aware. In a nutshell, CARASERVE exploits base model multiplexing to serve many LoRA adapters in a batch, coordinates LoRA computation on CPU and GPU to avoid cold-start, and employs a rank-aware scheduler to meet SLOs. CARASERVE is framework-agnostic and can be easily extended to various LLMs. Compared to existing systems, CARASERVE significantly improves serving efficiency by reducing the request serving latency by up to 50% and achieves an SLO attainment of 99%.

References

[1] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving, 2023.

[2] Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. Longlora: Efficient fine-tuning of long-context large language models. *arXiv:2309.12307*, 2023.

[3] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[4] Databricks. Llm inference performance engineering: Best practices. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices, 2023.

[5] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In *Advances in Neural Information Processing Systems*, 2023.

[6] Nvidia Developer. Nvidia nsight compute. https://developer.nvidia.com/nsight-compute, 2024.

[7] PyTorch Docs. Cpu threading and torchscript inference. https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html#runtime-api, 2023.

[8] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[9] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[10] Huggingface. Text generation inference. https://github.com/huggingface/text-generation-inference.

[11] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[12] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[13] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[14] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022.

[15] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *ASPLOS*, 2024.

[16] ModelTC. Light llm. https://github.com/ModelTC/lightllm.

[17] OpenAI. Custom instructions for chatgpt. https://openai.com/blog/custom-instructions-for-chatgpt, 2023.

[18] OpenAI. Gpt-3.5 turbo fine-tuning and api updates. https://openai.com/blog/gpt-3-5-turbo-fine-tuning-and-api-updates, 2023.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[20] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[21] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[22] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[23] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-lora: Serving thousands of concurrent lora adapters, 2023.

[24] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[25] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[26] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca., 2023.

[27] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 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*, pages 10–19, 2019.

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

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 2017.

[30] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association.

[31] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[32] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019.

[33] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association.

[34] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[35] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. PetS: A unified framework for Parameter-Efficient transformers serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.