Plug-and-Play Multi-LoRA Support Layer for vLLM

ABSTRACT

Low-Rank Adaptation (LoRA) is a parameter-efficient fine-tuning technique that is widely used to adapt a base LLM to downstream tasks. In production, companies may manage a large number of LoRA adapters for diverse applications. However, efficient multi-adapter serving often requires inference-engine modifications, creating a high adoption barrier. This project proposes a plug-and-play multi-LoRA support layer for vLLM that requires no vLLM code changes by shifting most engineering effort from runtime to build time. We introduce an offline preprocessing layer that merges the most frequently used (“hot”) LoRA adapters into the base model, while serving less common (“cold”) adapters using preprocessed delta weights. In an online evaluation using vLLM’s serving mode, our approach reduces end-to-end latency by 29.87% compared with vLLM’s default multi-LoRA setup.

1 Introduction

Modern LLM deployments often rely on Low-Rank Adaptation (LoRA) [1] to enable parameter-efficient fine-tuning for a wide range of downstream applications, avoiding full model fine-tuning. In practice, organizations may need to serve a large number of LoRA adapters. However, GPU memory is limited and cannot accommodate all adapters simultaneously, so systems often load adapters on demand. This can introduce substantial overhead from adapter swapping, including GPU memory transfers during inference.

Prior work has explored kernel- and system-level optimizations for multi-LoRA serving. PUNICA [2] proposes a specialized CUDA kernel, Segmented Gather Matrix-Vector Multiplication (SGMV), which enables batching across requests using different LoRA adapters, reporting up to 12× higher serving throughput than prior LLM serving systems while adding only a small per-token latency overhead. S-LoRA [3] introduces LoRA-aware fused kernels to support heterogeneous batching and a unified paging mechanism to reduce memory fragmentation, achieving up to 4× throughput improvement and enabling serving of orders-of-magnitude more adapters.

In this project, we propose a plug-and-play multi-LoRA support layer built upon vLLM, requiring only Python-side adjustments prior to inference. This approach targets a pervasive real-world scenario characterized by uneven popularity, where a single dominant "Hot" LoRA accounts for 80-90% of requests. Our key contributions are summarized as follows:

- *Weight Transformation Strategy*: We propose merging the Hot LoRA weights into the base model and mathematically constructing equivalent "delta adapters" for less popular (Cold) LoRAs. This allows us to reuse the existing vLLM architecture by simply feeding these pre-processed weights.

- *Optimized Inference Logic*: We eliminate the extra LoRA calculation overhead for the majority (Hot) traffic. For Cold LoRAs, we utilize the delta adapters to process requests directly, avoiding the computational cost of explicit runtime subtraction.

We evaluate this method by serving Llama-7B on vLLM. Results demonstrate that our pre-merged approach reduces end-to-end latency by 29.87% compared to vLLM's default multi-LoRA setup.

2 Methodology

2.1 Weight Transformation Strategy

For the "Hot" LoRA, we merge its weights into the base model to form a fused weight matrix W_{fused}. This approach **e**liminates the extra calculation of LoRA adapters and their addition to the base model for the majority of requests, as shown in Equation (1):

$$h = Wx + B(Ax) = (W + BA)x = W_{fused}x \tag{1}$$

For "Cold" LoRAs, we introduce a delta adapter mechanism. Instead of explicitly adding the cold adapter and subtracting the hot adapter at runtime, we compute mathematically equivalent delta adapters, as derived in Equation (2). The specific components are defined in Equations (3), (4), and (5).

$$\begin{aligned} h &= W_{fused}x + B_{cold}(A_{cold}x) - B_{hot}(A_{hot}x) \\ &= W_{fused}x + (B_{cold}A_{cold} - B_{hot}A_{hot})x \end{aligned} \tag{2}$$

$$\Delta B = \begin{bmatrix} B_{cold} & -B_{hot} \end{bmatrix}, \quad \Delta A = \begin{bmatrix} A_{cold} \\ A_{hot} \end{bmatrix}, \quad r_{\Delta} = r_{cold} + r_{hot} \tag{3-5}$$

This transformation guarantees compatibility across adapters with different ranks. Additionally, the rank of the delta adapter becomes the sum of the hot and cold ranks ($r_{delta} > r_{cold}$), which may result in a slight increase in inference latency.

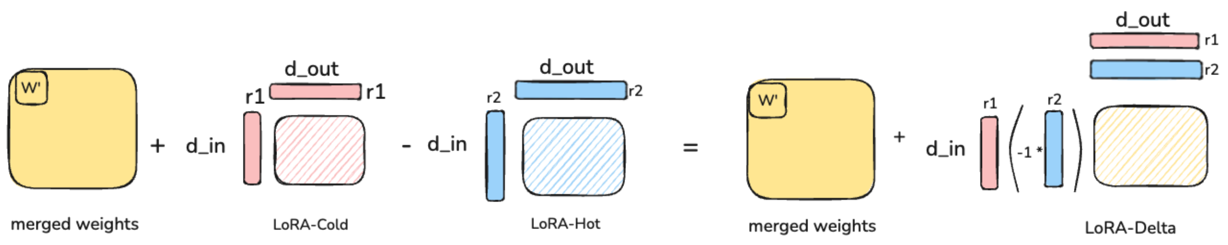

Figure 1. Transformation for Cold LoRA Adapters on a Pre-Merged Base Model

One implementation detail is that, even for the same base model, different LoRA adapters may modify different subsets of layers/weight matrices. Consequently, specific layers may require asymmetric handling: if a layer is modified by the Hot LoRA but not the Cold LoRA, we strictly subtract the Hot

LoRA to neutralize the fused weights. Conversely, if a layer is targeted only by the Cold LoRA, we simply apply its weights without subtraction.

2.2 System Workflow

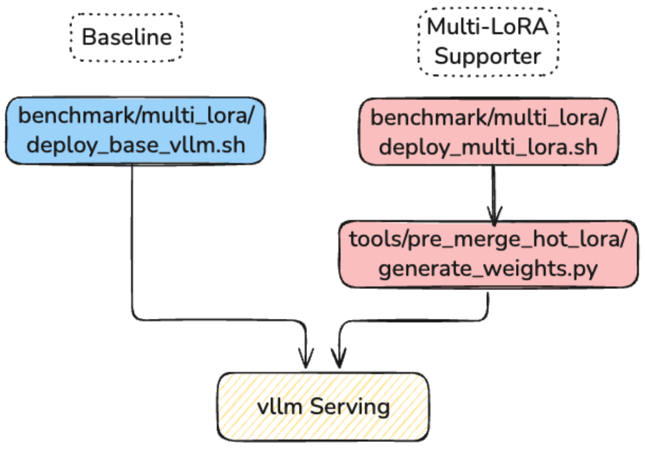
Baseline
Multi-LoRA Supporter
benchmark/multi_lora/deploy_base_vllm.sh
benchmark/multi_lora/deploy_multi_lora.sh
tools/pre_merge_hot_lora/generate_weights.py
vllm Serving

Figure 2. Workflows of the vLLM baseline and our multi-LoRA support layer

The workflow comparison between vLLM's default setup and our multi-LoRA support layer is shown in Figure 2. When multi-LoRA deployment is enabled, the pipeline first preprocesses LoRA weights: it merges the hot LoRA into the base model, computes the corresponding delta weights for the remaining adapters, and saves the fused base weights and delta adapters in the standard LoRA format. The system then starts serving using the existing vLLM deployment workflow.

In our initial design, we considered integrating this preprocessing step directly into vLLM by introducing new serving-time parameters. However, this would require changes to model download and weight storage logic (e.g., how fused base weights and adapters are cached), increasing integration and maintenance complexity. We therefore keep the preprocessing layer external to vLLM, which avoids modifying vLLM code and makes the workflow easier to manage and deploy.

3 Experiments

3.1 Experiment Setup

Hardware Environment: All experiments were performed on a server equipped with an NVIDIA A100 GPU.

Workload Configuration: We extended the standard vLLM benchmarking suite to support multi-LoRA popularity distributions. The workload characteristics are detailed as follows:

- **Dataset**: A synthetic dataset generated by benchmark_lora_popularity.py, comprising 200 requests.
- **LoRA Adapters**: Alpaca-LoRA as the hot adapter, and Guanaco-LoRA (cold adapter 1) and WizardLM-LoRA (cold adapter 2) as cold adapters.

- **Sequence Length**: Each request consists of a 400-token prompt and generates 128 output tokens.
- **Traffic Pattern**: Requests were sent to the serving endpoint at a rate of 10 requests per second (RPS) to mimic online inference traffic.
- **Evaluation Variables**: We varied the proportion of requests targeting the "Hot" (merged) LoRA, denoted as the Hot Ratio, using the set $\{0.0, 0.2, 0.4, 0.5, 0.6, 0.8, 1.0\}$.

3.2 Overall Performance

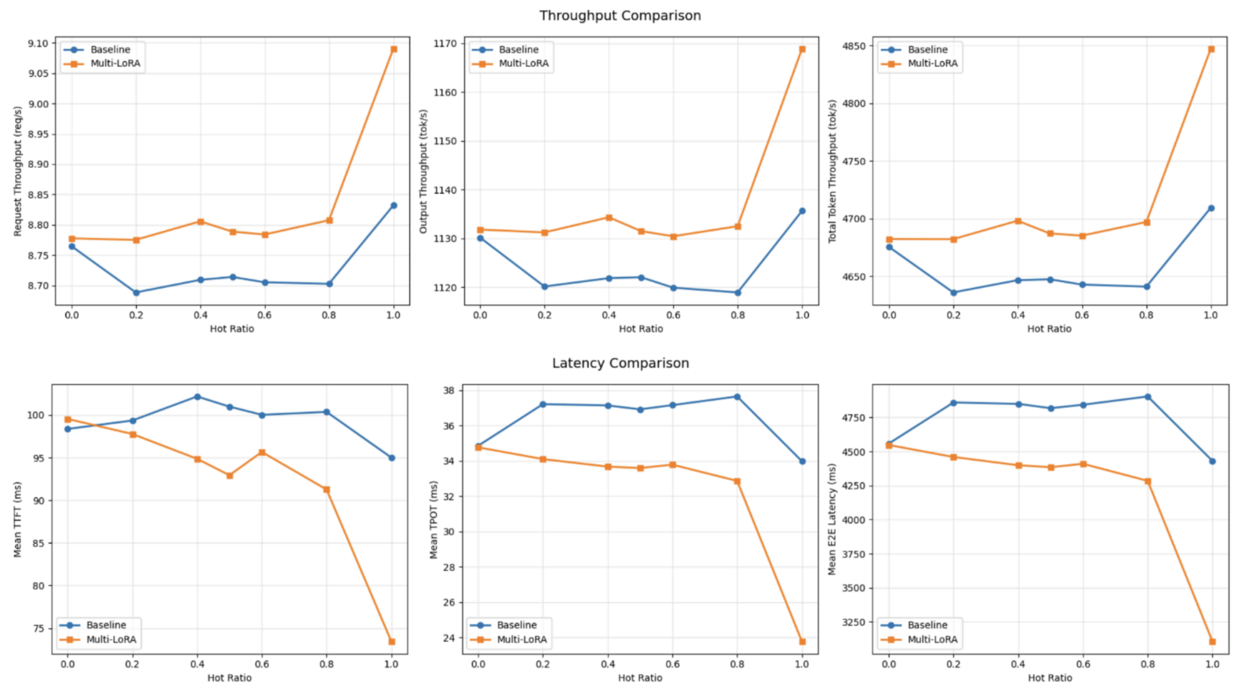

Figure 3. Overall benchmark results: Throughput and Latency under different Hot LoRA request distributions

The results in Figure 3 show that Multi-LoRA delivers only a minor throughput gain of 2.92%. This performance similarity is attributed to two factors. First, the fixed arrival rate of 10 req/s acts as a system bottleneck, preventing the GPU from reaching saturation. Second, given the A100's substantial memory capacity, the system can simultaneously accommodate all active adapters in VRAM. This eliminates any weight swapping overhead for the baseline, allowing it to operate at peak efficiency without memory bandwidth constraints.

In contrast, latency metrics demonstrate a strong correlation with the Hot Ratio, where higher ratios lead to significant reductions in Time to First Token (TTFT) and End-to-End latency, achieving peak improvements of approximately 30% at a ratio of 1.0. This performance boost stems from the fused weights effectively eliminating adapter calculation overhead for "hot" requests. However, at a Hot Ratio of 0.0, the Multi-LoRA approach exhibits slightly higher TTFT compared to the baseline. This confirms the expected trade-off where serving purely "cold" requests necessitates on-the-fly delta computations with an aggregated rank ($r_{delta} = r_{hot} + r_{cold}$), introducing a minor computational penalty.

3.3 Per-LoRA Level Performance

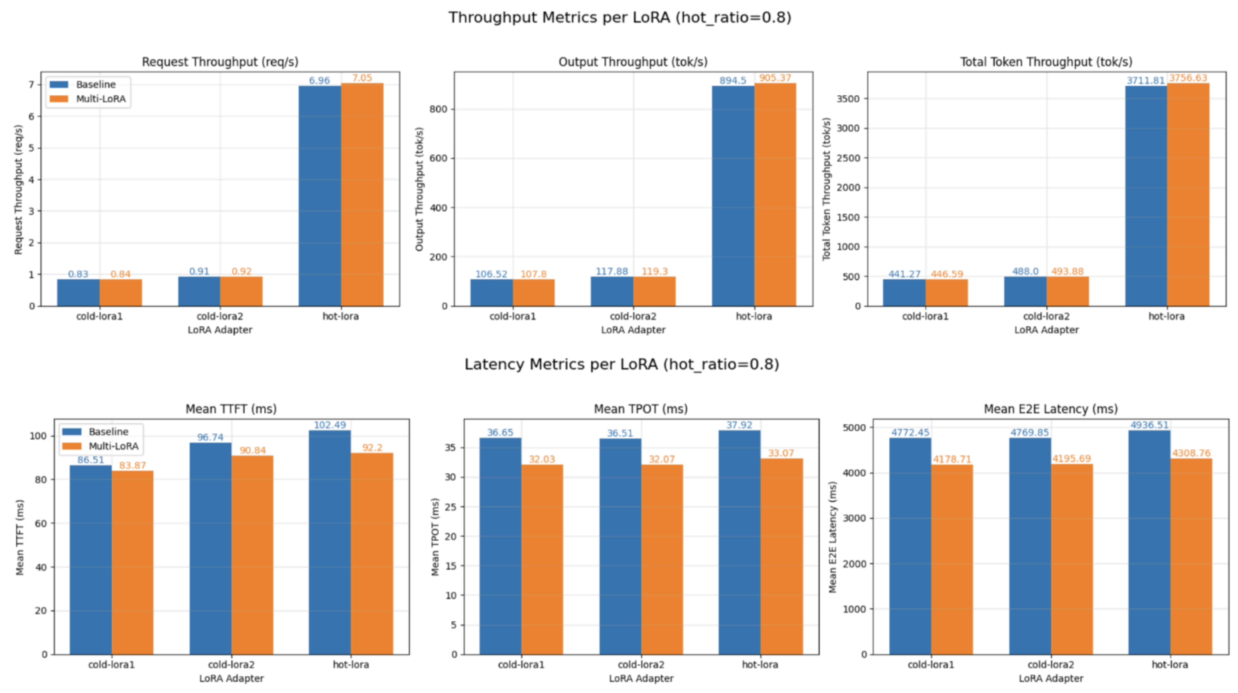

Figure 4. Per-LoRA benchmark results (hot ratio = 0.8): Throughput and Latency across different LoRA request types.

The performance breakdown for individual LoRA adapters is shown in Figure 4. We observe a notable improvement not only for the Hot LoRA but, counter-intuitively, for the two Cold LoRAs as well. This phenomenon is driven by the system-level offloading effect: merging weights eliminates the adapter calculation overhead for 80% of the requests (the Hot LoRA). This significantly alleviates system congestion and reduces the queuing latency for all requests. Consequently, the time saved in the queue far outweighs the minor computational penalty introduced by the expanded rank of the delta adapters resulting in net improvements in both throughput and latency for the Cold LoRAs.

Reference

[1] Hu, Edward J., et al. "Lora: Low-rank adaptation of large language models." *ICLR* 1.2 (2022): 3.
[2] Chen, Lequn, et al. "Punica: Multi-tenant lora serving." *Proceedings of Machine Learning and Systems* 6 (2024): 1-13.
[3] Sheng, Ying, et al. "Slora: Scalable serving of thousands of lora adapters." *Proceedings of Machine Learning and Systems* 6 (2024): 296-311.