ELORA: Efficient LoRA and KV Cache Management for Multi-LoRA LLM Serving

Jiuchen Shi[1,2], Hang Zhang[1], Yixiao Wang[1], Quan Chen[1], Yizhou Shan[3], Kaihua Fu[4], Wei Wang[4], Minyi Guo[1]
[1]*Shanghai Jiao Tong University*, [2]*The Hong Kong Polytechnic University*,
[3]*Huawei Cloud*, [4]*Hong Kong University of Science and Technology*
shijiuchen@sjtu.edu.cn, morninglory@sjtu.edu.cn, yixiaowang@sjtu.edu.cn, chen-quan@cs.sjtu.edu.cn,
shanyizhou@huawei.com, fukaihua@ust.hk, weiwa@cse.ust.hk, guo-my@cs.sjtu.edu.cn

2026 IEEE International Symposium on High Performance Computer Architecture (HPCA) | 979-8-3315-9302-5/26/$31.00 ©2026 IEEE | DOI: 10.1109/HPCA68181.2026.11408492

***Abstract*—Multiple Low-Rank Adapters (Multi-LoRA) are gaining popularity for task-specific Large Language Model (LLM) applications. For Multi-LoRA serving, caching hot LoRAs and KV caches in the GPU memory can improve inference performance. However, existing Multi-LoRA inference systems fail to optimize serving performance like Time-To-First-Token (TTFT), neglecting usage dependencies when caching LoRAs and KV caches. We therefore propose ELORA, a Multi-LoRA caching system to optimize the serving performance. ELORA comprises a *dependency-aware cache manager* and a *performance-driven cache swapper*. The cache manager maintains the usage dependencies between LoRAs and KV caches during inference with a unified caching pool. The cache swapper determines the swap-in or swap-out of LoRAs and KV caches based on a unified cost model, when the GPU memory is idle or busy, respectively. Experimental results show that ELORA reduces the TTFT by 45.7% on average, compared to state-of-the-art works.**

I. Introduction

Large Language Models (LLMs) are now widely used to understand and generate human-like text [6], [11]. While it is cost-inefficient to train LLMs for different tasks, parameter-efficient fine-tuning [20], [50] that freezes the large-scale base model and finetunes multiple Low-Rank Adapters (LoRAs) for various tasks is increasingly popular [13], [3]. For instance, in chatbot [37], [18], [12], multi-language translation [57], [56], and personal agents [4], [31], [49], multiple LoRA adapters (LoRA for short) can be tuned for different user demands and application scenarios. For these LLMs, Key-Value (KV) caches that store input context are often used to maintain coherence and speed up responses during extended interactions by avoiding repetitive computations [25], [1]. Researchers also reused history KV caches for queries with the same prefix [64], [16], [53], [55], boosting performance in iterative tasks.

To improve the serving performance of such Multi-LoRA applications, many works have investigated caching the base model, the KVs [55], [15], [39], [64] or "hot" LoRAs [52], [22], [29] in the GPU memory. vLLM [25] proposed to cache both LoRAs and history KV caches to improve inference performance. Fig. 1 shows an example of caching base model, LoRAs, and KVs. Different LoRAs have separate KV caches (e.g., LoRA-1 and LoRA-2). Moreover, vLLM statically partitioned the GPU memory for LoRAs and KVs, and managed their swap-in or out separately. This is because vLLM allocated different sizes of memory blocks for LoRAs and KVs, preventing their sharing with each other [48].

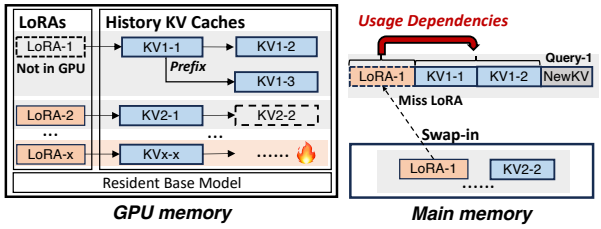


Fig. 1: An example caching state to show the ***Usage Dependencies*** between LoRAs and KV caches.

When a user query is received, the serving system checks whether the required LoRA and KVs are already in the GPU memory. If the required LoRAs and/or KVs are not cached, they are swapped in from the main memory. If the cache space for the LoRAs or KVs is full, some of them are swapped out with various caching policies. When queries use different LoRAs following stable distributions, this solution performs well because the optimal GPU memory space partition can be identified in a "brute-force" way. However, production traces [40], [60], [63] show that the distributions are dynamic. In this scenario, we observe that static GPU memory partition and independent cache management suffer from low efficiencies in intra-LoRA and inter-LoRA aspects.

In the intra-LoRA aspect, a query's KVs may remain cached while its required LoRA is swapped out. As shown in Fig. 1, when "Query-1" relying on LoRA-1 arrives, it can run only if LoRA-1 and its prefixed KV1-1 and KV1-2 are in the GPU memory. However, LoRA-1 is swapped out earlier due to the limited GPU memory space. In this case, the cached KVs are actually "invalid", because the query cannot run without the required LoRA, showing their inherent *usage dependencies*. If the GPU memory space of invalid KVs (e.g., KV1-3) were used to cache LoRA-1, Query-1 could run immediately. Invalid KVs of a LoRA may also prevent useful KVs of other LoRAs from being cached. For instance, KV2-2 is not cached while LoRA-1's KVs are invalid, preventing queries of LoRA-2 from running. Our experiments show that vLLM [25] suffers from 42.4% invalid KV caches on average.

In the inter-LoRA aspect, the required number of LoRAs and the hotness of KVs for different LoRAs change dynamically, due to the varying loads of different LoRAs. For the example in Fig. 1, more LoRAs (e.g., LoRA-x) need to be used

at the next time interval, and the KVs of LoRA-x become hot, but other LoRAs' KVs have occupied the GPU memory, which prevents them from being swapped in. However, with static GPU memory partitioning of LoRAs and KVs, their swap-in or out can only be managed separately, making it hard to uniformly balance their usage in the GPU memory.

To address the above problems, a scheme is required to integrate the usage dependency for each LoRA and its corresponding KV caches with the unified management of GPU memory. We observe that the usage dependencies between a LoRA and its KV caches can be delicately denoted by a tree structure. We therefore introduce a tree-based scheme to maintain the usage dependencies during inference, where nodes are KVs or LoRAs and edges are their dependencies. This can keep more valid KVs in GPU memory to improve efficiency, and thus help in reducing the Time-To-First-Token (TTFT) and Time-Per-Output-Token (TPOT) of queries. Moreover, it is also challenging to balance the GPU memory usage for LoRAs and KVs under varying loads. A cost model is also required to assess the most beneficial LoRAs and KV caches to swap in or out during the Multi-LoRA serving.

We therefore propose **ELORA**, a Multi-LoRA caching system that appropriately manages the swap-in or out of LoRAs and KV caches at the scheduling level. ELORA aims to reduce the TTFT and TPOT, and maximize the supported peak load of Multi-LoRA applications. It comprises a *dependency-aware cache manager* and a *performance-driven cache swapper*. The cache manager maintains the usage dependencies between KV caches and LoRAs based on the tree-based scheme with a unified caching pool, where LoRAs or KV caches are inserted or removed from leaves in the GPU memory to keep the tree connected. Based on usage dependencies, the cache swapper periodically determines the swap-in or out of LoRAs and KVs by using a cost model, which precisely assesses the benefits of swap-in or out LoRAs and KVs to the serving performance of future queries. This paper makes three major contributions.

- **Investigating caching management of LoRAs and KVs for the Multi-LoRA inference.** The analysis motivates us to maintain usage dependencies between LoRAs and KV caches, and to unify the management of their swap-in or out based on their impact on inference performance.
- **The design of a scheme that maintains the usage dependencies between LoRAs and KV caches.** Considering the usage dependencies, more valid KVs are cached in GPU memory, eliminating the intra-LoRA inefficiency.
- **The design of a cost model that guides the swap-in or out of LoRAs and KVs.** The model enables the unified swap-in or out of LoRAs and KVs, eliminating the inefficiency due to the inter-LoRA interference.

We evaluated ELORA with Llama3-8B, Llama2-34B, and Llama3-70B [46], [34] on 8 NVIDIA H800s with chatbot [35], [10], multi-language translation [56], and personal agents [7] scenarios. Compared to the state-of-the-art work, results show that ELORA reduces TTFT and TPOT by 45.7% and 37.8%, respectively, and improves the supported peak load by 78.9%.

TABLE I: Comparisons between ELORA and other works.

	Multi-LoRA serving	Online LoRA (un)loading	History KV reuse	Dynamic manage KVs and LoRAs
TensorRT-LLM [36]	✔		✔	
S-LoRA [42]	✔	✔		
vLLM [48]	✔	✔	✔	
ELORA	✔	✔	✔	✔

II. Related Work

LLM Fine-tuning. Recent studies have proposed efficient methods for fine-tuning LLMs [20], [30], [27], with LoRA adapters being among the most widely used. LoRA achieves fine-tuning with low costs by adding a low-rank branch [20]. Evolved models like DoRA [33] and AdaLoRA [59] that were developed based on LoRA, enhance fine-tuning efficiency by introducing flexible updates through weight decomposition and efficient trimming of insignificant singular values. These models share the same features as LoRA for the LLM inference, thus ELORA can adapt to them with minimal modifications.

KV Cache Management. Some engines like Orca [54] and FastTransformer [41] simply discarded requested KV caches after query processing, which forces recomputations for each new query. To reduce recomputations, FlexGen [43] offloaded the KV caches to host memory and disk for the reuse of subsequent queries. SGLang [64] introduced RadixAttention to handle various KV cache reuse via a global prefix tree and a Least Recently Used (LRU) policy, which is utilized as the underlying operator for ELORA's usage dependency tree implementation. ChunkAttention [64] improved GPU memory utilization by sharing KV caches for common prefixes across different queries. For multi-round dialogues, AttentionStore [15] and Pensieve [55] maintained a multi-level KV caching to store and manage all requested history KV caches to eliminate recomputations. Despite reusing history KV caches, these prior works failed to unify the management of KVs and LoRAs in a manner that accounts for their usage dependencies.

Multi-LoRA Serving Systems. TensorRT-LLM [36] supported Multi-LoRA serving with the reuse of history KVs, but required all LoRAs to be pre-compiled with the base model under its static graph compilation. This cannot support online loading or unloading during the Multi-LoRA serving.

To enable online LoRA loading into the GPU memory, Punica [9] introduced the operator to separate the base model from the task-specific adapter. S-LoRA [42] introduced a unified caching operator for LoRAs and running KV caches, which is utilized as ELORA's underlying operator. It did not reuse history KVs and swapped in LoRAs on demand. Punica and S-LoRA also realized that queries with various LoRAs can be batched to improve inference efficiency. Moreover, dLoRA [52] dynamically merged and unmerged adapters with the base model and migrated queries and adapters across replicas, which is orthogonal to our work. The above works did not consider reusing history KV caches to avoid recomputations.

SGLang [64] integrated the operator of S-LoRA [42] to support Multi-LoRA serving. However, it does not support reusing history KV caches when Multi-LoRA functionality

is enabled due to unresolved compatibility issues in implementations [19]. Moreover, vLLM [48] integrated the operator of Punica for Multi-LoRA serving while reusing history KV caches. It utilized the LRU for LoRAs and KV caches. However, vLLM managed LoRAs and KVs in separate GPU memory spaces, failing to account for their usage dependencies and balance the GPU memory usage under dynamic scenarios.

Table I compares ELORA with representative works.

III. Background and Motivation

In this section, we first introduce the background of Multi-LoRAs, then investigate the inefficiency of current systems.

A. Multi-LoRA Serving with Caching

Multi-LoRA. LoRA [20] is a popular method for efficiently fine-tuning pre-trained LLMs by adding lightweight adapters to original weights. Instead of updating all parameters, LoRA only learns a pair of low-rank matrices that modify the original weights. These matrices are much smaller than the original weight matrix, reducing computational and memory costs.

For the Multi-LoRA scenario, the pre-trained base model is loaded once, and multiple pairs of low-rank matrices are introduced, each corresponding to a specific task [21], [62]. For each task t, a unique pair of low-rank matrices A_t and B_t is learned, and the original weight matrix W is updated as:

$$W'_t = W + \Delta W_t = W + A_t B_t \tag{1}$$

For Multi-LoRA serving, based on the query's task, the corresponding LoRA matrices are loaded into the GPU memory before inferring. Queries using different LoRAs can be processed in a single batch using Segmented Gather Matrix-Vector multiplication (SGMV) [42], [9].

KV Caches for Multi-LoRAs. The LoRAs need to be loaded into the GPU memory during the inference [20], [42]. Moreover, most LLMs use a decoder-only transformer to predict the next token with KV caches computed from previous tokens [14], [11]. When a query matches an existing prefix, the stored history KV caches can be reused to eliminate redundant computations and reduce GPU memory usage.

Each LoRA adds a low-rank branch to the original weights, which affects the computations of the KV cache. For a hidden state h at a specific layer, the Key and Value matrices using LoRA t with original weights W_K and W_V are computed as:

$$K_t = (W_K + A_{t,K} B_{t,K})h, \quad V_t = (W_V + A_{t,V} B_{t,V})h \tag{2}$$

Thus, the KV cache for each LoRA differs due to task-specific modifications in the matrices, which require separate storage of the KV caches for different LoRA adapters [13].

The separate storage increases contention for limited GPU memory space, and thus the KV caches and LoRAs are usually offloaded to main memory and swapped in or out on demand [15], [42]. This can cause cold starts when loading them back into GPU memory. To reduce this overhead, we can pre-cache "hot" KV caches and LoRAs into GPU memory.

Multi-LoRA Serving. For a new query, if the required LoRA is not in the GPU memory while the GPU memory is full, this query needs to queue to wait for other KV caches or LoRAs swapped out from the GPU memory, and then load the required LoRA. Similarly, if the required KV caches are not in GPU memory, they will be swapped in from the main memory. Once the required LoRA and KVs are properly loaded and matched, the inference begins to generate the next token.

LLM inference typically has prefill and decode stages [54], [1], [8], corresponding to two performance metrics, i.e., TTFT and TPOT. The above Multi-LoRA serving workflow can introduce overheads due to the queue waiting for the GPU memory space, LoRA cold-starts, and KV cache cold-starts, affecting both the TTFT and TPOT.

B. Application Scenarios for Investigations

We built three commonly-used Multi-LoRA LLM inference applications based on real-world traces for investigations.

Chatbots. In each dialogue round, chatbots generate responses using full user history. Online services often let users choose specific scenarios (e.g., business analysis [47]), and apply Multi-LoRA inference to improve efficiency. We construct queries using LMSYS-33k dataset [63], which has 33,000 dialogues with model names, texts, and timestamps. Based on model names, we assign the target LoRA of each query and retain the original query distribution for different models. To form different query sending rates, we proportionally scale this dataset while preserving its original pattern [42], [52].

Multi-language Translations. This service uses Multi-LoRAs to dynamically select optimal models to enhance translation results [66]. We construct queries from the OPUS-100 dataset [56], which contains 55 million sentence pairs in 100 languages. We map each language translation pair to a specific LoRA, e.g., French to English. As the OPUS-100 dataset lacks timestamps, we adopt query arrival patterns from the Microsoft Azure function trace (MAFT) [40], [60], following previous works [52], [22]. We rank MAFT functions by invocation frequency, select the top-n query types, and map them to the n LoRAs to maintain query distribution.

Personal Agents. LLMs are widely used in this scenario, e.g., mobile assistants and home assistants, with Multi-LoRA enabling efficient multi-task support [31], [62]. We construct queries using the Google Taskmaster dataset [7], which features multi-turn, task-oriented dialogues that mirror real-world assistant interactions. We apply the same sampling based on MAFT as in the translation.

To adapt various LoRA numbers (n) to the above scenarios, we randomly choose the query patterns from n models, translation pairs, or task scenes in corresponding datasets and map to n LoRAs. Like other works [42], [52], we randomly select LoRAs from the HuggingFace repository of the corresponding LLMs, and this does not affect the serving performance [9]. The ranks of LoRAs in our evaluations are either 32 or 64.

The traces we utilized [63], [40], [56], [7] can capture the dynamics of queries accessing various LoRAs. As statistics, the required GPU memory for LoRAs (with corresponding KV caches) varies by 48.1% on average every 1 second, in which 73.9% variations are beyond 20% and others are below 20%.

TABLE II: Experiment specifications

	Specifications
Hardware	Intel Xeon Platinum 8480CL CPU, 256GB memory NVIDIA H800 (each of 80GB GPU memory) ×8 PCIe 5.0, 128GB/s interconnection bandwidth
Software	Llama3-8B, Llama2-34B, Llama3-70B, LMSYS-33K [63], OPUS-100 [56], Taskmaster [7], Microsoft Azure Function trace [40], [60]

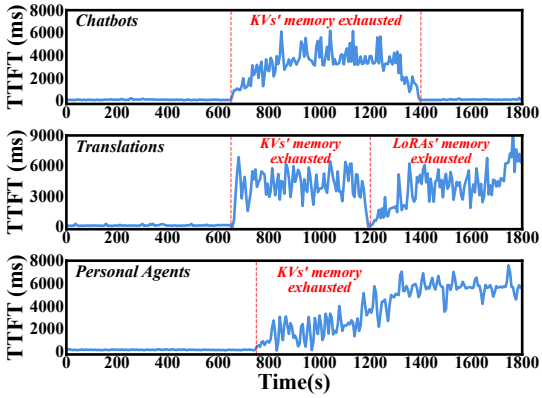


Fig. 2: The TTFT of the vLLM for different scenarios.

C. Low Multi-LoRA Serving Performance

In this subsection, three application scenarios described in Section III-B are used as benchmarks. We use the Llama3-8B, Llama2-34B, and Llama3-70B as base models, and evaluate them on eight NVIDIA H800 GPUs. We construct various LoRA numbers (20, 50, and 100) for each base model. Table II shows the hardware and software configurations.

We have tried to use the latest version of SGLang [61] that supported the Multi-LoRA serving but cannot reuse the history KV caches. However, evaluation results show the average TTFT of SGLang can be as high as 9568.9ms. This extremely low performance is similar to observations from others [19]. It has prevented us from further investigations, as we suspect it may be caused by poor Multi-LoRA compatibility of SGLang.

We therefore choose to use vLLM [48] that caches both LoRAs and history KVs as the representative serving system. vLLM integrates the Multi-LoRA serving kernels of Punica [9], [48], with more optimizations like prefix-caching to reuse history KV caches. It allocates fixed GPU memory space for LoRAs and KVs, and utilizes the LRU strategy to swap in or out LoRAs or KVs in the respective GPU memory area. vLLM sets a predefined allocation ratio of GPU memory space for LoRAs (empirically to be 0.2) and the memory block size to 32, referring to the vLLM latest version [48].

Fig. 2 shows the TTFT over time of vLLM for the three benchmarks with the Llama2-34B base model under the LoRA number of 50. Experiments with other base models show similar observations, as shown in Section VIII. With varying loads, we observe that vLLM experiences significantly high TTFT at certain periods, due to insufficient GPU memory space allocation for KV caches or LoRAs. As statistics, the TTFT of the three benchmarks are 1353.4ms, 2548.9ms, and 2339.6ms on average, respectively. This is because the static GPU memory allocation of vLLM cannot dynamically adapt to the varying loads in Multi-LoRA serving. The GPU memory allocation is static because vLLM allocates memory blocks with different sizes for LoRAs and KVs according to their respective requirements [48]. Memory blocks in the GPU memory area of KV caches cannot be used for LoRAs, making it impossible to dynamically adjust the pool sizes.

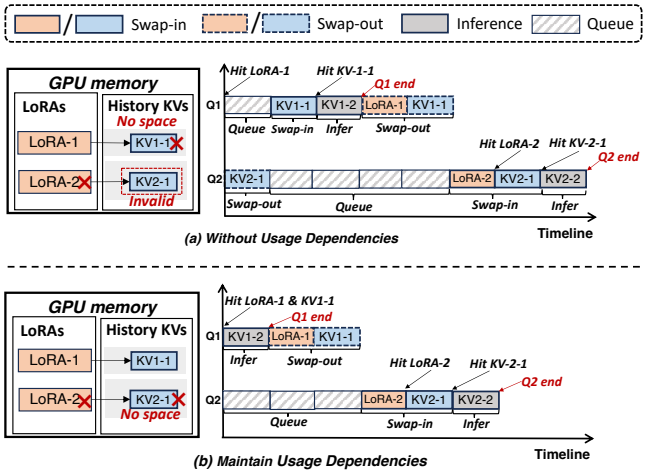


Fig. 3: Examples of serving two queries under: (a) without usage dependencies, and (b) maintaining usage dependencies.

While redeployment can change the GPU memory partition, it results in large overheads that can block normal inference for tens of seconds [5], [2]. Moreover, even if dynamic GPU memory allocation is achieved with fine-grained memory blocks [9], [42], [22], it is still hard to define an appropriate allocation policy with varying loads of LoRAs.

D. Diving into Underlying Reasons

Our investigations show that the poor performance is caused by 1) *inefficient GPU memory usage without considering intra-LoRA usage dependencies*, and 2) *inappropriate swap-in or out of KVs and LoRAs under the inter-LoRA varying loads*.

1) Inefficient GPU memory Usage: Fig. 3 shows an example of serving two queries (*Q1* and *Q2*) of two LoRA adapters (*LoRA-1* and *LoRA-2*) under this kind of method.

As shown in Fig. 3(a), it is possible that KV2-1 is cached while the corresponding LoRA-2 is not in the GPU memory, without considering usage dependencies between LoRA and its KVs. Prior work like vLLM [25] allocated static GPU memory for LoRAs and KV caches, respectively, and managed their swap-in/out separately, which can lead to this situation. This occurs because, when queries access more LoRAs, the GPU memory allocated for LoRAs becomes insufficient, forcing most LoRAs (including those "hot" ones) to be swapped out. By contrast, the GPU memory for KV caches is excessive, thus some evicted LoRAs' KV caches reside in the GPU.

Under this case, KV2-1 is "invalid", since $Q2$ cannot run without the LoRA. At the same time, $Q1$ is also blocked although its required LoRA is cached, because it needs to wait

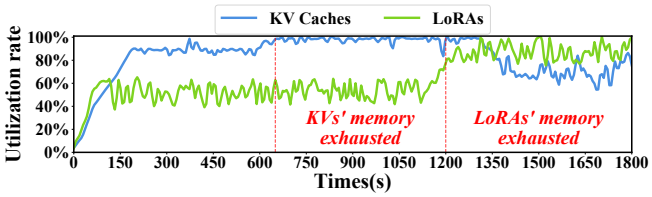


Fig. 4: The utilization rate of the GPU memory space allocated to LoRAs and KV caches over time in the translation scenario.

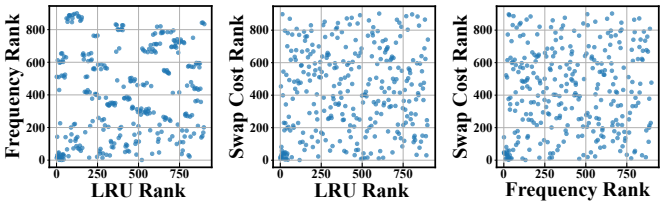


Fig. 5: The relationships among the visited frequency, swap costs, and LRU of the KV caches or LoRAs.

for the required KV1-1 to be swapped in, before which KV2-1 should be swapped out to free some GPU memory space. After $Q1$ returns, $Q2$ needs to swap in LoRA-2 and KV2-1 again to perform the inference. Neglecting the usage dependency, the system causes redundant swap-in/out, greatly increasing queuing overhead. From our evaluations in Section VIII, vLLM results in 42.4% invalid KV caches on average.

Fig. 3(b) shows a better caching case where LoRAs and KVs are managed based on the usage dependency. In this case, $Q1$ runs directly because both LoRA-1 and the KV1-1 are in the GPU memory. After $Q1$ returns, $Q2$ runs after LoRA-1 and KV1-1 are swapped out, and the required LoRA-2 and KV2-1 are swapped in. In this way, the redundant swap-in or out is eliminated, and the response time of both $Q1$ and $Q2$ reduces.

While prior work does not consider the usage dependency between LoRA and its KV caches, the limited GPU memory space is not efficiently used.

2) Inappropriate Swap-in or out of KV caches and LoRAs: Previous works [48], [64] manage LoRAs and KVs in GPU memory separately, and adopt caching strategies like LRU for swap-in or out. They cannot balance GPU memory usage for LoRAs and KVs when loads of different LoRAs change.

Take benchmark *translation* in Fig. 2 as an example. TTFT increases up to 6729.7ms and 8962.9ms during the period of 650s-1200s and 1200s-1800s, respectively. Correspondingly, Fig. 4 shows the GPU memory utilization rates of the LoRA and the KV parts. After looking into detailed serving trace, we find that the long TTFT originates from different reasons. During the 650s-1200s, the GPU memory space for KVs is exhausted while the utilization rate of GPU memory space for LoRAs is 59.2% on average. In this case, the frequent swap-in or out of KVs results in the long TTFT. During the 1200s-1800s, the GPU memory space for LoRAs is exhausted in contrast, because queries of more LoRAs are received during that period. According to the trace, queries of 29 LoRAs are received before 1200s, while that value is 48 after 1200s.

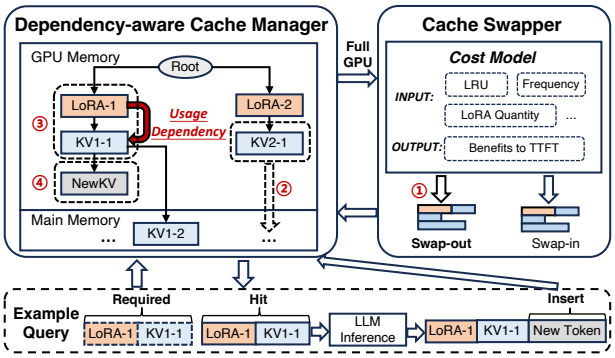


Fig. 6: Design overview of ELORA.

We should dynamically balance the GPU memory usage of LoRAs and KVs. However, even if the GPU memory space of LoRAs and KVs is dynamically managed through fine-grained memory blocks, relying on LRU for the swap-in or out is not efficient. This is because TTFT is related to many factors, like the swap cost and visited frequency. Fig. 5 shows the relationship between the visited frequency, LRU, and swap cost of each KV cache and LoRA. In the figure, each point represents a LoRA or KV cache, and its x or y-axis represents its corresponding ranks of LRU/Frequency/Swap Cost. As observed, the points are randomly distributed, which means that there is no clear correlation among these factors. Therefore, only considering the LRU strategy cannot represent the situations of other important metrics for serving performance.

Relying on the LRU to manage the GPU memory space for LoRAs and KV caches is not efficient to minimize the TTFT, even if dynamic GPU memory usage is enabled.

IV. ELORA Methodology

Fig. 6 shows the design overview of ELORA. It mainly comprises two parts: a *dependency-aware cache manager* and a *performance-driven cache swapper*.

The most challenging part of ELORA is managing LoRAs and KV caches based on the usage dependencies to eliminate invalid KV caches. Thus, ELORA's cache manager introduces a tree-based scheme to address this problem, where nodes represent LoRAs or KV caches, and edges represent the usage dependencies among them. To maintain usage dependencies, this scheme places LoRAs on the second layer, and swaps out leaf nodes in the GPU memory and swaps in root nodes in the main memory during Multi-LoRA serving (Section V).

When the loads of queries using different LoRAs change, the used LoRA number can increase and the hotness of some KV caches of some LoRAs changes. ELORA's cache swapper periodically decides the swap-in or out of different LoRAs and KV caches based on metrics like LRU, visit frequency, and the LoRA quantity. The challenging part here is to build the cost model to directly evaluate the benefits to TTFT for swapping in or out each LoRA or KV cache (Section VI).

Specifically, ELORA works as follows. Firstly, the cache manager constructs the usage dependencies between LoRAs and KV caches into a tree-based structure. Secondly, during

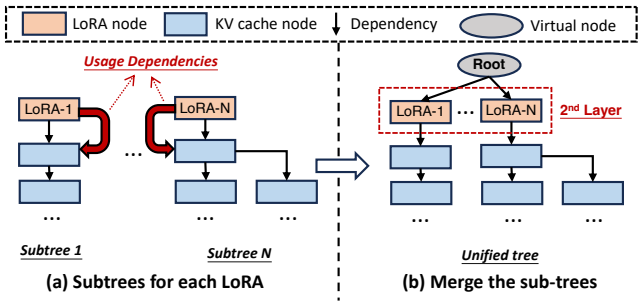


Fig. 7: The construction process of the usage dependencies among LoRAs and KV caches.

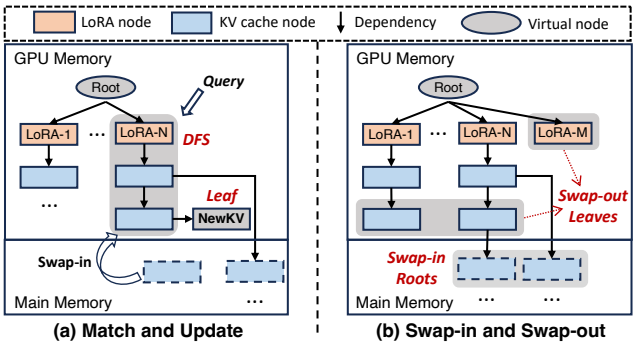


Fig. 8: Maintaining the usage dependencies among LoRAs and KV caches during the query inference.

serving, it inserts newly loaded LoRAs into the second layer of the tree and inserts or deletes KV caches at the leaves of their corresponding LoRA branches. Thirdly, after each monitor interval, the cache swapper retrieves the states of nodes from the cache manager and decides the swapped-in or out KV caches and LoRAs when the GPU memory is idle or busy, respectively. The swap-in or out decisions are sent back to the cache manager to perform memory operations.

We use an example for better explanations as shown in Fig. 6. ① As the GPU memory is full at the start, the cache swapper gets the states of the tree and utilizes the cost model to evaluate the nodes in this tree, and determine the most "cold" one "KV2-1". ② The cache manager swaps out the "KV2-1" from the GPU memory. ③ When a new query arrives, the cache manager fits its required LoRA and KV caches "LoRA-1" and "KV1-1" in the dependency tree. ④ This query then proceeds to generate the next new token, and lastly, the cache manager inserts its KV cache "NewKV" in the leaf of the corresponding LoRA branch in the tree.

V. Dependency-aware Cache Manager

In this section, we first analyze how to construct the usage dependencies among LoRAs and KV caches, then introduce their maintenance during serving the queries.

A. Usage Dependency Constructing

As we analyzed in Section III-D1, a LoRA and its corresponding KV caches have their inherent usage dependencies. When ignoring these dependencies, invalid KV caches will occupy the GPU memory space, leading to low performance for Multi-LoRA serving. We adopt a tree-based scheme to construct the usage dependencies. as shown in Fig. 7.

For each specific query, it will first match the required LoRA and then its corresponding KV caches. The KV caches corresponding to different tokens also have their matching orders. For instance, in the sentence "To be or not to be", the KV cache for the token "To" should be matched in front of "be". Therefore, as shown in Fig. 7(a), the LoRAs and subsequent KVs can be intuitively connected by a chain like the branch of LoRA-1, where nodes represent LoRAs or KV caches and edges represent the usage dependencies. Moreover, a KV cache for a token may have several possible subsequent KV caches. For instance, the subsequent tokens for the prefix sentence "To be" can be "or not to be" or "the best". Thus, the LoRA and its subsequent KV caches can also construct a subtree like the branch of LoRA-N in this figure.

Since these subtrees constructed above are still separate, we need to merge these subtrees into a unified one, as shown in Fig. 7(b). We use a virtual root node to connect the subtrees for different LoRAs to form a unified tree, with all LoRA nodes placed on the second layer of the tree. In this way, newly arrived queries can first match the required LoRA node in this tree, and then match the KV cache nodes in this LoRA branch. Through the construction of usage dependencies described above, the usage dependencies among LoRAs and KV caches within the same LoRA branch are established, while KV caches in different LoRA branches remain independent.

In the dependency tree, ELORA divides each LoRA or KV cache into the same fixed-size memory blocks, and a LoRA or KV cache block is represented by a node in the dependency tree. Thus, the specific LoRA rank or KV cache size does not impact ELORA's caching strategy. The node label for each KV cache node is the hash value of the token sequence, and for each LoRA node is the LoRA ID. For the swapping decisions in Section VI, we also record the related data for each node. Each node retains its corresponding information, i.e., the visit frequency, the last recent usage time, and the node size. These data will be updated when each node is generated, matched, or swapped in or out during the inference.

B. Dependency Maintaining During Inference

To maintain the usage dependencies among LoRAs and KV caches during the query inference, we need to correctly match and update the LoRAs and KV caches in the dependency tree. Moreover, we need to swap in or swap out appropriate nodes in the dependency tree according to the cache swapper's decisions (Section VI) when the GPU memory is idle or busy.

For the matching and updating, as shown in Fig. 8(a), when a query arrives, it needs to match the required LoRAs and KV caches. This query will first match the LoRA node in the second layer. If the LoRA resides in the main memory, this node is swapped in the GPU memory asynchronously. Then, within the subtree of this LoRA branch, this query begins to match history KV caches according to Depth-First-Search (DFS) of the tree until the leaf node is reached or no

corresponding node can be found. During the KV matching process, if the required KV cache resides in the main memory, it will first be swapped into the GPU memory. Through the above prefix matching process, we can maximize the reuse of KV caches that have already been computed according to the usage dependencies. At last, this query generates a new token with a new KV cache, and we will insert it below the last matched node of its corresponding LoRA subtree. Also, during the decoding process, the new KV cache will be continuously inserted into the leaves of this LoRA branch.

When the GPU memory is idle or busy, some LoRAs or KV caches need to be swapped in or swapped out to fully utilize the GPU memory and main memory resources. As shown in Fig. 8(b), the cache manager will control the swapping-out to start from the leaf nodes in the GPU memory, as well as control the swapping-in to start from the root nodes of each subtree in the main memory. This is because, during the node matching process in the dependency tree, the nodes higher up will always be prioritized for matching, and all their children nodes depend on them. In this way, the usage dependencies among LoRAs and KV caches can be maintained during the inference, and all KV caches in the GPU memory are valid ones, thus the GPU memory space can be fully utilized.

It is worth noting that ELORA's usage dependency tree is different from the RadixAttention tree of SGLang [64], which only handles various KV reuse patterns without considering LoRAs. Moreover, based on the design of the usage dependency tree, ELORA further introduces a cost model tailored for comprehensively deciding the swap-in/out of both LoRAs and KV caches in Section VI.

VI. Performance-driven Cache Swapper

In this section, we first analyze how the quantity of LoRAs in the GPU memory affects TTFT. Then, we introduce a cost model to assess benefits to TTFT of swapping in or out LoRAs and KVs. Lastly, we introduce the workflow of cache swapper.

A. Considering the LoRA Quantity on TTFT

As the LoRA quantity used changes dynamically over time, the LoRA quantity in the GPU memory can impact the TTFT.

Fig. 9 shows the TTFT under the chatbot scenario with different static GPU memory allocation ratios for LoRAs in the vLLM. In this experiment, the used LoRA number in the traces is set at 20 and 50, as well as the average sending rate is 5 queries per second. We can observe that the TTFT reduces significantly before reaching a target ratio, and the target ratio increases when the required LoRA number changes from 20 to 50. This is because the query inference can only start once the required LoRA is matched in GPU memory, otherwise, the query is queued. An insufficient LoRA loading quantity in GPU memory can cause a large amount of LoRA cold-starts, leading to a significant increase in TTFT. Therefore, sufficient LoRA quantity is needed under different dynamic scenarios.

In realistic execution scenarios, ELORA does not need to utilize Fig. 9 to determine the '"target ratio", but determines

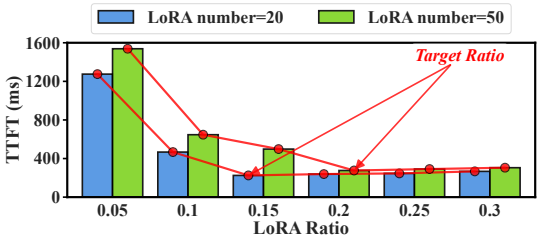


Fig. 9: The TTFT of vLLM in the chatbot scenario under different GPU memory allocation ratios for LoRAs.

the required loaded LoRA quantity using the following estimation methods. The current required LoRA quantity is estimated based on two factors: the usage frequency probability $prob_i$ of LoRA i, which is obtained from the recorded data in the dependency tree, and the recent inference batch size BS from the last 5 seconds. Using these data, we calculate the expected number of LoRAs required for inference (Low_{lora}) as follows:

$$Low_{lora} = \sum_{i=1}^{n} fe_i = \sum_{i=1}^{n} \left(1 - (1 - prob_i)^{BS}\right) \quad (3)$$

In this equation, fe_i represents the probability that the LoRA i is present in the recent batch, i.e., 1 minus the probability that no queries in this batch use this LoRA. With the Low_{lora}, our cost model will encourage the loaded number of LoRAs in GPU to approach it in Section VI-B.

B. Cost Model to Access Benefits to TTFT

When performing swap-in or out for LoRAs and KV caches, the goal is to retain the most valuable KVs and LoRAs in GPU memory as much as possible, thereby optimizing the TTFT for incoming queries. To achieve this goal, our key idea is to design a cost model to evaluate the expected benefits to TTFT of retaining a KV cache or LoRA in GPU memory. Our cost model is carefully built with following two parts.

Firstly, we should address the issues of high TTFT caused by pre-caching insufficient LoRAs, as analyzed in Section VI-A. The cost model needs to try to load a sufficient quantity of LoRAs. Thus, we first define $LoRA\_Eval_i$ as the reward coefficient that encourages the loaded quantity of LoRAs to be close to Low_{lora} (Eq. 3) as:

$$LoRA\_Eval_i = min(1, \frac{Now_{LoRA_i}}{Low_{lora}}) \quad (4)$$

In this formula, Now_{LoRA_i} represents the number of LoRAs after the swap operation of node i. This formula encourages the number of LoRAs loaded in GPU to approach the expected LoRA number (Low_{lora}). In our evaluations, for 94.8% of the time, ELORA can ensure the loaded LoRA number is within +-5% error relative to the Low_{lora}.

Secondly, we should estimate the expected cold-start latency reduction to TTFT for future queries when retaining each LoRA or KV cache in the GPU memory. As we analyzed in Section III-D2, we consider the performance metrics that include the visited frequency, the LRU, and the cost of swap-in or out of nodes. Therefore, we define $Retain\_Eval_i$ as:

$$Retain\_Eval_i = cost_i \times visit_i \times (1 - \text{sigmoid}(t_i)) \quad (5)$$

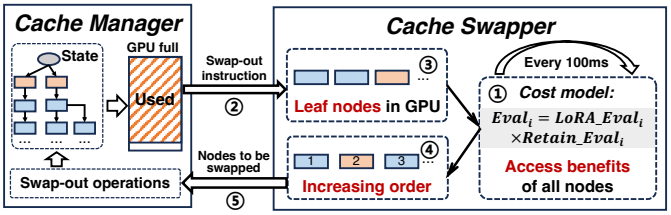


Fig. 10: The execution workflow of the cache swapper with the swap-out instruction as examples.

In this formula, the first item transfer $cost_i$ is computed using the PCIe bandwidth and size of the KV or LoRA, and the second item visit frequency probability $visit_i$ is obtained based on the recorded data on the dependency tree. The third item is a time decay function similar to the forget-gates in the LSTM, whose t_i represents the time difference between the current time and the last recent usage time. The inclusion of visit frequency follows prior KV cache management studies [39], [58]. The sigmoid-based item considers the LRU, which ensures that less recently used KVs or LoRAs get higher eviction priority, as commonly used in prior work [45], [28].

Combining the formulas of the LoRA reward coefficient and the expected TTFT benefits of future queries, we finally design the cost model to access a KV cache or LoRA i as:

$$Eval_i = LoRA\_Eval_i \times Retain\_Eval_i \tag{6}$$

As for the definition, a KV cache or LoRA with a higher $Eval_i$ has more benefits to be stored in GPU memory, also meaning it incurs higher costs if it is swapped in GPU memory from the main memory. This cost model evaluates the relative relationship between each KV cache or each LoRA in terms of benefits to the TTFT when retaining them in the GPU memory. Then, we can use these relationships to decide their swap-in or out orders when the GPU memory is full or idle, respectively.

The cost model can handle varying KV block sizes across LoRA branches. This is because it conducts swap-in/out for each node in the usage dependency tree that represents a fixed-size memory block, which naturally accounts for the storage differences across branches (i.e., different numbers of nodes).

C. Workflow of the Cache Swapper

Fig. 10 shows the operation workflow of the cache swapper.

After each 100ms interval, the cache swapper first updates the accessing of benefits ($Eval_i$) of all nodes in the tree based on the cost model in Eq. 6 (①). At the same time, the cache manager calculates the GPU memory usage based on the storage state of the usage dependency tree. If the GPU memory is full, the cache manager will send the swap-out instruction to the cache swapper (②). According to Section V-B, the leaf nodes in the GPU memory will be sent as candidate nodes to the cache swapper (③). With the candidate nodes, the cache swapper sorts them in increasing order of $Eval_i$ for the swap-out (④). The cache manager continuously swaps out the nodes one by one according to the sorting result (⑤). Moreover, since the number of LoRAs in the GPU after each LoRA node swapping could change, ELORA updates the evaluation function after each LoRA swapping.

Similarly, if GPU memory is idle (e.g., below 70% utilized), the cache manager sends the swap-in instruction to the cache swapper. The root nodes of each path in the main memory will be the candidate nodes. The decision process is the same, and just change the sorting orders of nodes to descending.

ELORA naturally supports both small and large GPU memory changes, since it updates $Eval_i$ (Eq. 6) for each LoRA or KV and decides their swap-in or outs at every fine-grained 100ms. If a few LoRAs/KVs are required for inference with small changes, and GPU memory has space, ELORA is capable of directly loading them. Above methods are supported by the minimal $Eval_i$ updating overhead of up to 3.1us, as well as ELORA's asynchronous swap-in/out implementations with the swapping overhead only up to 0.47ms in our evaluations.

VII. Implementation of ELORA

ELORA can be adapted to popular LLM engines [64], [54], [1] by replacing their memory management module with few modifications. It applies to LLMs based on decoder-only transformer [46], [14], [11] that cover most practical scenarios. ELORA is implemented based on vLLM [48] with an extra 7856 and 1766 lines of Python and C++ codes. It uses Tensor Parallelism [42], [48] for distributed inference of LLMs. For serving queries that use LoRAs with various ranks, ELORA employs the SGMV operator [42], [9] to enable their batching.

Unified Caching Pool for LoRAs and KVs: We extend the BlockManager of vLLM [25], [64] to achieve this. During the initialization, both GPU and main memory are partitioned into memory blocks of the same size, which is similar to S-LoRA [42], but we also extend this pool to store history KV caches. We also perform block-wise partitioning of LoRAs along the rank dimension, while other dimensions of LoRAs align with those of the KV caches.

Usage Dependency Tree: Built on top of the unified memory, it is a data structure that merely logically records the memory addresses of different LoRA and KV cache memory blocks, while the actual data resides in their respective physical locations across GPU and main memory. We utilize an efficient trie tree [51] that is similar to SGLang [64] to implement this tree, whose node matching and updating are less than 1ms. Moreover, since operations (like search, insert, and delete) for this tree are executed by the CPU, it is stored in the main memory with a maximum 676.5KB memory usage. When the GPU and host memory are both exhausted, cold KV blocks are evicted, and their entries on the dependency tree are deleted.

Asynchronous Swapping: To further mitigate the cold start overhead, we adopt the asynchronous swap-in or out similar to other work [15], by using the Stream library in Torch [38]. After a query arrives, if its required LoRA or KV caches are not in GPU memory, we swap in the corresponding memory blocks and just let this query wait without blocking other queries' inference. This realizes the overlap of inference and data transferring with no extra swapping overhead.

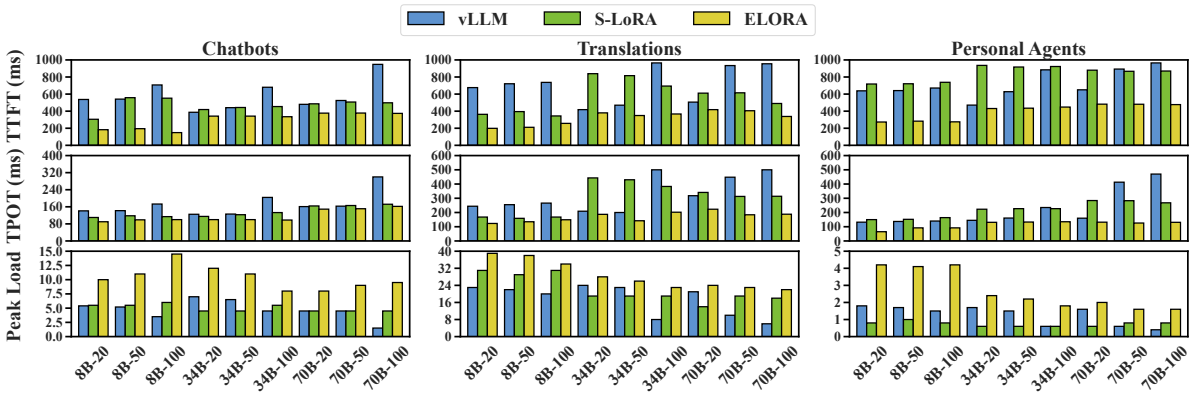


Fig. 11: The average TTFT, TPOT, and supported peak load of ELORA, vLLM, and S-LoRA in various scenarios. The x-axis represents the model size and LoRA number, e.g., 8B-20 represents Llama3-8B model with the LoRA number of 20.

VIII. Evaluation of ELORA

In this section, we first show the performance of ELORA under various Multi-LoRA applications. Then, we investigate the effectiveness of each module and scalability of ELORA.

A. Evaluation Setup

Table II has shown our experimental platform. The chatbot, translation, and personal agent are described in Section III-B. We use Llama3-8B, Llama2-34B, and Llama3-70B (8B, 34B, and 70B for short) as base models. Based on the parameter size, we use 1, 4, and 8 NVIDIA H800 GPUs to deploy them, respectively. Each H800 has 80GB GPU memory, which is the same as NVIDIA A100 or H100. Moreover, we use various LoRA numbers (20, 50, and 100) for each base model.

We use the state-of-the-art Multi-LoRA serving systems vLLM [25] and S-LoRA [42] as baselines. vLLM integrates the Multi-LoRA serving kernels of Punica [9], [48], with more optimizations like prefix-caching to reuse history KV caches. It partitions GPU memory and allocates static GPU memory space for LoRAs and KV caches, and uses the LRU to swap out the KV caches or LoRAs when GPU memory is full. Refer to vLLM's latest version [48], we set the GPU memory allocation ratio for LoRAs to 0.2. Moreover, S-LoRA utilizes a unified caching pool for LoRAs and KV caches. It does not reuse history KV caches and swaps in LoRAs on demand.

We do not select TensorRT-LLM [36] and SGLang [61] as baselines due to the following reasons. First, TensorRT-LLM requires all LoRAs to be pre-compiled with the base model, preventing dynamic loading at runtime under Multi-LoRA serving. Second, although SGLang integrates the kernel of S-LoRA [42], it cannot reuse history KV caches when the Multi-LoRA functionality is enabled. As we discussed in Section III-C, the TTFT of SGLang is as high as 9568.9ms on average. These may be caused by implementation issues [19].

All the experiments are conducted under continuous batching, which is the most popular LLM batching strategy [54]. ELORA focuses on optimizing the caching replacement, which is compatible with different batching strategies.

Following prior works [65], [52], we utilize the TTFT, TPOT, and supported peak load as performance metrics. The TTFT and TPOT data in this paper are the average values for queries. The supported peak load is set as the supported maximum queries per second when the TTFT is below 500ms.

B. Latency and Supported Peak Load

We first evaluate ELORA on the inference latency and the supported peak load. For each application scenario, the evaluations are conducted under various models and LoRA numbers. For each model with a specific LoRA number, we conduct 10 sets of sending rates from 0 to the supported peak load of ELORA. Fig. 11 shows the average TTFT, TPOT, and supported peak load of ELORA and baselines.

As observed, ELORA reduces the TTFT and TPOT, as well as improves the supported peak load in all test cases. The average reduction of TTFT and TPOT is 45.7% and 37.8% compared to vLLM, and 43.3% and 31.4% compared to S-LoRA. The average supported peak load of ELORA is increased by 78.9% and 49.9% compared to vLLM and S-LoRA, respectively. For each case, we also repeat it 20 times, and the standard error in all cases of ELORA is 1.7% on average. For the P99/P95, ELORA decreases the TTFT and TPOT by 73.8%/76.1% and 61.2%/62.1% compared to vLLM, respectively, while those values are 66.1%/68.7% and 57.3%/58.9% compared to S-LoRA. The Multi-LoRA serving performance increase of ELORA originates from maintaining the usage dependencies between LoRAs and KV caches and retaining the most beneficial LoRAs and KV caches in GPU memory to eliminate cold-starts.

The reasons for ELORA decreasing the TPOT are as follows. ELORA efficiently retains hot KV caches in GPU to accelerate prefill and reduce corresponding computations, while baselines with suboptimal caching discard certain KVs which requires recomputation of prefill with increased computations. Like other works [1], [17], [32], ELORA serves numerous queries simultaneously with prefill and decode using the time-sharing GPU. Thus, the increased prefill computations block the decode of other queries in baselines, thereby increasing the TPOT compared to ELORA. In our evaluations, vLLM and S-LoRA lead to 1.38X and 1.87X computation time for the prefill compared to ELORA, respectively.

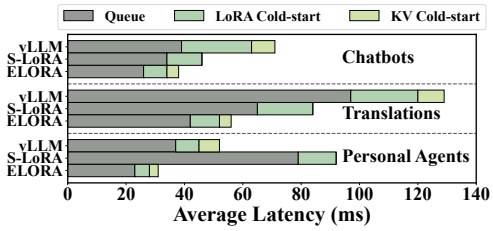
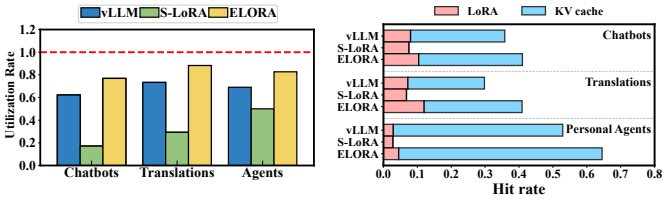


Fig. 12: The breakdown of the latency in TTFT.



(a) GPU memory utilization.

(b) Cache hit rate.

Fig. 13: The average GPU memory usage and cache hit rate.

Moreover, compared to vLLM, ELORA decreases more TTFT (average 49.4%) in the translation scenario than others (average 43.8%). This is because the distribution of LoRAs in this scenario varies more with the OPUS-100 and MAFT datasets. vLLM's static GPU memory partition results in poorer cache management, while ELORA maintains consistent performance. Compared to S-LoRA, ELORA achieves the best TTFT reduction (average 53.2%) in the personal agent than others (average 38.4%). This is because this scenario has the longest conversation length, and S-LoRA's drawback of not retaining history KVs is signified. Due to the similar reason, S-LoRA is worse than vLLM in most cases of personal agents.

C. Diving into the High Serving Performance

To better understand the source of performance improvement of ELORA, Fig. 12 shows the breakdown of the average queuing, LoRA cold-start, and KV cold-start latency in TTFT. We can observe that ELORA achieves the lowest queue, LoRA cold-start, and KV cold-start latency in all scenarios, indicating the highest GPU memory utilization efficiency.

For in-depth analysis, we sample the average GPU memory utilization of ELORA and baselines, shown in Fig. 13a. ELORA improves GPU memory utilization by 1.2X and 2.6X over vLLM and S-LoRA, respectively, due to its dynamic swapping of LoRAs and KV caches in a unified caching pool. By contrast, S-LoRA wastes GPU memory by not retaining history KV caches, while vLLM's static GPU memory partition makes the GPU memory for LoRAs or KVs under-utilized under dynamic loads. These factors also contribute to a lower queue and cold-start latency for ELORA, as shown in Fig. 12.

We also compare the average KV cache and LoRA hit rates of ELORA and baselines across different scenarios, as shown in Fig. 13b. ELORA increases the cache hit rate by 1.3X and 3.4X compared to vLLM and S-LoRA, respectively. This is because ELORA maintains the usage dependencies between LoRAs and KV caches to eliminate invalid KV caches, which

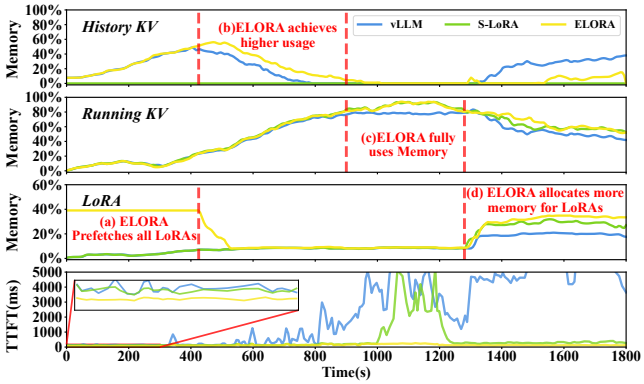


Fig. 14: GPU memory usage over time of different systems.

enhances the GPU memory utilization efficiency. Its efficient swapping strategy also prefetches appropriate KV caches and LoRAs into GPU memory. S-LoRA has the lowest hit rate because it does not reuse KV caches. As a result, ELORA achieves lower queue and cold-start latency in Fig. 12.

D. Analysis of GPU Memory Usage Over Time

In this subsection, we compare GPU memory usage between ELORA and baselines. We take the example of using the Llama2-34B, LoRA number of 100 for the chatbot. Other scenarios have similar results. Fig. 14 shows the GPU memory usage for history KV caches, LoRAs, and running KV caches.

From 0s-400s shown in **(a)**, ELORA proactively fetches all LoRAs into GPU memory based on the cost model to eliminate the cold-start overhead of LoRAs under low GPU memory pressure. In contrast, vLLM and S-LoRA load the LoRAs on demand, leading to a higher TTFT. From 400s-900s shown in **(b)**, as the load increases, ELORA swaps out some LoRAs and retains the most history KV caches in GPU memory due to the unified caching pool. In contrast, vLLM's static GPU memory partition retains fewer history KVs while S-LoRA directly discards them, leading to poorer KV cache reuse and higher TTFT. Moreover, the history KVs gradually decrease in this period as they are swapped out to free GPU memory for running KVs when the load increases.

From 900s-1300s in **(c)**, ELORA swaps out all history KV caches to free up GPU memory for running KV caches of the current inference. By contrast, the static GPU memory partition of vLLM results in the KV cache memory pool being exhausted to its maximum capacity (80%), leading to queuing and rapid growth of TTFT. Meanwhile, due to the increase of the load, directly discarding history KVs in S-LoRA causes lots of recalculations, resulting in a high TTFT. Lastly, from 1300s-1800s in **(d)**, with the increase of the required number of LoRAs, the static memory space for LoRAs (20%) in vLLM is exhausted, leading to high TTFT of vLLM. In contrast, ELORA can allocate more memory for LoRAs, leading to higher GPU memory usage and lower TTFT.

E. Effectiveness of the Cache Manager

In this subsection, we show the performance of ELORA-WOM, a variant of ELORA that does not maintain usage

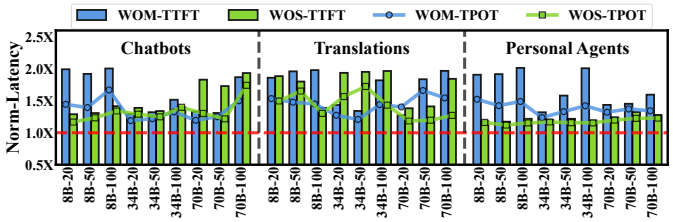


Fig. 15: The TTFT and TPOT of ELORA's variants normalized to ELORA, represent by bars and curves, respectively.

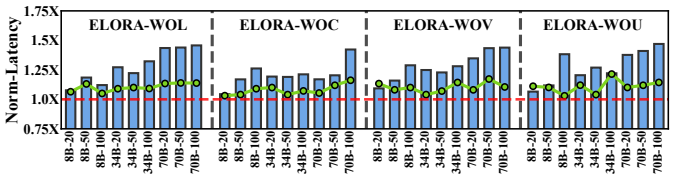


Fig. 16: The TTFT and TPOT when eliminating different components of ELORA's cost model under the chatbot.

dependencies between LoRAs and KV caches in unified GPU memory with the cache manager. ELORA-WOM still uses the cache swapper to swap in or out LoRAs or KV caches.

The blue bars and curves of Fig. 15 show the TTFT and TPOT of ELORA-WOM normalized to ELORA. As observed, the TTFT and TPOT of ELORA-WOM are higher than ELORA in all cases, with an average increase of 1.51X and 1.34X, respectively. We also sample the history KV caches during the inference and find ELORA-WOM suffers from an average of 48.6% invalid KV caches. Moreover, the supported peak load of ELORA-WOM is also decreased by 19.3%.

When ignoring the usage dependencies, lots of invalid KV caches occupy the GPU memory, leading to low GPU memory utilization and low serving performance.

F. Effectiveness of the Cache Swapper

In this subsection, we show the performance of ELORA-WOS, a variant of ELORA that uses a simple LRU policy to replace the cost model (Eq. 6) in the cache swapper. The usage dependencies between LoRAs and KV caches are still maintained with the cache manager during inference.

The green bars and curves of Fig. 15 show the TTFT and TPOT of ELORA-WOS normalized to ELORA. As observed, the TTFT and TPOT of ELORA-WOS are increased in all test cases, with an average increase of 1.42X and 1.29X, respectively. Moreover, the supported peak load of ELORA-WOS is also decreased by 18.6%.

Without ELORA's cost model, inappropriate LoRAs or KV caches will be swapped in or out when GPU memory is idle or busy, respectively. This results in more cold-starts of LoRAs and KVs, and decreases the performance.

G. Effectiveness of Different Parameters of the Cost Model

In this subsection, we investigate the effectiveness in the serving performance of different parameters in ELORA's cost model (Eq. 6). We construct four variants by removing different components of the cost model, i.e., ELORA-WOL, ELORA-WOC, ELORA-WOV, and ELORA-WOU eliminates the caching of enough LoRAs (Eq. 4), swap cost ($cost_i$ in Eq. 5), the visit frequency ($visit_i$ in Eq. 5), and the LRU considerations ($(1-sigmoid(t_i))$ in Eq. 5), respectively.

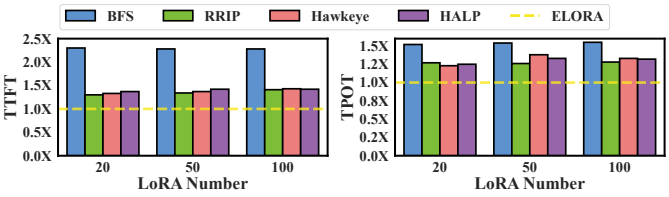


Fig. 17: The TTFT and TPOT of other caching policies.

The bars and curves of Fig. 16 show the TTFT and TPOT of the four variants normalized to ELORA for the chatbot, respectively. Other scenarios have similar results. All variants increase the TTFT and TPOT in all cases. ELORA-WOL, ELORA-WOC, ELORA-WOV, and ELORA-WOU averagely increase the TTFT by 1.21X, 1.19X, 1.25X, 1.21X, respectively, while values for TPOT are 1.11X, 1.09X, 1.15X, 1.14X. Moreover, the decrease of the supported peak load is 9.2%, 7.6%, 10.7%, and 9.3%.

Above results present that each parameter in the cost model has its individual effect to improve the serving performance.

H. Comparing to Other Cache Replacement Policies

In this subsection, we compare ELORA's caching scheme to other advanced policies, including the BFS, RRIP [24], Hawkeye [23], and HALP [44]. Different from ELORA that conducts swapping in a DFS way, BFS is implemented by prioritizing to swap-in/out an entire LoRA branch. The selection is based on the largest/smallest summation of each node's $Eval_i$ (Eq. 6) in this LoRA branch. For RRIP/Hawkeye/HALP, we utilize them to replace ERLOA's cost model, respectively. Fig. 17 shows the TTFT and TPOT of ELORA and other caching policies with the Llama2-34B in the chatbot scenario. Other models and scenarios have similar results.

We can first observe that BFS's TTFT and TPOT are 2.29X and 1.54X of ELORA's on average. The BFS that swaps the entire branch has coarse granularity compared to ELORA that swaps each node. Statistics show that a LoRA branch can occupy up to 35.6% of the GPU memory, causing BFS's GPU memory utilization to be reduced by 29.1% compared to ELORA's. The large branch swapping of BFS also causes high PCIE overhead, which is averagely 9.71X of ELORA's.

Moreover, results show the TTFT of RRIP/Hawkeye/HALP is 1.35X/1.38X/1.41X of ELORA, while the TPOT is 1.27X/1.31X/1.31X on average. This is because they are designed for other caching scenarios (e.g., content delivery network in Youtube [44]). By contrast, ELORA's caching scheme is customized for Multi-LoRA serving that incorporates more effective metrics (e.g., the loaded LoRA quantity).

I. Scalability with Lots of LoRAs

In this subsection, we investigate the effectiveness of ELORA under thousands of LoRAs, although real-world

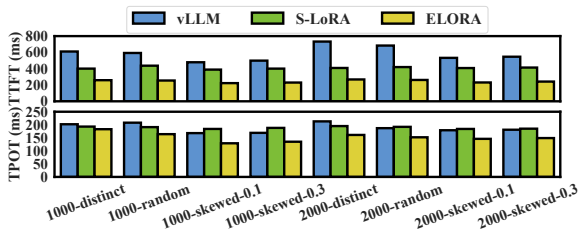


Fig. 18: The TTFT and TPOT of ELORA and baselines with different combinations of LoRA numbers and distributions.

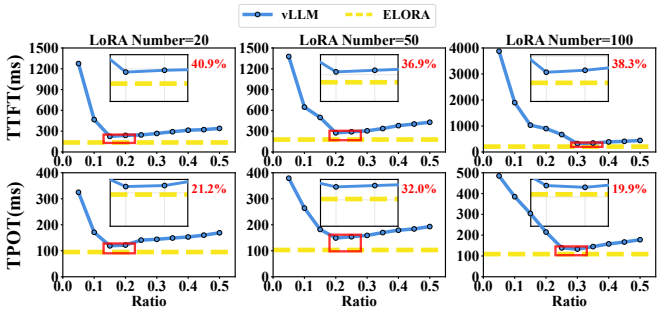


Fig. 19: The TTFT and TPOT of vLLM under different GPU memory allocation ratios for LoRAs in the chatbot scenario.

scenarios always only have tens of LoRAs [62], [4]. We also use the Llama2-34B model under the chatbot scenario as examples, and other scenarios have similar results. The LoRA number is 1000 or 2000, and we set the LoRA distributions to: 1) *Random*, each query randomly selects a LoRA to use. 2) *Distinct*, each query uses an individual LoRA. 3) *Skewed-x*, we construct queries using different LoRAs based on the Gaussian distribution and set different standard deviations x.

Fig. 18 shows the TTFT and TPOT of ELORA and baselines, respectively. We can observe that ELORA has the lower TTFT and TPOT in all test cases, with an average decrease of 48.7% and 21.9%, respectively. The above results present the scalability of ELORA under a large number of LoRAs.

J. Comparing to vLLM with Oracle GPU Allocation Ratio

In this subsection, we compare ELORA to vLLM with the oracle GPU memory allocation ratio for LoRAs. We also use Llama2-34B in the chatbot scenario as examples. We brute-force profile the GPU memory allocation ratios for LoRAs with a 0.05 step to get the oracle vLLM's performance.

Fig. 19 shows the TTFT and TPOT of vLLM under different GPU allocation ratios in the chatbot scenario. We omit the data after the ratio of 0.5, since the TTFT and TPOT of vLLM continuously increase after 0.5. We can observe that the TTFT or TPOT is first decreased and then increased after reaching a specific ratio. Moreover, the specific ratio increases with the increase of the required LoRA number (20, 50, and 100). Moreover, TTFT and TPOT in this oracle configuration for vLLM remain higher than those of ELORA, with an average increase of 38.7% and 24.4%, respectively.

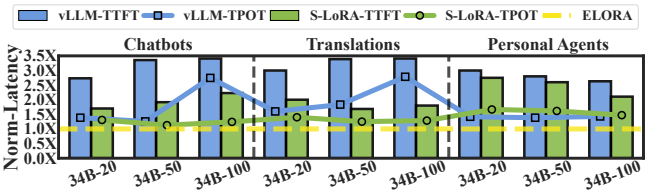


Fig. 20: The TTFT and TPOT of baselines normalized to ELORA's for the Llama2-34B when evaluating on NPUs.

The oracle vLLM is also equivalent to combining vLLM with S-LoRA (which can unifiedly manage GPU memory allocation for LoRAs and KV caches). Nevertheless, our results show that the oracle vLLM remains obviously inferior to ELORA. It is worth noting that such brute-force profiling to get the oracle vLLM is impractical in real dynamic serving.

K. Scalability on NPUs

In this subsection, we extend ELORA to NPUs to show the hardware scalability. We also use the Llama2-34B as examples here, and test it on four of our in-house NPUs. Each NPU has 256 TFLOPS FP16 and 64GB global memory space, as well as the interconnected bandwidth of the 4 NPUs is 168GB/s.

Fig. 20 shows the TTFT and TPOT of baselines normalized to ELORA. ELORA still achieves the lowest TTFT and TPOT, with an average decrease of 69.8% and 38.4% compared to vLLM, and 49.4% and 26.2% compared to S-LoRA, respectively. The average supported peak load of ELORA is increased by 96.1% and 65.3% compared to vLLM and S-LoRA. These results show that ELORA has strong scalability and can achieve improvements on various hardware.

L. Overhead of ELORA

Time Overhead: It mainly comes from the dependency tree matching and updating in the cache manager, and the swapping decisions of the cache swapper. For the cache manager, we employ an efficient trie tree for rapid matching and updating, which is commonly used in other works [26], [64]. Even if the GPU memory is fully utilized and the size of tree reaches the maximum, the average overhead for matching and updating of a query is less than 0.5ms. For swapping memory blocks of the cache swapper, the overhead during a query inference can be done within 5ms. The above overheads are acceptable relative to each query inference, which can take several seconds.

Memory Overhead: ELORA's cache manager records the memory address, visit frequency, last recent usage time, and size of each memory block, with an overhead of just 232Bytes per 16MB memory block (0.0014%). Moreover, ELORA's cache swapper stores the computed costs (Eq. 6) of memory blocks for swap-in or out decisions, with only 24Bytes per memory block (0.0001%). Both overheads scale linearly with the memory block number and are all negligible.

IX. Conclusion

In this paper, we propose ELORA to optimize the caching of LoRAs and KV caches to improve the Multi-LoRA serving

performance. ELORA's cache manager maintains the usage dependencies between KV caches and LoRAs based on a tree-based scheme with a unified caching pool. Based on this scheme, the invalid KV caches are eliminated to improve the GPU memory utilization. ELORA's cache swapper periodically determines the swap-in or out of LoRAs and KVs by using the cost model which reflects the benefits to the performance of queries. The evaluation results show that ELORA reduces the TTFT and TPOT by 45.7% and 37.8% on average, respectively, compared to the state-of-the-art works.

ACKNOWLEDGMENT

We sincerely thank our anonymous reviewers for their helpful comments and suggestions. This work is partially sponsored by the National Key Research and Development Program of China (2024YFB4505703), National Natural Science Foundation of China (62232011, 62302302), and Natural Science Foundation of Shanghai Municipality (25ZR1402241). Quan Chen is the corresponding author.

REFERENCES

[1] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in llm inference with sarathi-serve," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 117–134.

[2] K. Alizadeh, S. I. Mirzadeh, D. Belenko, S. Khatamifard, M. Cho, C. C. Del Mundo, M. Rastegari, and M. Farajtabar, "Llm in a flash: Efficient large language model inference with limited memory," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 12 562–12 584.

[3] Alpaca-lora. (2023) Instruct-tune llama on consumer hardware using alpaca-lora. [Online]. Available: https://github.com/tloen/alpaca-lora

[4] Apple. (2025) Introducing apple's on-device and server foundation models. [Online]. Available: https://machinelearning.apple.com/research/introducing-apple-foundation-models

[5] A. Aryan, A. K. Nain, A. McMahon, L. A. Meyer, and H. S. Sahota, "The costly dilemma: generalization, evaluation and cost-optimal deployment of large language models," *arXiv preprint arXiv:2308.08061*, 2023.

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

[7] B. Byrne, K. Krishnamoorthi, C. Sankar, A. Neelakantan, D. Duckworth, S. Yavuz, B. Goodrich, A. Dubey, K.-Y. Kim, and A. Cedilnik, "Taskmaster-1:toward a realistic and diverse dialog dataset," in *2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing*, Hong Kong, 2019.

[8] J. Chen, J. Shi, Q. Chen, and M. Guo, "Kairos: Low-latency multi-agent serving with shared llms and excessive loads in the public cloud," *arXiv preprint arXiv:2508.06948*, 2025.

[9] L. Chen, Z. Ye, Y. Wu, D. Zhuo, L. Ceze, and A. Krishnamurthy, "Punica: Multi-tenant lora serving," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 1–13, 2024.

[10] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, H. Zhang, B. Zhu, M. Jordan, J. E. Gonzalez *et al.*, "Chatbot arena: An open platform for evaluating llms by human preference," *arXiv preprint arXiv:2403.04132*, 2024.

[11] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[12] copilot. (2022) copilot. [Online]. Available: https://github.com/features/copilot

[13] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[14] L. Floridi and M. Chiriatti, "GPT-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[15] B. Gao, Z. He, P. Sharma, Q. Kang, D. Jevdjic, J. Deng, X. Yang, Z. Yu, and P. Zuo, "Cost-efficient large language model serving for multi-turn conversations with cachedattention," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 111–126.

[16] I. Gim, G. Chen, S.-s. Lee, N. Sarda, A. Khandelwal, and L. Zhong, "Prompt cache: Modular attention reuse for low-latency inference," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 325–338, 2024.

[17] R. Gong, S. Bai, S. Wu, Y. Fan, Z. Wang, X. Li, H. Yang, and X. Liu, "Past-future scheduler for llm serving under sla guarantees," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 798–813.

[18] Google. (2023) Bard. [Online]. Available: https://bard.google.com/

[19] S. Ha. Why can't i use multi-lora adapter and radix attention together? [Online]. Available: https://github.com/sgl-project/sglang/issues/2880

[20] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.

[21] C. Huang, Q. Liu, B. Y. Lin, T. Pang, C. Du, and M. Lin, "Lorahub: Efficient cross-task generalization via dynamic lora composition," *arXiv preprint arXiv:2307.13269*, 2023.

[22] N. Iliakopoulou, J. Stojkovic, C. Alverti, T. Xu, H. Franke, and J. Torrellas, "Chameleon: Adaptive caching and scheduling for many-adapter llm inference environments," *arXiv preprint arXiv:2411.17741*, 2024.

[23] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 78–89.

[24] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 60–71, Jun. 2010.

[25] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[26] W. Lee, J. Lee, J. Seo, and J. Sim, "InfiniGen: Efficient generative inference of large language models with dynamic KV cache management," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 155–172.

[27] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 3045–3059.

[28] S. Li, W. Li, C. Cook, C. Zhu, and Y. Gao, "Independently recurrent neural network (indrnn): Building a longer and deeper rnn," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 5457–5466.

[29] S. Li, H. Lu, T. Wu, M. Yu, Q. Weng, X. Chen, Y. Shan, B. Yuan, and W. Wang, "Toppings: CPU-assisted, rank-aware adapter serving for llm inference," in *Proc. USENIX ATC*, 2025.

[30] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," *arXiv preprint arXiv:2101.00190*, 2021.

[31] Y. Li, H. Wen, W. Wang, X. Li, Y. Yuan, G. Liu, J. Liu, W. Xu, X. Wang, Y. Sun *et al.*, "Personal llm agents: Insights and survey about the capability, efficiency and security," *arXiv preprint arXiv:2401.05459*, 2024.

[32] C. Lin, Z. Han, C. Zhang, Y. Yang, F. Yang, C. Chen, and L. Qiu, "Parrot: Efficient serving of LLM-based applications with semantic variable," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 929–945.

[33] S.-Y. Liu, C.-Y. Wang, H. Yin, P. Molchanov, Y.-C. F. Wang, K.-T. Cheng, and M.-H. Chen, "Dora: Weight-decomposed low-rank adaptation," in *Forty-first International Conference on Machine Learning*.

[34] Llama3. (2024) Llama3. [Online]. Available: https://huggingface.co/collections/meta-llama/meta-llama-3-66214712577ca38149ebb2b6

[35] H. Luo, Q. Sun, C. Xu, P. Zhao, Q. Lin, J. Lou, S. Chen, Y. Tang, and W. Chen, "Arena learning: Build data flywheel for llms post-training via simulated chatbot arena," *arXiv preprint arXiv:2407.10627*, 2024.

[36] NVIDIA. A tensorrt toolbox for optimized large language model inference. [Online]. Available: https://github.com/NVIDIA/TensorRT-LLM

[37] OpenAI. (2020) Chatgpt. [Online]. Available: https://openai.com/blog/chatgpt

[38] PyTorch Contributors. (2023) torch.stream — pytorch 2.0.1 documentation. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.Stream.html

[39] R. Qin, Z. Li, W. He, J. Cui, F. Ren, M. Zhang, Y. Wu, W. Zheng, and X. Xu, "Mooncake: Trading more storage for less computation—a kvcache-centric architecture for serving llm chatbot," in *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, 2025, pp. 155–170.

[40] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. 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, pp. 205–218.

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

[42] Y. Sheng, S. Cao, D. Li, C. Hooper, N. Lee, S. Yang, C. Chou, B. Zhu, L. Zheng, K. Keutzer *et al.*, "SLoRA: Scalable serving of thousands of lora adapters," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 296–311, 2024.

[43] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Re, I. Stoica, and C. Zhang, "FlexGen: High-throughput generative inference of large language models with a single GPU," in *Proceedings of the 40th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, 23–29 Jul 2023, pp. 31 094–31 116.

[44] Z. Song, K. Chen, N. Sarda, D. Altınbüken, E. Brevdo, J. Coleman, X. Ju, P. Jurczyk, R. Schooler, and R. Gummadi, "HALP: Heuristic aided learned preference eviction policy for YouTube content delivery network," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1149–1163. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/song-zhenyu

[45] R. C. Staudemeyer and E. R. Morris, "Understanding LSTM–a tutorial into long short-term memory recurrent neural networks," *arXiv preprint arXiv:1909.09586*, 2019.

[46] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton-Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," *CoRR*, vol. abs/2307.09288, 2023.

[47] M.-F. Tsai, C.-J. Wang, and P.-C. Chien, "Discovering finance keywords via continuous-space language models," *ACM Transactions on Management Information Systems (TMIS)*, vol. 7, no. 3, pp. 1–17, 2016.

[48] vLLM Community. vLLM: A high-throughput and memory-efficient inference and serving engine for llms. [Online]. Available: https://github.com/vllm-project/vllm

[49] J. Wang, H. Xu, H. Jia, X. Zhang, M. Yan, W. Shen, J. Zhang, F. Huang, and J. Sang, "Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration," *Advances in Neural Information Processing Systems*, vol. 37, pp. 2686–2710, 2024.

[50] Z. Wang, J. Liang, R. He, Z. Wang, and T. Tan, "LoRA-Pro: Are low-rank adapters properly optimized?" in *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

[51] Wikipedia. (2023) Trie. [Online]. Available: https://en.wikipedia.org/wiki/Trie

[52] B. Wu, R. Zhu, Z. Zhang, P. Sun, X. Liu, and X. Jin, "dLoRA: Dynamically orchestrating requests and adapters for lorallm serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 911–927.

[53] J. Yao, H. Li, Y. Liu, S. Ray, Y. Cheng, Q. Zhang, K. Du, S. Lu, and J. Jiang, "CacheBlend: fast large language model serving for rag with cached knowledge fusion," in *Proceedings of the Twentieth European Conference on Computer Systems*, ser. EuroSys '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 94–109.

[54] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for transformer-based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 521–538.

[55] L. Yu, J. Lin, and J. Li, "Stateful large language model serving with pensieve," in *Proceedings of the Twentieth European Conference on Computer Systems*, 2025, pp. 144–158.

[56] B. Zhang, P. Williams, I. Titov, and R. Sennrich, "Improving massively multilingual neural machine translation and zero-shot translation," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, D. Jurafsky, J. Chai, N. Schluter, and J. Tetreault, Eds. Online: Association for Computational Linguistics, Jul. 2020, pp. 1628–1639.

[57] B. Zhang, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, vol. 11, 2016.

[58] H. Zhang, X. Ji, Y. Chen, F. Fu, X. Miao, X. Nie, W. Chen, and B. Cui, "PQCache: Product quantization-based kvcache for long context llm inference," *Proc. ACM Manag. Data*, vol. 3, no. 3, Jun. 2025.

[59] Q. Zhang, M. Chen, A. Bukharin, P. He, Y. Cheng, W. Chen, and T. Zhao, "Adaptive budget allocation for parameter-efficient fine-tuning," in *The Eleventh International Conference on Learning Representations*, 2023.

[60] Y. Zhang, Í. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 724–739.

[61] Y. Zhang. Release v0.4.8. [Online]. Available: https://github.com/sgl-project/sglang/releases/tag/v0.4.8

[62] J. Zhao, T. Wang, W. Abid, G. Angus, A. Garg, J. Kinnison, A. Sherstinsky, P. Molino, T. Addair, and D. Rishi, "LoRA Land: 310 fine-tuned llms that rival gpt-4, a technical report," *arXiv preprint arXiv:2405.00732*, 2024.

[63] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 595–46 623, 2023.

[64] L. Zheng, L. Yin, Z. Xie, C. L. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez *et al.*, "Sglang: Efficient execution of structured language model programs," *Advances in neural information processing systems*, vol. 37, pp. 62 557–62 583, 2024.

[65] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 193–210.

[66] W. Zhu, H. Liu, Q. Dong, J. Xu, S. Huang, L. Kong, J. Chen, and L. Li, "Multilingual machine translation with large language models: Empirical results and analysis," *arXiv preprint arXiv:2304.04675*, 2023.