Improving the Serving Performance of Multi-LoRA Large Language Models via Efficient LoRA and KV Cache Management

Hang Zhang*, Jiuchen Shi*, Yixiao Wang, Quan Chen†, Yizhou Shan, Minyi Guo

Shanghai Jiao Tong University

Abstract

Multiple Low-Rank Adapters (Multi-LoRAs) are gaining popularity for task-specific LLM applications. For multi-LoRA serving, caching hot KV caches and LoRA adapters in high bandwidth memory of accelerations can improve inference performance. However, existing Multi-LoRA inference systems fail to optimize serving performance like Time-To-First-Toke (TTFT), neglecting usage dependencies when caching LoRAs and KVs. We therefore propose FASTLIBRA, a Multi-LoRA inference caching system to optimize the serving performance. FASTLIBRA 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 the inference with a unified caching pool. The cache swapper determines the swap-in or out of LoRAs and KV caches based on a unified cost model, when the HBM is idle or busy, respectively. Experimental results show that FASTLIBRA reduces the TTFT by 63.4% on average, compared to state-of-the-art works.

1 Introduction

Large Language Models (LLMs) are now widely used to understand and generate human-like text [8, 30]. While it is cost-inefficient to train LLMs for different tasks, parameter-efficient fine-tuning [19, 42] that freeze the large-scale base model and fine tunes multiple Low-Rank Adapters (Lo-RAs) for different tasks are increasingly popular [14, 28]. For instance, in applications like chatbot [18, 32], personal agents [3,26], and multi-language translation [47,48], multiple LoRAs can be tuned for different user languages 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 [1,22]. Researchers also

proposed to reuse history KVs for queries with the same prefix [17, 44, 46, 53], boosting performance in iterative tasks.

To improve the serving performance of such Multi-LoRA applications, many works have investigated to cache the base model, the KVs [16, 34, 46] or "hot" LoRA adapters (LoRAs in short) [21, 24, 37], in the high bandwidth memory (HBM) of accelerators (e.g., global memory of Nvidia GPUs or global memory of various AI accelerators [9,11,13]). While caching both KVs and LoRAs can improve inference performance, vLLM [22] and SGLang [53] proposed to cache both LoRAs and KVs. Figure 1 shows an example of caching base model, LoRAs, and KVs. In general, LoRAs have separate KV caches (e.g., LoRA-1 and LoRA-2). Moreover, the HBM space is statically partitioned for caching LoRAs and KVs, because these works allocate different sizes of memory blocks for LoRAs and KVs, preventing their sharing with each other [41].

When a user query is received, the serving system checks whether the required LoRA and KVs are already in the HBM or not. 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/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 HBM space partition can be identified in a "brute-force" way. However, production traces [35, 50, 52] show that the distributions are dynamic. In such scenario, we observe that static HBM partition and independent cache management suffer from low efficiencies in both 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 Figure 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 HBM. However, LoRA-1 is swapped out earlier due to the limited HBM 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 HBM space of invalid KVs (e.g., KV1-3) were used to cache LoRA-1, Query-1 could run immediately. Invalid

^{*}Hang Zhang and Jiuchen Shi contributed equally to this work.

[†]Quan Chen is the corresponding author.

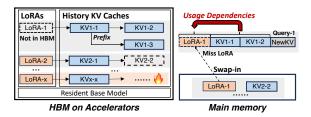


Figure 1: An example caching state to show the *Usage Dependencies* between LoRAs and KV caches.

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 [22] suffers from up to 46.5% invalid KV caches.

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 Figure 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 HBM which prevents them to be swapped-in. However, with static HBM partition of LoRAs and KVs, their swap-in or out can only be managed separately, making it hard to uniformly balance the usage of LoRAs and KVs in HBM.

To address the above problems, a scheme is required to integrate the usage dependency for each LoRA and its KVs with a unified caching pool. By such means, we can keep valid KVs in HBM as much as possible to improve the HBM efficiency. Moreover, unified managing the caching of LoRAs and KVs helps in reducing the Time-To-First-Token (TTFT) and Time-Per-Output-Token (TPOT) of each query. It is challenging to to balance the usage of LoRAs and KVs.

Based on the two insights, we propose FASTLIBRA, a Multi-LoRA inference caching system that optimizes the caching of LoRAs and KVs considering the usage dependency. FASTLIBRA aims to reduce the TTFT, while maximizing the peak serving throughput 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 a tree-based scheme with a unified caching pool, where nodes are KV caches or LoRAs and edges are their dependencies. To maintain usage dependencies during inference, LoRAs or KV caches are inserted or removed from leaves in the HBM to keep the tree connected. Based on the unified caching management of the cache manager, the cache swapper periodically determines the swap-in/out of LoRAs and KVs by using a unified cost model that precisely reflects the benefits of swap-in/out LoRAs and KVs to the performance of queries. This paper makes three contributions.

 Investigating the caching management of LoRAs and KVs for the Multi-LoRA inference. The analysis motivates us to maintain the usage dependencies between LoRAs and KV caches, and unified manage their swapin/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 HBM, eliminating the intra-LoRA inefficiency.
- The design of a cost model that guides the swap-in/out
 of LoRAs and KVs. The model enables the unified
 swap-in and out of LoRAs and KVs, eliminating the
 inefficiency due to the inter-LoRA interference.

We have implemented FASTLIBRA on top of vLLM [31] and evaluated it with Llama-7B/13B/34B models [39] on four High-performance NPUs with three typical scenarios (chatbot [7,29], multi-language translation [47], and personal agents [5]). The design of FASTLIBRA does not rely on any specific hardware architecture, and it is applicable to other accelerators. Experimental results show that FASTLIBRA reduces the TTFT and TPOT by 63.4% and 40.1%, respectively, as well as improves the peak serving throughput by 35.2%, compared to state-of-the-art Multi-LoRA inference systems.

2 Background and Motivation

In this section, we first introduce the background of Multi-LoRA serving with caching, then investigate the inefficiency of current Multi-LoRA serving systems.

2.1 Multi-LoRA Serving with Caching

Multi-LoRA. LoRA is a popular method for efficiently finetuning pre-trained LLMs by adding lightweight adapters to original weights [19]. Instead of updating all parameters of a model, LoRA only learns a pair of small low-rank matrices that modify the original weights. These matrices are much smaller than the original weight matrix, which can reduce the computational cost and memory usage.

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 [20,51]. 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 HBM for usage before inferencing. Queries using different LoRAs can be processed in a single batch using Segmented Gather Matrix-Vector multiplication (SGMV) [6, 37], improving both efficiency and throughput.

KV Caches for Multi-LoRAs. The LoRAs need to be loaded into the HBM for usage during the inference [19,

37]. Moreover, most LLMs use a decoder-only transformer to predict the next token with KV caches computed from previous tokens [8, 15]. When a query matches an existing prefix, the stored KV caches are reused to reduce HBM usage and eliminate redundant computations, such as in multi-turn dialogues [16, 17]. Therefore, maintaining the history KV caches in HBM can maximize the reuse.

Each LoRA adds a low-rank branch to the original weights that participate in the KV cache computation. For each query *q* using LoRA *t*, the KV cache is computed as:

$$KV_Cache_{q,t} = W_{k,v}q + A_tB_tq$$
 (2)

Therefore, as mentioned in Figure 1, KV caches for different LoRAs are stored separately due to task-specific modifications by each LoRA branch [14, 19].

The separate storage further increases contention for limited HBM space, and thus the KV caches and LoRAs are usually offloaded to main memory and swapped-in/out ondemand [16, 37]. However, this can cause cold-starts when loading them back into HBM, affecting performance metrics like TTFT and TPOT. To reduce this overhead, we need to pre-cache "hot" history KV caches and LoRAs into HBM.

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

The LLM inference typically has the prefill and decode stages [1,45], corresponding to two performance metrics of the Time to First Token (TTFT) and Time Per Output Token (TPOT). The above workflow can introduce overheads due to the queue to wait for HBM space, LoRA cold-starts, and KV cold-starts, affecting both TTFT and TPOT.

2.2 Low Multi-LoRA Serving Performance

We use vLLM [41] that caches both LoRAs and history KVs [19,53] as the representative serving system to perform the investigation. It allocates fixed HBM space for LoRAs and KV caches, and utilizes Least Recent Use (LRU) policy to swap-in/out LoRAs or KV caches in the respective HBM area. vLLM sets a predefined allocation ratio of the HBM space for the LoRA (empirically set to be 0.2) and the memory block size to 32, referring to the latest version of vLLM [41].

Three Multi-LoRA scenarios, *chatbot*, *multi-language translation*, and *personal agent* are used as benchmarks in the investigation. LMSYS-33k [52], Opus-100 [47], and Taskmaster [5] datasets are used to generate the queries, respectively. Since datasets Opus-100 and Taskmaster lack the timing information, the same to state-of-the-art Multi-LoRA management works [21, 37, 43], we use Microsoft Azure Function

Table 1: Experiment specifications

	Specifications
Hardware	Arm CPU (192 cores), 256GB main memory
	NPU with 256 TFLOPS FP16 and 64GB HBM \times 4
	PCIe × 16 Gen4.0
	Llama-7B, Llama-13B, and Llama-34B
Software	LMSYS-33K [52], Opus-100 [47], Taskmaster [5],
	_
	Microsoft Azure Function trace [35,50]
⊋ 9000	Chatbots KVs' HBM exhausted
Ĕ 7000	CHRIDOIS AVS FIDM CAUSISING
€ 5000	M^W. □
£ 3000	1 A AA MART WALL
E	0 200 400 600 800 1000 1200 1400 1600 1800
9000 E 7000	Translations KVs' HBM LoRAs' HBM exhausted №
₹ 7000 5000	exhausted
E 3000	V.V. 1400V W WV W
⊢ 1000	0 200 400 600 800 1000 1200 1400 1600 1800
2 3000 2 2500	Personal Agents KVs' HBM exhausted
£ 2000	الملاملات المسترين
₹ 1500 ₹ 1000	1 IV WWWIND AMARKA
E 500	1
•	0 200 400 600 800 1000 1200 1400 1600 1800
	Time(s)

Figure 2: The TTFT of vLLM for various scenarios.

Trace [35,50] to adapt its query arriving timing information. Moreover, we use Llama-7B/13B/34B as the base models, and conduct the experiments on four High-performance NPUs. Table 1 summarizes the hardware and software configurations.

Figure 2 shows the TTFT of vLLM for the three benchmarks with the Llama-7B base model. Experiments with other base models show similar observations, as shown in Section 6. With varying loads, we observe that vLLM experiences significantly high TTFT at certain periods, due to insufficient HBM space allocation for KV caches or LoRAs. As statistics, the TTFT of the three benchmarks are 1032.4ms, 1905.1ms, and 730.8ms on average, respectively. This is because the static HBM allocation of vLLM cannot dynamically adapt to the varying loads in Multi-LoRA serving. The HBM allocation is static because vLLM allocates memory blocks with different sizes for LoRAs and KVs according to their respective requirements [41]. Memory blocks in the KV cache HBM area cannot be used for LoRAs, and vice versa, making it impossible to dynamically adjust the pool sizes.

While redeployment can change the HBM partition, it results in significant overhead that blocks the normal inference for tens of seconds [2, 4]. Moreover, even if dynamic HBM allocation is achieved with more fine-grained memory blocks [6,21,37], it is still challenging to define an appropriate allocation policy with varying loads of LoRA adapters.

2.3 Diving into Underlying Reasons

Our investigations show that the poor serving performance is caused by 1) inefficient HBM usage without considering intra-LoRA usage dependencies, and 2) inappropriate swap-in/out of KVs and LoRAs when LoRAs have varying loads.

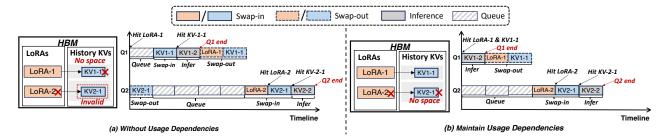


Figure 3: Examples of serving two queries under: (a) without usage dependencies, and (b) maintaining usage dependencies.

2.3.1 Inefficient HBM Usage

Figure 3 shows an example of serving two queries (Q1 and Q2) of two LoRA adapters (LoRA-1 and LoRA-2).

As shown in Figure 3(a), it is possible that KV2-1 is cached while the corresponding LoRA-2 is not in the HBM, without considering usage dependencies between LoRA and its KVs. Prior work (e.g., vLLM, and SGLang) all manage LoRAs and KVs in this way. In this case, KV2-1 is "invalid", since Q2 cannot run at all without the LoRA adapter. At the same time, Q1 is also blocked although its LoRA adapter is cached, because it needs to wait for the required KV1-1 to be swapped in, before which KV2-1 should be swapped out to free some HBM space. After Q1 returns, Q2 needs to swap-in the LoRA-2 and KV2-1 again to perform the inference. Without considering the usage dependency, the serving system causes redundant swap-in/out, greatly increasing the queuing overhead. From our evaluations in Section 6, vLLM results in 48.1% invalid KV caches on average.

Figure 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 history KV1-1 are in the HBM. 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/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 KVs, the limited HBM space is not efficiently used.

2.3.2 Inappropriate Swap-in/out of KVs and LoRAs

Previous works [41,53] separately manage LoRAs and KVs in individual HBM areas, and adopt caching strategies like Least-Recent-Used (LRU) for swap-in/out. These works cannot dynamically balance the HBM usage for LoRAs and KVs when the loads of different LoRAs change.

Take the benchmark *translation* in Figure 2 as an example. The TTFT increases up to 5036.1ms and 8617.9ms during the period of 700s-1100s and 1100s-1800s, respectively. Correspondingly, Figure 4 shows the HBM utilization rates of the LoRA and the KV parts. After looking into the detailed serving trace, we find that the long TTFT originates from different

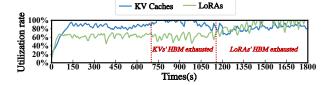


Figure 4: The utilization rate of the HBM space allocated to LoRAs and KV caches over time in the translation scenario.

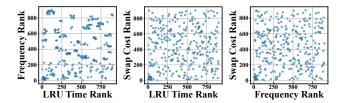


Figure 5: The relationships among the visited frequency, swap costs, LRU of the KV caches or LoRAs.

reasons. During 700s and 1100s, the HBM space for KVs is exhausted while the utilization rate of HBM space for LoRAs is 58.9% on average. In this case, the frequent swap-in/out of KVs results in the long TTFT. During 1100s and 1800s, the HBM space for LoRAs is exhausted on the contrary, because queries of more LoRA adapters are received during that period. According to the trace, queries of 41 LoRAs are received before 1100s, while that is 75 after 1100s.

It would be beneficial to dynamically balance the HBM usage of LoRAs and KVs. However, even if the HBM space of LoRAs and KVs is dynamically managed through fine-grained memory blocks, relying on the LRU policy to determine the swap-in/out is not efficient. This is because the TTFT is related to many factors, like the swap cost and visited frequency. Figure 5 shows the relationship between the visited frequency, LRU time, and swap cost of each KV cache and LoRA. In the figure, each point represents a LoRA or KV cache, and its *x*-axis or *y*-axis represents its corresponding ranks of LRU Time/Frequency/Swap Cost. As observed, the points are randomly distributed, which means there is not clear correlation among these key factors.

Relying on LRU to manage the HBM space is not efficient to minimize the TTFT, even if dynamic HBM usage is enabled.

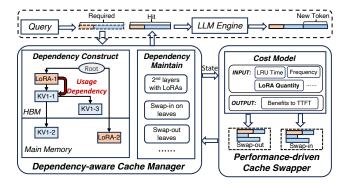


Figure 6: Design overview of FASTLIBRA.

3 FASTLIBRA Methodology

In this section, we summarize the challenges of FASTLIBRA, and introduce the overview of FASTLIBRA.

3.1 Challenges of FASTLIBRA

According to the above analysis, two technical challenges should be addressed to resolve the above problems.

Firstly, the neglect of the intra-LoRA usage dependencies between LoRAs and KV caches brings invalid KV caches in HBM. This can prevent other useful KVs or LoRAs from being loaded, increasing the cold-start overheads. A suitable scheme is required to construct the usage dependencies among the LoRAs and KV caches and consistently maintain the dependencies when serving the queries.

Secondly, when loads of different LoRAs vary, the required HBM space for caching LoRAs and the hotness of KV caches of different LoRAs change accordingly. Thus, we need to balance the HBM usage for LoRAs and KVs, and swap-in or swap-out the appropriate KVs and LoRAs to optimize the TTFT. An appropriate mechanism is needed to access the benefits or harms of swapping-in or out each LoRA or KV cache to the TTFT of future queries.

3.2 Overview of FASTLIBRA

Figure 6 shows the design overview of FASTLIBRA. It comprises a *dependency-aware cache manager* and a *performance-driven cache swapper*. The cache manager manages LoRAs and KV caches in the HBM and main memory together, and maintains usage dependencies to eliminate invalid KV caches. After each monitor interval, the cache swapper decides the swap-in or swap-out LoRAs and KV caches from the main memory or HBM based on a unified cost model. The cache manager then conducts specific swapping operations.

The most challenging part is managing LoRAs and KV caches based on the usage dependencies to eliminate invalid KV caches. FASTLIBRA introduces the tree-based dependency maintenance scheme to address this problem. In the dependency tree, nodes represent LoRAs or KV caches, and

edges represent the usage dependencies among them. When a query arrives, its required LoRAs and KVs in this tree are matched according to the Depth-First-Search (DFS). To maintain the usage dependencies, this scheme places the LoRAs on the second layer, as well as only swaps-out leaf nodes in the HBM and swaps in root nodes in the main memory (Section 4).

When the loads of queries using different LoRA branches change, the used LoRA number can increase and the hotness of some KV caches of some LoRAs changes. FASTLIBRA periodically decides the swap-in/out of different LoRAs and KV caches based on performance metrics like LRU time, visit frequency, the LoRA quantity, etc. The challenging part here is to establish the cost model to directly evaluate the benefits or harms to the TTFT of swapping-in/out each LoRA or KV cache (Section 5).

Specifically, FASTLIBRA works as follows. 1) The cache manager organizes LoRAs and KV caches in HBM and main memory, constructing their usage dependencies into a dependency tree. 2) During inference, it inserts newly loaded LoRAs into the second layer of the tree, and inserts or deletes KV cache nodes at the leaves of their corresponding LoRA branches. 3) After each monitor interval, the cache swapper retrieves the states of nodes from the cache manager, and decides the swapped-in/out KV caches and LoRAs when the HBM is idle/busy. The decisions are made using a cost model that considers metrics like LRU time, visit frequency, and loaded LoRA quantity, and accesses their impact on TTFT. 4) The swap-in/out decisions are sent back to the cache manager for performing corresponding memory operations. 5) For a new query, if its LoRAs or KV caches are in main memory but HBM is full, the cache manager swaps out "cold" LoRAs or KVs based on the cache swapper's decisions, then swaps in the required ones. 6) This query proceeds for inference to generate the next token with the required LoRA and KVs.

FASTLIBRA can be adapted to other LLM inference engines [1,45,53] by replacing their memory management module with few modifications. It applies to LLMs based on decoder-only transformer [8,15,39] that cover popular LLM practical scenarios. The design of FASTLIBRA does not rely on any specific hardware architecture, and it is applicable to other accelerators.

4 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.

4.1 Usage Dependency Constructing

As we analyzed in Section 2.3.1, a LoRA and its corresponding KV caches have their inherent usage dependencies. When ignoring these dependencies, invalid KV caches will occupy

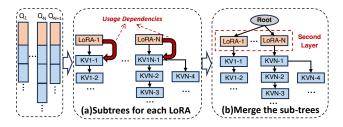


Figure 7: The constructing process of the usage dependencies among LoRAs and KV caches.

the HBM space, leading to low performance for Multi-LoRA inference. In this subsection, we adopt a tree-based scheme to construct the usage dependencies among KV caches and LoRAs used by the queries in the HBM and main memory, as shown in Figure 7.

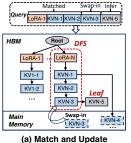
For each specific query, it will first match the required LoRA and then its corresponding KV caches. The KV caches corresponding to different tokens also have their matching orders. For instance, in the sentence "To be or not to be", the KV cache for the token "To" should be matched in front of "be". Therefore, as shown in Figure 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.

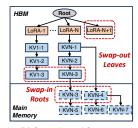
Since these subtrees constructed above are still separate, we need to merge these subtrees into a unified one, as shown in Figure 7(b). We use a virtual root node to connect the subtrees for different LoRAs to form a unified tree. In this way, all LoRA nodes are placed on the second layer of the tree,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

4.2 Dependency Maintaining During Inference

To maintain the usage dependencies among LoRAs and KV caches during query inference, we need to correctly match and update the LoRAs and KV caches on the dependency tree. Moreover, we need to swap-in and swap-out appropriate nodes in the dependency tree according to the cache swapper's decisions (Section 5) when the HBM is busy or idle.

For the matching and updating, as shown in Figure 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 HBM asynchronously. Then, within





h and Update (b) Swap-in and Swap-out

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

the subtree of this LoRA branch, this query begins to match history KV caches according to Deep-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 HBM. 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 continuously be inserted into the leaves of this LoRA branch.

When the HBM is idle or busy, some LoRAs or KV caches needs to be swapped-in or swapped-out to fully utilize the HBM and main memory resources. As shown in Figure 8(b), the cache manager will control the swapping-out to start from the leaf nodes in the HBM, 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 that reside in the HBM are valid ones, thus the HBM can be fully utilized.

4.3 Implementing Unified Tree-based Caching

FASTLIBRA is implemented based on vLLM [41] with an extra 8324 and 1644 lines of Python and C++ codes, respectively. We describe how FASTLIBRA establishes unified HBM and main memory pool for LoRAs and KVs, as well as implementing the usage dependency tree and asynchronous swap-in/out.

Unified Caching Pool for LoRAs and KVs: To achieve a unified memory pool for HBM and main memory, we extend the BlockManager of vLLM [22,53]. During the initialization phase, both HBM and main memory are partitioned into memory blocks of the same size. This block-wise memory allocation policy is similar to S-LoRA [37], but we also extend this pool to store history KV caches. To retain LoRAs,

we perform block-wise partitioning of LoRAs along the rank dimension. Since the other dimensions of LoRA align with those of the KV caches, this approach ensures full alignment with the KV cache and avoids memory fragmentation.

When some LoRAs or KV caches are swapped-out, FASTLIBRA recycles their memory blocks in the memory pool of HBM for future allocations. Moreover, whenever a new KV cache is generated on HBM, it will be directly retained in the HBM without deciding to place it in the HBM or main memory. This eliminates redundant memory operations.

Usage Dependency Tree: We build the usage dependency tree on top of the unified memory pool, which logically records the memory address of each memory block without altering the actual physical memory allocation pattern. We utilize an efficient trie tree [12] to implement the usage dependency tree whose node matching and updating is fast as less than 1ms. In the usage dependency tree, each path from the root to a leaf represents a conversation record, and the subtrees with the same parent node have a shared prefix.

The node label for each KV cache node is the token sequence, and for each LoRA node is the LoRA ID. Each node also retains the corresponding important information, i.e., visit frequency, last recent usage time, and the node size. These data will be updated when each node is generated, matched, or swapped-in/out.

Asynchronous Swapping-in/out: To further mitigate the cold start overhead, we adopted an asynchronous swap-in/out strategy similar to existing work [16]. We use the Stream library in Torch [33] to implement this. After a query arrives, if the corresponding LoRA or KV caches for the query is not in HBM, we swap in the corresponding memory blocks and just let this query wait while inferring other requests that are ready. This realizes the overlap of inference and data transferring, thus improving the inference efficiency.

5 Performance-driven Cache Swapper

In this section, we first analyze the impact of the quantity of LoRAs loaded into HBM on TTFT. Then, we introduce a cost model considering multiple metrics to access the benefits to TTFT of swapping-in/out different LoRAs and KV caches. At last, we introduce the workflow of the cache swapper.

5.1 Considering LoRA Quantity on TTFT

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

Figure 9 shows the TTFT under the chatbot scenario with different HBM allocation ratios for LoRAs in the vLLM. In this experiment, the used LoRA number is set at 50 and 100, as well as the average sending rate is 2 queries per second. We can observe that the TTFT reduces significantly before reaching a target ratio, and the target ratio is increased when the required LoRA number changes from 50 to 100. This is

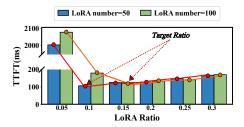


Figure 9: The TTFT of vLLM under different HBM allocation ratios for LoRAs.

because the query inference can only start once the required LoRA is matched in HBM, otherwise the query is queued. Insufficient LoRA loading quantity in HBM 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.

We estimate the current required LoRA quantity 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, we calculate the expected number of LoRAs required for inference (Low_{lora}) as follows:

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

In this formula, the fe_i represents the probability that the LoRA i is present in recent batch, i.e., 1 minus the probability that none of the queries in this batch use it. We consider Low_{lora} as an important aspect of cost model.

5.2 Cost Model to Access Benefits to TTFT

When performing swap-in or swap-out operations for LoRAs and KV caches, the goal is to retain the most valuable KVs and LoRAs in HBM as much as possible, thereby optimizing the TTFT for incoming queries. To achieve this, our key idea is to design a cost model to evaluate the expected benefits to TTFT of retaining a specific KV cache or LoRA *i* in HBM.

As analyzed in Section 2.3.2 and Section 5.1, the cost model needs to try to load sufficient LoRAs, and consider metrics with the visited frequency, the LRU time, and the cost of swap-in/out of nodes. Thus, we first define the *LoRA_Evai* as the reward coefficient that encourages the loaded LoAR quantity to be close to the *Lowlora* (Equation 3) as:

$$LoRA_Eval_i = max(1, \frac{Low_{lora}}{Now_{LoRA}})$$
 (4)

In this formula, $LoRA_Eval_i$ gives a larger reward when Now_{LoRA} is farther from the Low_{lora} , and is set to 1 when Now_{LoRA} is greater than or equal to Low_{lora} .

Then, we define the $Retain_Eva_i$ to represent the expected benefit of retaining node i in HBM, which can be estimated as

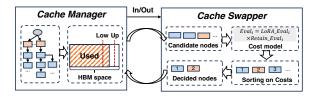


Figure 10: The operation workflow of the cache swapper.

the expected cold-start latency reduction to TTFT for future queries after caching it in HBM. The definition is as:

$$Retain_Eval_i = cost_i \times prob_i \times (1 - sigmoid(t_i))$$
 (5)

In this formula, the first item transfer $cost_i$ can be computed using the PCIe bandwidth and size of the KV or LoRA, and the second item visit frequency probability $prob_i$ is 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 [38], whose t_i represents the time difference between the current time and LRU time. This function enhances the weight of KVs or LoRAs that were visited more recently.

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:

$$Eval_{i} = LoRA_Eval_{i} \times Retain_Eval_{i}$$
 (6)

As for the definition, a KV cache or LoRA with higher *Eval*_i has more benefits to be stored in the HBM, and in other words, meaning it incurs higher costs if it is swapped-in HBM 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 HBM. Then, we can use these relationships to decide their swap-in and swap-out orders when the HBM is full or idle (Section 5.3).

5.3 Workflow of the Cache Swapper

Based on our cost model in Equation 6, Figure 10 shows the operation workflow of the cache swapper under the cooperation with the cache manager.

After each monitor interval of 100ms, the cache manager first calculates the HBM usage based on the storage state of the usage dependency tree. We set HBM usage upper and lower thresholds (95% and 70% in our evaluations) to determine whether HBM is busy or idle, following existing cache management works [41,54]. The upper threshold is set below 100% to leave HBM space for KV caches generated by running queries. Moreover, if we just set the upper threshold and directly swap-in/out according to it, HBM usage may be frequently higher/lower than it in a short time, leading to Ping-Pong swappings, and thus we set the lower threshold to address it. If the HBM usage is larger than the upper threshold, the cache manager will send the swap-out instruction to the cache swapper. Moreover, according to Section 4.2, the

leaf nodes in the HBM will be sent as candidate nodes to the cache swapper. Similarly, if the HBM usage is smaller than the lower threshold, the swap-in instruction along with the root nodes of each path in the main memory will be sent.

After receiving the candidate nodes from the cache manager, the cache swapper then assesses their benefits to inference performance based on the cost model in Equation 6. If swap-in is currently required, the cache manager sorts the candidate nodes with the descending order of their $Eval_i$. Otherwise, it sorts them with the increasing order for the swap-out. Following the greedy algorithm, the cache manager continuously swap-in or swap-out the nodes one by one according to the sorting until the HBM is at a balanced status.

6 Evaluation of FASTLIBRA

In this section, we first show the serving performance of FASTLIBRA under various Multi-LoRA application scenarios. Then, we dive into the reasons behind the performance gains achieved by FASTLIBRA and the effectiveness of each module.

6.1 Evaluation Setup

Table 1 has shown our experimental platform. We use Llama-7B/13B/34B as the base model for our evaluations. Based on the parameter size, we use 1, 2, and 4 NPU cards to deploy the Llama-7B, Llama-13B, and Llama-34B, respectively. Following previous work [37,43], we construct LoRAs based on the model parameters and set the rank of them to 32 and 64 randomly. We construct various numbers (i.e., 20,50, and 100) of LoRAs for each model, and the parameters of the LoRAs are randomly generated using a normal distribution.

We use Multi-LoRA inference systems vLLM [22] and S-LoRA [37] as baselines. vLLM partitions HBM and allocates static HBM space for LoRAs and KV caches, and uses the prefix-caching to reuse history KV caches. It uses the LRU policy to directly discard the KV caches or LoRAs when HBM is full. We adapt a swap-out policy based on LRU for vLLM to offload its history KV caches and LoRAs in the main memory when the HBM is full. Moreover, S-LoRA utilizes a unified caching pool for LoRAs and KV caches, but it does not reuse history KV caches and discards them after the query finishes. It swaps-in the required LoRAs on-demand and swap-out them when no queries use them.

Following prior works [43,54], we utilize the TTFT, TPOT, and peak throughput as the metrics for Multi-LoRA serving. The peak throughput is determined as the supported maximum number of queries per second when the TTFT is below 500ms.

6.2 Application Scenarios

We construct three commonly-used LLM inference application scenarios based on real-world traces.

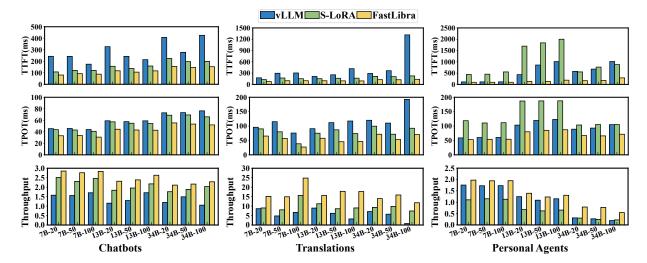


Figure 11: The average TTFT, TPOT, and supported peak throughput of FASTLIBRA, vLLM, and S-LoRA in various scenarios. The X-axis represents the model size and LoRA number, e.g., 7B-20 represents Llama-7B with the LoRA number of 20.

Chatbots. In each round of dialogue, chatbots use all the user's previous history to generate new responses. Online services often allow users to choose specific dialogue scenarios, such as business analysis [40], and use Multi-LoRA inference to improve efficiency. We construct queries based on the LMSYS-33k dataset [52], which contains 33,000 dialogues with real human preferences. Each sample includes the model name, the dialogue text, and the timestamp. Based on the model name, we generate the target LoRA of each query and maintain the original query distribution for different models. Moreover, we proportionally scale the LMSYS-33k dataset to achieve different average query sending rates while preserving the original pattern, similar to previous work [37, 43].

Multi-language Translations. This kind of service use Multi-LoRAs to dynamically select and apply the best model to enhance translation results [55]. We construct queries based on the OPUS-100 dataset [47], which contains 55 million sentence pairs across 100 languages. We make each language translation pair correspond to a specific LoRA, e.g., from French to English. Since the OPUS-100 dataset lacks timestamps, we sample query arrival patterns from the Microsoft Azure function trace (MAFT) [35, 50], following previous work [21,43]. We sort this trace's functions by invoking frequency, select the top-n types of queries, and map them to the n LoRAs to maintain query distribution.

Personal Agents. LLMs are widely applied in personal agents to provide customized support, such as mobile smart assistants and home assistants, with Multi-LoRA commonly applied for this multi-task serving [26, 51]. We utilize the Google Taskmaster [5] to construct queries, which are designed to train and evaluate task-oriented dialogue systems. It contains multi-turn dialogues with complex context management and information exchange, reflecting real-life interactions with assistants. We apply the same sampling method based on MAFT as in the translation scenario.

To adapt different used LoRA numbers (n) to the above three scenarios, we randomly choose the query patterns from n models, translation pairs, or task scenes in the corresponding dataset and map them to n LoRAs, respectively.

6.3 Latency and Peak Throughput

We first evaluate FASTLIBRA on the inference latency and the peak throughput in the three application scenarios. For each 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 peak throughput of FASTLIBRA, then we collect the average TTFT and TPOT of them. Figure 11 shows the average TTFT, TPOT, and peak supported throughput of each model with each LoRA number under FASTLIBRA, vLLM, and S-LoRA.

As observed, FASTLIBRA reduces the TTFT and TPOT, as well as improves the peak throughput in all the test cases. The average reduction of TTFT and TPOT is 60.3% and 33.9% compared to vLLM, and 50.1% and 28.6% compared to S-LoRA. The average peak throughput of FASTLIBRA is 1.7X and 1.6X of vLLM and S-LoRA, respectively. The performance increase of FASTLIBRA originates from maintaining the usage dependencies between LoRAs and KV caches and retaining the most beneficial LoRAs and KV caches in HBM to eliminate the cold-start overhead. The decrease of TPOT of FASTLIBRA is smaller than the TTFT compared to baselines because the cold-start overhead mainly impacts more on the prefill stage of LLM inference, which directly leads to increased TTFT.

Compared to vLLM, FASTLIBRA decreases more TTFT (average 68.9%) in the translation scenario than in other scenarios (average 52.5%). This is because the distribution of LoRAs in this scenario varies more with the OPUS-100 and MAFT datasets. vLLM's static HBM partition results

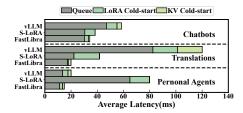


Figure 12: The breakdown of the average queue, LoRA coldstart, and KV cold-start latency in TTFT.

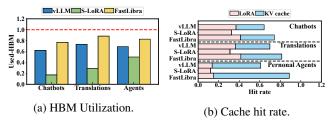


Figure 13: The average HBM usage and cache hit rate of FASTLIBRA and baselines of different scenarios.

in poorer cache management, while FASTLIBRA maintains consistent performance. Compared to S-LoRA, FASTLIBRA achieves the best TTFT reduction (average 81.1%) in the personal agent than others (average 34.6%). This is because this scenario has the longest average conversation length, and S-LoRA's drawback of not retaining history KVs is signified. Similarly, while S-LoRA outperforms vLLM in chatbot and translation scenarios due to larger LoRA distribution changes, it struggles in personal agents due to the long conversation.

6.4 Diving into the High Serving Performance

In this section, we show the breakdown of TTFT, the HBM utilization, and the cache hit rate of FASTLIBRA and baselines, to dive into the reasons for FASTLIBRA's high serving performance.

Figure 12 shows the breakdown of the average queue, LoRA cold-start, and KV cold-start latency in TTFT in different scenarios. We can observe that FASTLIBRA achieves the lowest queue, LoRA cold-start, and KV cold-start latency in all scenarios. This means that FASTLIBRA has the highest HBM utilization efficiency.

For in-depth analysis, we sample the average HBM utilization of FASTLIBRA and baselines across different scenarios, shown in Figure 13a. FASTLIBRA improves HBM 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. In contrast, S-LoRA wastes HBM by not retaining history KV caches, while vLLM's static HBM partition makes the HBM for LoRAs or KVs under-utilized under dynamic loads. These factors also contribute to lower queue and cold-start latency for FASTLIBRA, as shown in Figure 12.

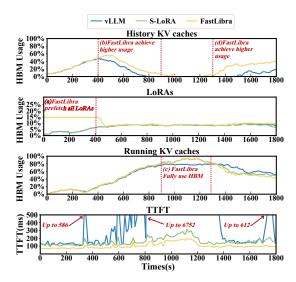


Figure 14: The HBM allocation over time of FASTLIBRA and baselines in different application scenarios.

We also compare the average KV cache and LoRA hit rates of FASTLIBRA and baselines across different scenarios, as shown in Figure 13b. FASTLIBRA increases the cache hit rate by 1.3X and 3.2X compared to vLLM and S-LoRA, respectively. This is because FASTLIBRA maintains the usage dependencies between LoRAs and KV caches to eliminate invalid KV caches which enhances the HBM utilization efficiency. Its efficient swapping strategy also prefetches appropriate KV caches and LoRAs into HBM. S-LoRA has the lowest hit rate because it does not reuse history KV caches. As a result, FASTLIBRA achieves lower queue and cold-start latency for both LoRA and KV caches in Figure 12.

6.5 Investigating HBM Allocation Over Time

In this subsection, we compare HBM allocation between FASTLIBRA and baselines to show the effectiveness of FASTLIBRA's cache management. We take the example of using Llama-13B model, LoAR number of 100, and average sending rate of 1.6 for the chatbot scenario. Other scenarios have similar results. Figure 14 shows the HBM allocations for history KV caches, LoRAs, and running KV caches under FASTLIBRA and baselines.

From 0s to 400s shown in (a), FASTLIBRA proactively fetches all LoRAs into HBM based on the cost model to eliminate the cold-start overhead of LoRAs under low HBM pressure. In contrast, vLLM and S-LoRA load LoRAs ondemand, leading to higher TTFT in this period. From 400s to 900s shown in (b), as the query sending rate increases, FASTLIBRA swaps out some LoRAs and retains the most history KV caches in HBM due to the unified caching pool. In contrast, vLLM's static HBM partition retains fewer history KVs while S-LoRA directly discards them, leading to poorer KV cache reuse and higher TTFT. Moreover, history KVs

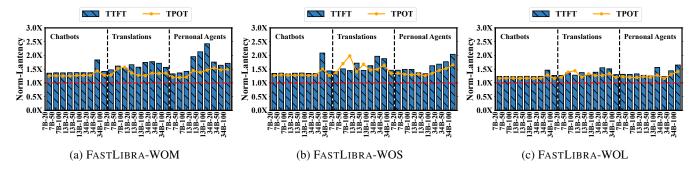


Figure 15: The TTFT and TPOT of the variants of FASTLIBRA in different application scenarios, respectively.

gradually decrease in this period as they are swapped out to free HBM for running KVs when the sending rate rises.

From 900s to 1300s in (c), FASTLIBRA swaps-out all history KV caches to free up HBM for running KV caches of the current inference. By contrast, the static HBM partition of vLLM results in the KV cache memory pool being exhausted to its maximum capacity (80%), leading to queuing and the rapid growth of TTFT. At last, from 1300s to 1800s in (d), FASTLIBRA can retain more history KV caches than vLLM and S-LoRA, leading to a higher HBM usage. During this period, vLLM has higher running KV caches than S-LoRA and FASTLIBRA because of the previous query queuing.

6.6 Effectiveness of the Cache Manager

In this subsection, we show the performance of FASTLIBRA-WOM, a variant of FASTLIBRA that does not maintain usage dependencies between LoRAs and KV caches with the cache manager. The FASTLIBRA-WOM still uses the cache swapper to swap-in/out LoRAs or KVs in the unified caching pool.

Figure 15a shows the TTFT and TPOT of FASTLIBRA-WOM normalized to FASTLIBRA. As observed, the TTFT and TPOT of FASTLIBRA-WOM are higher than FASTLIBRA in all cases, with an average increase of 1.27X and 1.18X, respectively. We also sample the history KV caches during the inference and find FASTLIBRA-WOM suffers from an average of 48.6% invalid KV caches. Moreover, the peak supported throughput of FASTLIBRA-WOM is decreased by 19.8% compared to FASTLIBRA.

When ignoring the usage dependencies between LoRAs and KV caches, lots of invalid KV caches occupy the HBM space but cannot be matched due to their front LoRAs are not loaded, leading to low HBM utilization efficiency. In this case, the useful LoRAs or KV caches cannot be loaded, thus leading to low query serving performance.

6.7 Effectiveness of the Cache Swapper

In this subsection, we show the performance of FASTLIBRA-WOS, a variant of FASTLIBRA that uses a simple LRU policy to replace the cost model (Equation 6) in the cache swapper.

The usage dependencies between LoRAs and KV caches are still maintained with the cache manager during inference.

Figure 15b shows the TTFT and TPOT of FASTLIBRA-WOS normalized to FASTLIBRA. We can observe that both the TTFT and TPOT of FASTLIBRA-WOS are increased in all test cases, with an average increase of 1.24X and 1.15X, respectively. Moreover, the supported peak throughput of FASTLIBRA-WOS is also decreased by 17.2%.

Without FASTLIBRA's cost model to access the benefits or harms to TTFT for swap-in/out, inappropriate LoRAs or KV caches will be swapped-in/out when HBM is idle/busy. This results in more cold-start overheads of LoRAs and KVs, thus decreasing the serving performance.

6.8 Effectiveness of the Enough LoRAs

In this subsection, we show the performance of FASTLIBRA-WOL, a variant of FASTLIBRA that ignores the required LoRA quantity of the cache swapper. Thus, when evaluating the benefits of retaining a node in HBM, FASTLIBRA-WOL eliminates the LoRA reward (Equation 4) in the cost model.

Figure 15c shows the TTFT and TPOT of FASTLIBRA-WOL normalized to FASTLIBRA. FASTLIBRA-WOL increases the TTFT and TPOT in all the test cases, with an average increase of 1.13X and 1.11X, respectively. With the FASTLIBRA-WOL, the peak supported throughput is also decreased by 13.1% on average. Compared to FASTLIBRA-WOS, FASTLIBRA-WOL's serving performance is increased as it considers part of our cost model (Equation 5), but still has a gap to the FASTLIBRA.

The insufficient LoRA loading in some dynamic scenarios can lead to a large number of LoRA cold-starts. As each query inference can only start once the required LoRA is matched in HBM, it can leads to the increase of TTFT.

6.9 The Impacts of a Large Number of LoRAs

In this subsection, we investigate the effectiveness of FASTLI-BRA when thousands of LoRAs exist, although real-world scenarios always only have tens of LoRAs [3, 51]. we use the Llama-7B under the chatbot scenario as an example. The LoRA number is 1000 or 2000, and we set three types of

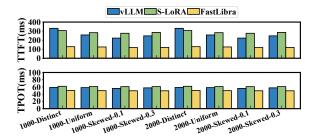


Figure 16: The TTFT and TPOT of FASTLIBRA and baselines with different LoRA numbers and distributions. The x-axis represents the combination of LoRA number and distribution.

LoRA distributions: 1) *Uniform*, where queries have an equal usage probability for each LoRA. 2) *Distinct*, where queries are handled by polling to use a LoRA. 3) *Skewed-x*, where we construct queries using different LoRAs based on Gaussian distribution and set different standard deviations *x*.

Figure 16 shows the TTFT and TPOT of FASTLIBRA, vLLM, and S-LoRA, respectively. We can observe that FASTLIBRA has both the lower TTFT and TPOT in all the test cases, with an average decrease of 55.4% and 16.2%, respectively. Specially, vLLM's and S-LoRA's TTFT varies obviously across different scenarios, while FASTLIBRA can always maintain a stable and low level of TTFT. The above results prove the generality of FASTLIBRA under the large number of LoRAs with different distributions.

6.10 Overhead of FASTLIBRA

The overheads of FASTLIBRA mainly come from three parts: the dependency tree matching and updating in the cache manager, the monitoring of the HBM usage, and the swapping decisions of the cache swapper. For the cache manager, we employ an efficient trie tree for rapid matching and updating. Even if the HBM resources are fully utilized and the size of tree reaches the maximum, the average overhead for matching and updating is less than 0.5ms. Moreover, for monitoring HBM usage and the swapping decisions of the cache swapper, the time overhead for them can be done within 5ms.

The above overheads are all acceptable relative to the entire inference process of each query, which can take seconds or even tens of seconds.

7 Related Work

LLM Fine-tuning. Recent studies have proposed efficient methods for fine-tuning large language models [19, 23, 25], with LoRA adapters being among the most widely used. LoRA achieves fine-tuning with low costs by adding a low-rank branch [19]. Moreover, evolved models like DoRA [27] and AdaLoRA [49] that developed based on LoRA, enhance fine-tuning by introducing flexible updates through weight

decomposition and efficient trimming of insignificant singular values. While improving fine-tuning efficiency, these models share the same features as native LoRA for query inference, i.e., adding branches to the original transformer layers with similar computation and memory patterns, thus FASTLIBRA can adapt to them with minimal modifications.

KV Cache Management. Original LLM inference engines like Orca [45] and FastTransformer [36] directly discard requested KV caches after query processing. To reduce the recomputations, SGLang [53] introduced RadixAttention, which reuses history KV caches using a global prefix tree and LRU strategy. ChunkAttention [53] further improves HBM utilization by sharing KV caches for common prefixes across queries at runtime. For multi-round conversations, Attention-Store [16] and Pensieve [46] maintained a multi-level KV cache system to store and manage all requested history KV caches to eliminate recalculations. Although the above works can reuse history KV caches to improve query inference performance, they neglect to unified manage KV caches along with LoRAs in the HBM under the Multi-LoRA scenario.

Multi-LoRA Serving. Several inference systems have been proposed for Multi-LoRA serving. S-LoRA [37] and Punica [6] separated the base model from the task-specific adapter and dynamically loaded them into HBM. They utilized customized operators to realize that queries using different LoRAs can be batched to improve inference efficiency. dLoRA [43] dynamically switched between merged and unmerged modes to reduce the inference latency, which is orthogonal to the work in this paper. These works did not consider caching history KV caches to avoid recomputation. Moreover, vLLM [41] and SGLang [10] integrated S-LoRA's operators for batched Multi-LoRA inference with static HBM allocation and LRU eviction for LoRAs and KV caches. However, they managed LoRAs and KV caches separately, failing to account for their usage dependencies and balance the HBM usage, resulting in poor Multi-LoRA serving performance.

8 Conclusion

In this paper, we propose FASTLIBRA to optimize the caching of LoRAs and KV caches to improve the Multi-LoRA serving performance. FASTLIBRA'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 in the HBM can be eliminated to improve the HBM utilization efficiency. FASTLIBRA's cache swapper periodically determines the swap-in/out of LoRAs and KVs by using a unified cost model which reflects the benefits of swap-in/out LoRA and KVs to the performance of future queries. We have implemented FASTLIBRA on top of vLLM and experimental results show FASTLIBRA reduces the TTFT and TPOT by 63.4% and 40.1% on average, respectively, compared to state-of-the-art works.

References

- [1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.
- [2] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. arXiv preprint arXiv:2312.11514, 2023.
- [3] Apple. Introducing apple's on-device and server foundation models, 2025.
- [4] Abi Aryan, Aakash Kumar Nain, Andrew McMahon, Lucas Augusto Meyer, and Harpreet Singh Sahota. The costly dilemma: generalization, evaluation and costoptimal deployment of large language models. *arXiv* preprint arXiv:2308.08061, 2023.
- [5] Bill Byrne, Karthik Krishnamoorthi, Chinnadhurai Sankar, Arvind Neelakantan, Daniel Duckworth, Semih Yavuz, Ben Goodrich, Amit Dubey, Kyu-Young Kim, and Andy 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.
- [6] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *Proceedings of Machine Learning and Systems*, 6:1–13, 2024.
- [7] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, et al. Chatbot arena: An open platform for evaluating llms by human preference. *arXiv preprint* arXiv:2403.04132, 2024.
- [8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.
- [9] Google Cloud. Introduction to tpus, 2023.
- [10] SGL Community. sglang: A fast serving framework for large language models and vision language models., 2024.
- [11] Wikipedia contributors. High bandwidth memory, 2023.

- [12] Wikipedia contributors. Trie, 2023.
- [13] NVIDIA Corporation. Nvidia a100 tensor core gpu, 2024.
- [14] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.
- [15] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020.
- [16] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Attentionstore: Cost-effective attention reuse across multi-turn conversations in large language model serving. arXiv preprint arXiv:2403.19708, 2024.
- [17] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.
- [18] Google. Bard, 2023.
- [19] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.
- [20] Chengsong Huang, Qian Liu, Bill Yuchen Lin, Tianyu Pang, Chao Du, and Min Lin. Lorahub: Efficient crosstask generalization via dynamic lora composition. arXiv preprint arXiv:2307.13269, 2023.
- [21] Nikoleta Iliakopoulou, Jovan Stojkovic, Chloe Alverti, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Chameleon: Adaptive caching and scheduling for manyadapter llm inference environments. *arXiv preprint arXiv:2411.17741*, 2024.
- [22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with page-dattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.
- [23] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv* preprint arXiv:2104.08691, 2021.
- [24] Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang

- Yuan, and Wei Wang. Caraserve: Cpu-assisted and rank-aware lora serving for generative llm inference. *arXiv* preprint arXiv:2401.11240, 2024.
- [25] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv* preprint *arXiv*:2101.00190, 2021.
- [26] Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, et al. Personal llm agents: Insights and survey about the capability, efficiency and security. arXiv preprint arXiv:2401.05459, 2024.
- [27] Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. Dora: Weight-decomposed low-rank adaptation. arXiv preprint arXiv:2402.09353, 2024.
- [28] Alpaca lora team. Instruct-tune llama on consumer hardware using alpaca-lora, 2023.
- [29] Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Qingwei Lin, Jianguang Lou, Shifeng Chen, Yansong Tang, and Weizhu Chen. Arena learning: Build data flywheel for llms post-training via simulated chatbot arena. *arXiv* preprint arXiv:2407.10627, 2024.
- [30] Ben Mann, N Ryder, M Subbiah, J Kaplan, P Dhariwal, A Neelakantan, P Shyam, G Sastry, A Askell, S Agarwal, et al. Language models are few-shot learners. *arXiv* preprint arXiv:2005.14165, 1, 2020.
- [31] Matias Martinez. The impact of hyperparameters on large language model inference performance: An evaluation of vllm and huggingface pipelines. *arXiv preprint arXiv:2408.01050*, 2024.
- [32] OpenAI. Chatgpt, 2020.
- [33] PyTorch Contributors. torch.stream pytorch 2.0.1 documentation, 2023.
- [34] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Kimi's kvcache-centric architecture for llm serving. *arXiv e-prints*, pages arXiv–2407, 2024.
- [35] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In 2020 USENIX annual technical conference (USENIX ATC 20), pages 205–218, 2020.

- [36] Noam Shazeer. Fast transformer decoding: One writehead is all you need. *arXiv preprint arXiv:1911.02150*, 2019.
- [37] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. Slora: Scalable serving of thousands of lora adapters. *Proceedings of Machine Learning and Systems*, 6:296–311, 2024.
- [38] Ralf C Staudemeyer and Eric Rothstein Morris. Understanding lstm–a tutorial into long short-term memory recurrent neural networks. *arXiv preprint arXiv:1909.09586*, 2019.
- [39] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288, 2023.
- [40] Ming-Feng Tsai, Chuan-Ju Wang, and Po-Chuan Chien. Discovering finance keywords via continuous-space language models. *ACM Transactions on Management Information Systems (TMIS)*, 7(3):1–17, 2016.
- [41] vLLM Community. vllm: A high-throughput and memory-efficient inference and serving engine for llms.
- [42] Zhengbo Wang, Jian Liang, Ran He, Zilei Wang, and Tieniu Tan. Lora-pro: Are low-rank adapters properly optimized? *arXiv preprint arXiv:2407.18242*, 2024.
- [43] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. {dLoRA}: Dynamically orchestrating requests and adapters for {LoRA}{LLM} serving. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 911–927, 2024.
- [44] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving with cached knowledge fusion. arXiv preprint arXiv:2405.16444, 2024.
- [45] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 521–538, 2022.
- [46] Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful large language model serving with pensieve. *arXiv preprint arXiv:2312.05516*, 2023.

- [47] Biao Zhang, Philip Williams, Ivan Titov, and Rico Sennrich. Improving massively multilingual neural machine translation and zero-shot translation. *arXiv preprint arXiv:2004.11867*, 2020.
- [48] Bill Zhang. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 11, 2016.
- [49] Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. Adalora: Adaptive budget allocation for parameter-efficient fine-tuning. *arXiv* preprint arXiv:2303.10512, 2023.
- [50] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 724–739, 2021.
- [51] Justin Zhao, Timothy Wang, Wael Abid, Geoffrey Angus, Arnav Garg, Jeffery Kinnison, Alex Sherstinsky, Piero Molino, Travis Addair, and Devvret Rishi. Lora land: 310 fine-tuned llms that rival gpt-4, a technical report. arXiv preprint arXiv:2405.00732, 2024.
- [52] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-asa-judge with mt-bench and chatbot arena. *Advances* in Neural Information Processing Systems, 36:46595– 46623, 2023.
- [53] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *arXiv e-prints*, pages arXiv–2312, 2023.
- [54] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Dist-serve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.
- [55] Wenhao Zhu, Hongyi Liu, Qingxiu Dong, Jingjing Xu, Shujian Huang, Lingpeng Kong, Jiajun Chen, and Lei Li. Multilingual machine translation with large language models: Empirical results and analysis. *arXiv preprint arXiv*:2304.04675, 2023.