HyperInfer: Accelerating Large Language Model Inference via Importance-Informed Prefix-KV Caching and Prefetching

Ruidong Zhang, Weijian Chen, Ping Chen,
Shuibing He, Wenjie Liang, Rongchen Liu, Siling Yang, Xuechen Zhang

***Abstract*—Modern advanced large language model (LLM) applications often prepend long contexts before user queries to improve model output quality. These contexts frequently repeat, either partially or fully, across multiple queries. Existing systems typically store and reuse the keys and values of these contexts (referred to as prefix KVs) to reduce redundant computation and time to first token (TTFT). When prefix KVs need to be stored on disks due to insufficient CPU memory, reusing them does not always reduce TTFT, as disk I/O latency is high. In this paper, we present HyperInfer, an importance-informed multi-tier prefix KV caching and prefetching system designed to reduce TTFT in LLM inference. HyperInfer first employs an I/O-efficient algorithm to identify and load only the most important KVs, thereby minimizing I/O overhead. Then, to further hide loading latency, it leverages the inter-layer similarity of token importance to speculatively prefetch the next layer's critical KVs during the current layer's computation. Finally, HyperInfer optimizes prefix-KV storage and cache utilization via importance-informed KV placement, further reducing TTFT in end-to-end inference. Our experimental results show that HyperInfer can reduce TTFT by up to 1.75× compared to state-of-the-art systems, while maintaining comparable inference accuracy. The code is available at https://github.com/ISCS-ZJU/HyperInfer.**

***Index Terms*—Prefetching, KV Cache, LLM Inference**

I. Introduction

GENERATIVE large language models (LLMs), such as ChatGPT and GPT-4, are increasingly utilized in applications like chatbots, document summarization, and translation [12], [17], [31] due to their powerful understanding and generation capabilities. To generate more relevant and high-quality responses, these applications often prepend context-rich prefixes to user queries before sending the entire requests to the model for inference. These prefixes may include the latest news retrieved from web searches relevant to the queries [19], historical conversation with the user [8], and examples of question-answer pairs [25] to guide the model's output format.

This work was supported by the National Science Foundation of China (U25A20423), Major Projects of Zhejiang Province (LD24F020012), Hangzhou Chengxi Sci-Tech Innovation Corridor Special Development Fund (K20241957), Key Research Project of Zhejiang Lab (2025SSYS0005), and the Science and Technology Program of Zhejiang Province (2026SDXT012).

R. Zhang, W. Chen, P. Chen, S. He, W Liang, R. Liu and S. Yang are with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310027, China, also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou 310027, China, and also with Zhejiang Key Laboratory of Big Data Intelligent Computing, Hangzhou 310027, China E-mail: {zhangruidong, weijianchen, zjuchenping, heshuibing, liangwenjie, rcliu, slingzjunet}@zju.edu.cn.

X. Zhang is with the Department of Information Technology, Kennesaw State University, Marietta Campus. E-mail: xzhang54@kennesaw.edu

Ping Chen and Shuibing He are the corresponding authors.

While longer requests enhance the quality of the model's output, they also significantly increase prefill latency, particularly the time to first token (TTFT). This occurs because the computational complexity of processing a request grows super-linearly with its length [34]. Such an increase can negatively impact user experience, particularly in real-time applications like chatbots where a quick TTFT is essential.

Existing systems have observed that many requests share identical initial tokens, known as prefixes. Storing and reusing keys (K) and values (V) of these common prefixes can decrease TTFT by eliminating redundant computations. However, conventional systems only cache these prefix KVs in GPU or CPU memory [10], [15], [38], [44], which can become insufficient for lengthy sequences or large batches, thus restricting TTFT reduction. AttentionStore [8] expands KV storage to local disks and pre-loads them into CPU memory based on the scheduler's predictions for future requests. Yet, disk I/O bandwidth limitations can pose a bottleneck, especially under high request volumes or in preemptive scheduling environments where anticipating future requests is difficult. Our study shows that the I/O latency from SSD to GPU can be rarely hidden by query computation and accounts for 51%-98% of the total TTFT, as shown in §II-C.

Meanwhile, recent research indicates that some tokens' KVs are more critical to the quality of model output than others, and discarding less important KVs can still result in comparable LLM output quality [22], [43]. Based on this insight, we propose HyperInfer, an importance-informed multi-tier prefix KV caching and prefetching system, which integrates local disks for massive storage and employs CPU and GPU memory as high-speed caches, to reduce LLM inference latency while maintaining comparable output quality. The core idea is to minimize I/O overhead by selectively loading only the important prefix tokens' KVs and to hide I/O latency by prefetching the next layer's important KVs concurrently with the current layer's computation. However, building such an efficient system involves three challenges.

First, the important tokens in a shared prefix can vary among multiple queries, incurring substantial I/O overhead to identify important prefix KV for each query. Existing systems need to load all prefix keys into GPU memory to calculate attention scores with the query, which determines the importance of each token's KVs. Loading all keys from disk storage significantly impacts I/O efficiency, impeding TTFT reduction. A smarter method to identify key tokens with minimal I/O overhead is essential.

Second, a natural strategy to reduce I/O latency is to prefetch the KV data for layer $i+1$ during layer i's computation. However, identifying important tokens for layer $i+1$

relies on the computation results of layer i, causing a read-after-compute dependency. Consequently, we cannot load the important KVs for layer $i+1$ concurrently with layer i's computation. We need a method to break this dependency, allowing layer i's computation to overlap with the loading of layer $i+1$'s important KVs, thereby hiding I/O latency.

Third, existing systems often consolidate consecutive KVs into large objects (e.g., chunks), to optimize disk I/Os and PCIe transfer bandwidths. However, our system's selective retrieval of important KVs makes these methods less efficient for two key reasons. (1) Accessing important KVs also loads irrelevant KVs from the same chunk, diminishing disk read performance and filling cache with unnecessary data. (2) Cache management, typically based on chunk access patterns [15], [37], [44], overlooks the significance of individual prefix KVs within each chunk. This can result in crucial KV-rich chunks being stored on slower storage, reducing cache hit ratios.

To address the first challenge, we study the important token distributions among multiple heads of each layer in the LLM. We find that important token indices are highly similar across heads within the same layer. Based on this insight, we propose a similarity-guided important token identification method that only loads the keys from a subset of heads to identify crucial KVs within the entire prefix.

To address the second challenge, we exploit the high similarity of important token index sets between adjacent layers and introduce the layer-aware KV prefetching mechanism, which uses the current layer's importance set to predict the next layer's important tokens. This allows the system to prefetch the next layer's crucial KVs during the current layer's computation, hiding I/O latency and reducing TTFT. A lightweight verification and recovery strategy handles mispredictions to guarantee accuracy with minimal overhead.

To address the third challenge, we propose importance-informed KV placement methods. We employ a KV reordering technique to reorder and repack the prefix KVs stored on disk, thereby increasing the density of important KVs in the chunks and improving the effective disk read bandwidth. Additionally, we introduce a new score-based cache management policy considering token's importance to further enhance cache hit ratios. We implement HyperInfer and evaluate it on three OPT models (6.7B to 30B) across four latency workloads, with additional generation-quality evaluation on Qwen3-8B over two document-grounded QA datasets. Our experimental results show that, compared with a state-of-the-art prefix KV storage system, HyperInfer reduces critical-path I/O time by up to 4.4× and TTFT by up to 1.75×, while maintaining an inference accuracy drop of less than 0.2%.

The main contributions of this paper are as follows:

- We present HyperInfer, the first importance-informed prefix KV caching and prefetching system that integrates three storage tiers: GPU memory, CPU memory and disk.
- We introduce a similarity-guided important token identification method for efficiently locating important KVs, and complement it with importance-informed KV placement, which incorporates KV reordering and a new score-based cache management policy to further minimize data movement from slower storage media.
- We propose the layer-aware KV prefetching mechanism, which exploits the similarity of important tokens across adjacent layers. This allows the system to prefetch the next layer's KVs, decoupling loading of important KVs from token identification and fully overlapping I/O with computation.
- We implement HyperInfer and our evaluation demonstrates that it reduces TTFT by up to 1.75× compared to the state-of-the-art prefix KV storage systems while maintaining comparable inference accuracy.

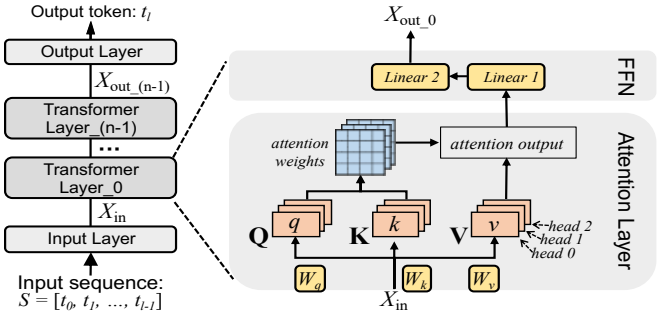


Fig. 1: The LLM model structure.

II. Background and Motivation

A. Architecture and Workflow of Large Language Models

Model inference. A generative LLM consists of an input layer, a stack of transformer layers, and an output layer (Figure 1). Given an input sequence $S = [t_0, t_1, \ldots, t_{l-1}]$ of l tokens, the input layer encodes S, the transformer layers process it sequentially, and the output layer generates the next token t_l. The new token is appended to S and reprocessed to generate subsequent tokens. Generating the first token is the *prefill* phase; generating the rest is the *decoding* phase.

Transformer layer computation. Each transformer layer mainly consists of an attention layer and a feed-forward network (FFN), as shown in Figure 1. During the prefill phase, the input tensor X_{in} is projected by the weight matrices W_q, W_k, and W_v to produce the query (Q), key (K), and value (V) tensors. Each tensor consists of multiple attention heads (e.g., three in Figure 1), where each head corresponds to a 2D tensor denoted as q, k, and v, respectively. The attention weights, computed from Q and K, represent token relevance and are applied to V to obtain the attention output. This output then passes through a two-layer FFN to produce X_{out}, which has the same shape as X_{in}.

B. Impact of Long Contexts and Prefix Reuse

Long TTFT due to the use of context-rich prefixes. Directly using large models for inference may yield suboptimal results [42]. Applications such as RAG, tool-augmented agents, and multi-turn dialogue often prepend retrieved documents, tool descriptions, or conversation history as reusable prefixes [2], [8], [19], [25].

Figures 2(a) and 2(b) show that although context-rich prefixes enhance response quality, they substantially increase the

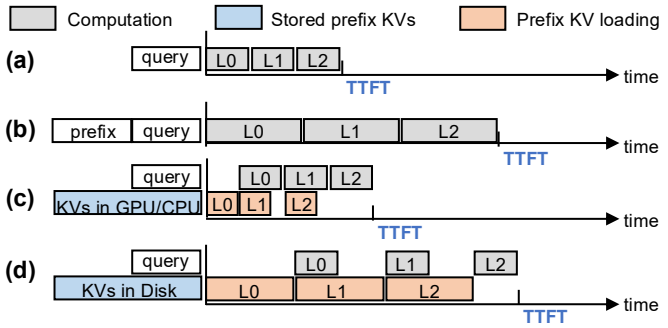


Fig. 2: TTFT under different cases. We assume an LLM with three Transformer layers, denoted as "Lx".

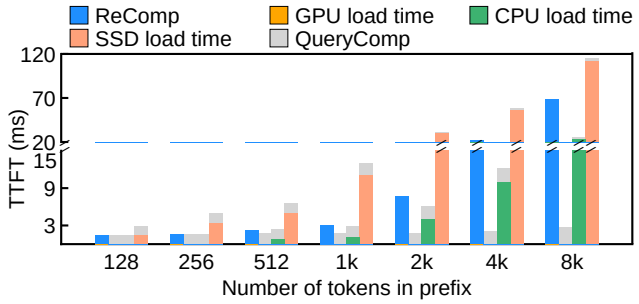


Fig. 3: TTFT breakdown. "ReComp" refers to not reusing the prefix KV. "QueryComp" denotes the remaining computation after loading the prefix KV.

delay before the first token is produced, known as the time to first token (TTFT). For example, Chameleon adds more than 2,600 tokens before each user query [24]. Given that an average real-world query contains approximately 750 tokens [33], this quadruples the input length and increases TTFT by up to nine times for the OPT-30B model. Such latency severely affects user experience in TTFT-sensitive applications such as real-time chatbots and also reduces system throughput, leading to higher operational costs.

Mitigating latency via prefix KV reuse. Researchers have observed that these prefixes are often partially or completely shared across different requests [8], [10], [15], [21], [38], [44]. For example, similar queries might retrieve partially or entirely the same related documents using RAG; the same GPT plugin can be used multiple times, resulting in identical system prompts across requests. Recomputing the K and V tensors in Figure 1 for the same prefix leads to wasted computational resources and increased TTFT. To optimize TTFT, existing systems store and reuse the K and V tensors of these shared prefixes (referred to as *prefix KV cache* or simply *prefix KV*). Note that the Q tensor of the prefix is not stored, as it is not needed for subsequent computations [8]. When a new request with a repeated prefix arrives, the system asynchronously preloads its prefix KVs into GPU memory, thereby reducing TTFT during the prefill phase of the new request.

C. Limitations of Existing Prefix KV Management Systems

An effective prefix KV management system needs to satisfy two key requirements. First, it should provide sufficient storage capacity to hold a large number of prefix KVs. Second, it needs to ensure low latency for retrieving prefix KVs, as excessive loading delay can become a bottleneck in LLM inference. However, existing systems fail to satisfy both requirements simultaneously. They typically either achieve low latency with limited capacity, or provide sufficient capacity but suffer from high I/O latency.

Memory-based systems face a capacity wall. Systems such as PromptCache [10], SGLang [44], RAGCache [15], and ChunkAttention [38] store prefix KVs solely in GPU and/or CPU memory to ensure low retrieval latency, as illustrated in Figure 2(c). This design achieves low TTFT by leveraging the high bandwidth of GPU and CPU memory for rapid data access. However, GPU and CPU memory are inherently capacity-limited, causing these systems to quickly exhaust available storage in RAG workloads.

Disk-based systems are bottlenecked by limited bandwidth. AttentionStore [8] addresses capacity constraints by incorporating local disks into the storage hierarchy. To mitigate the high I/O latency of disk retrieval, it employs layer-wise prefetching to overlap data loading with computation, as depicted in Figure 2(d). Nevertheless, the substantial data transfer overhead often cannot be fully hidden by computation, particularly under heavy request loads or preemptive scheduling. Figure 3 shows that loading prefix KVs from GPU or CPU results in shorter TTFT compared to recomputation, but loading from SSD leads to longer TTFT. This is due to the I/O latency from SSD to GPU, which is rarely hidden by query computation and accounts for 51%-98% of the total TTFT. Consequently, prefix KV loading has become a new bottleneck in model inference, particularly for longer prefixes.

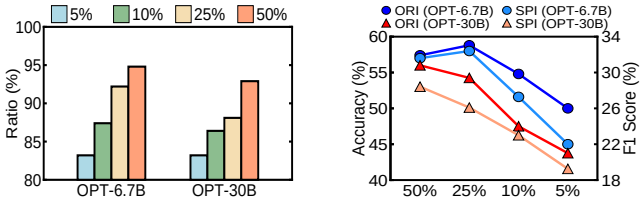
D. Not All KVs Are Equally Important

Recent research [18], [22], [32], [43] indicates that not all tokens' KVs are equally important for maintaining LLM inference accuracy. These methods generate and store the full set of KVs during the prefill phase, then identify and discard the less important tokens' KVs during decoding by analyzing the attention weights. This approach reduces the computational load during the decoding phase while maintaining comparable LLM inference accuracy.

Inspired by this, we propose HyperInfer to meet the capacity and low-latency retrieval requirements (§II-C) via importance-informed KV management. By leveraging a multi-tier hierarchy encompassing GPU memory, CPU memory, and disk, HyperInfer ensures sufficient storage capacity for prefix KVs. Crucially, it minimizes TTFT by selectively loading important KVs, prefetching next-layer KVs to overlap I/O with computation, and caching KVs in GPU memory and CPU memory to avoid disk I/O. However, realizing such a system poses three key challenges.

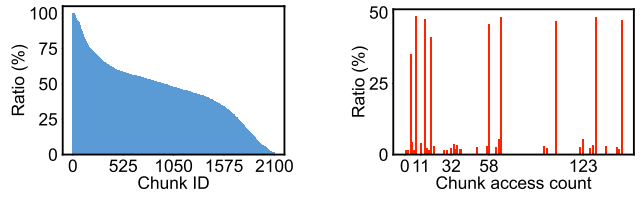
III. Design Challenges of HyperInfer

Challenge 1: Importance identification introduces high I/O overhead for prefix KV loading. Existing methods identify important KVs by loading all prefix keys (K) into GPU memory [18], [22], [32], [43], so reducing only the loading time of values (V) yields limited TTFT gains. A naive solution



(a) Recall Ratio (b) Generation Quality

Fig. 4: The recall ratios and the impacts on model generation quality of the static pre-identification method (SPI) under various important token retention percentages. "ORI" denotes the baseline where the SPI method is not applied.



(a) (b)

Fig. 5: (a) The ratio of important KVs within each chunk. (b) Average ratio of important tokens in all chunks for a given chunk access frequency.

to reduce I/O overhead is to pre-record the important tokens of each prefix based on prior queries and load only their KVs when the same prefix reappears. However, this static approach fails to account for query-dependent token importance. In practice, token relevance within the same prefix can vary significantly across different queries. In RAG, for example, different queries sharing the same document prefix may attend to different segments, causing static selection to miss critical tokens and degrade accuracy. We evaluate this limitation using OPT-6.7B on RTE [9] and OPT-30B on SQuAD [30], measuring accuracy and F1 score [21], respectively. As shown in Figure 4, even with recall ratios above 80%, missing key KVs reduces accuracy by up to 5% on RTE and 3.3% on SQuAD. These results motivate a dynamic approach that identifies important tokens per query while minimizing I/O overhead.

Challenge 2: Read-after-computation dependencies hinder effective prefetching. While prefetching is a standard strategy to hide I/O latency [4], [5], [8], [11], [32], it is inherently conflicted with importance-based selective loading. In our setting, we aim to fetch only the critical KVs to save bandwidth. However, the set of important tokens for layer $i{+}1$ is only determined after completing the computation of layer i. This creates a strict read-after-compute dependency, rendering concurrent prefetching infeasible. Consequently, the system is forced into a serial execution pattern where I/O latency cannot be masked by computation.

History-based prefetching is insufficient because KV importance is query-dependent as evidenced in *Challenge 1*. We quantify this limitation in §VI-D.

Challenge 3: The existing prefix KV storage and caching systems are suboptimal considering token's importance. Existing systems typically store and manage cache by grouping KV pairs from several consecutive or all prefix tokens into chunks [8], [38], [44], which enhances the efficiency of disk reads and PCIe transfers. Each chunk contains a mix of important and unimportant KVs. When retrieving KVs for important tokens, entire chunks are loaded into memory, including the unimportant ones. This practice causes read amplification, diminishing effective bandwidth and filling cache with unnecessary data, which lowers cache hit ratios.

Furthermore, managing chunks in CPU and GPU caches based solely on traditional metrics like recency or frequency can further reduce GPU cache hit ratios and increase PCIe data transfers. This is because these metrics ignore the importance of KVs and the proportion of important KVs in each chunk. As a result, less critical chunks may occupy valuable GPU memory, while more critical ones are relegated to the CPU memory. This misallocation decreases the GPU hit ratio for important KVs and necessitates more data transfers between CPU and GPU memory.

We experimentally illustrate this challenge using OPT-6.7B on RTE. We designate half of the prefix tokens as important and the remaining half as unimportant. Figure 5(a) shows that only 46% of the KVs in the loaded chunks are important on average, leading to 2.2× read amplification. Figure 5(b) shows the ratio of important KVs for chunks with different access frequency. The hotness or coldness of a chunk is not related to the ratio of important KVs it contains. These observations underscore the need for new KV storage and cache management methods to reduce the loading of unimportant KVs from disk and improve cache hit ratios.

IV. HyperInfer Design

In this section, we present HyperInfer, an importance-informed three-tiered prefix KV caching and prefetching system. First, we describe the overall architecture of HyperInfer (§IV-A). Then, we introduce an I/O-efficient technique to identify important tokens (§IV-B). Furthermore, we propose a prefetching technique to mitigate I/O bottlenecks by overlapping computation and I/Os (§IV-C). Finally, we explain how prefix KVs are managed across three storage tiers to further reduce the latency when loading them into GPU memory (§IV-D).

A. Overview

To address the challenges outlined in §III, HyperInfer is designed based on three core principles: (1) employing an I/O-efficient method to identify important KVs, ensuring only essential KVs are loaded during the prefill phase; (2) speculatively loading important KVs for the next layer during the current layer's computation, thereby breaking the read-after-compute dependency and hiding I/O latency; and (3) optimizing tiered storage management to improve cache hit ratios and I/O efficiency.

Figure 6 presents the overall architecture of HyperInfer. In the data plane, all prefix KVs are stored on disks in chunks, with some prefix KVs cached in either CPU memory or GPU memory. The data in the two cache spaces are exclusive to

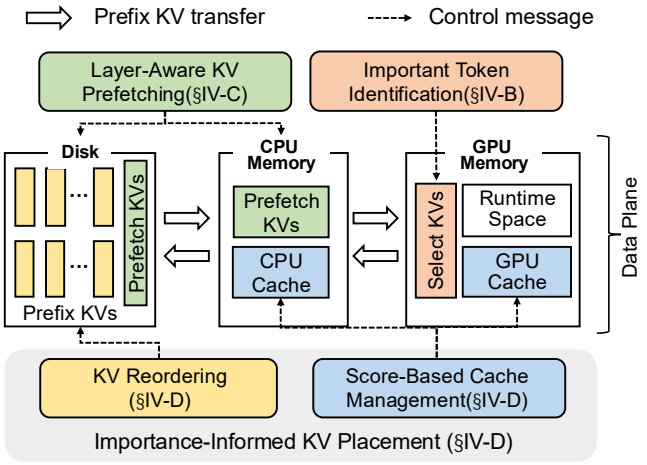
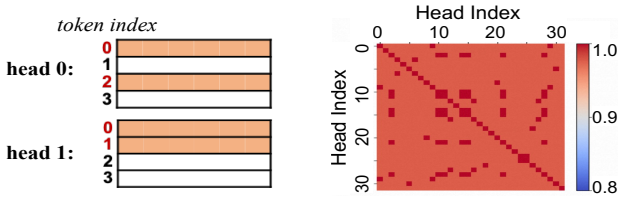


Fig. 6: Overview of the HyperInfer system.

avoid space wastage. The metadata in the CPU memory is organized using a radix tree [44], which facilitates quick searches for the reusable prefix KVs. The runtime space stores model parameters and intermediate data needed for GPU inference. HyperInfer has three control components: *Important Token Identification* (§IV-B), *Layer-Aware KV Prefetching* (§IV-C), and *Importance-Informed KV Placement* (§IV-D). The Important Token Identification module identifies important tokens within a chunk by loading only partial keys rather than all of them, reducing the amount of data loaded from disks. The Layer-Aware KV Prefetching module proactively loads critical KVs for future layers during current computation to hide I/O overhead. The Importance-Informed KV Placement module manages the storage and data movement of prefix KVs across disks and the two cache spaces to maximize cache efficiency.

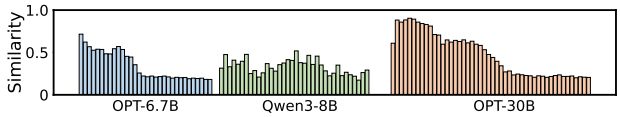
Workflow of HyperInfer. When an inference request arrives, HyperInfer first searches the radix tree to locate reusable prefix KVs [38], [44]. It then processes the prefix layer by layer. Before computing each layer, HyperInfer identifies the current layer's important tokens (§IV-B), verifies the KVs prefetched from the previous layer, and synchronously loads any missing important KVs. During the layer computation, HyperInfer uses the current layer's important-token indices to prefetch important KVs for the next layer (§IV-C). After prefill, newly generated KVs are managed by the KV placement module (§IV-D) for future reuse, while the decoding phase remains unchanged [34].

Importance metric. In this paper, we use the column-wise sum of the attention weight matrix as the token's importance, following the same method in H2O [43]. A higher sum indicates greater token importance. HyperInfer treats the importance metric as a pluggable component. At the system level, its identification (§IV-B), prefetching (§IV-C), and placement (§IV-D) mechanisms depend only on per-layer token importance rankings, rather than on how these rankings are computed. Thus, any importance computation method [18], [22] that produces such rankings can be integrated by replacing the default metric. For GQA models, token scores from query heads sharing the same KV head are summed before selecting that KV head's top-ranked KVs.

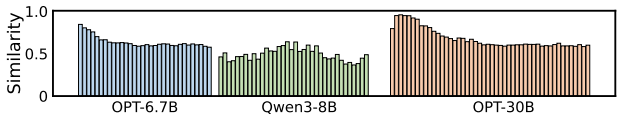


(a) Two Heads of K Tensors (b) Jaccard Scores Between Heads

Fig. 7: Examples of similarities. (a) The orange rows represent the important keys whose token indices are marked in red. (b) Darker squares indicate a higher similarity between the important token index sets of two heads.



(a) select the top 10% most important tokens



(b) select the top 40% most important tokens

Fig. 8: Similarities of important token index sets across all transformer layers in OPT-6.7B, Qwen3-8B (a GQA-based model), and OPT-30B.

B. Similarity-Guided Important Token Identification

Observation I: There is a high similarity in the set of important token indices across different heads within the same layer of an LLM.

As discussed in §II-A, each token has K and V tensors for every head. We observe that important token indices are highly similar across heads within the same layer, as k and v tensors originate from the same large K and V tensors [34], [41]. Thus, a token important in one head is likely important in others. While this cannot be formally proven for all LLMs, its generality and practicality are validated through accuracy evaluations (§VI-B) on four datasets across three models.

To quantify the similarity between important token index sets, we use the Jaccard index [7], defined as $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$, where A and B denote the important token index sets of two heads. The value ranges from 0 (completely different) to 1 (identical). As shown in Figure 7(a), for an input with four tokens where 50% are important, $h_0 = \{0, 2\}$ and $h_1 = \{0, 1\}$, yielding $J(h_0, h_1) = \frac{1}{3}$. A real similarity heatmap from the middle layer of OPT-6.7B (Figure 7(b)) shows high similarity among heads, with average values exceeding 0.95.

Observation II: The similarity of important tokens (represented as important token index sets) exists across different sampling ratios and LLM scales.

To further understand this phenomenon, we analyze the average similarity of important token index sets across different heads for the OPT-6.7B, Qwen3-8B, and OPT-30B models when selecting the top 10% and 40% most important tokens (Figure 8). Each bar denotes the average similarity for one

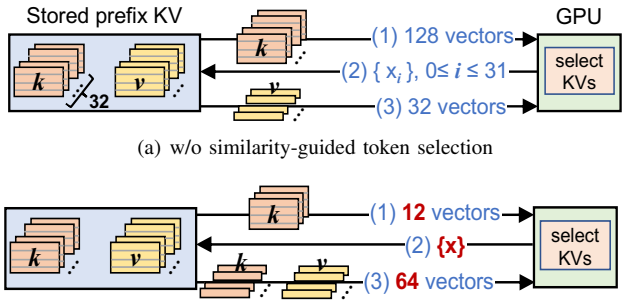


Fig. 9: The process of transformer layer computation with prefix kv. Each k and v tensor has four row vectors because of four tokens in the prefix.

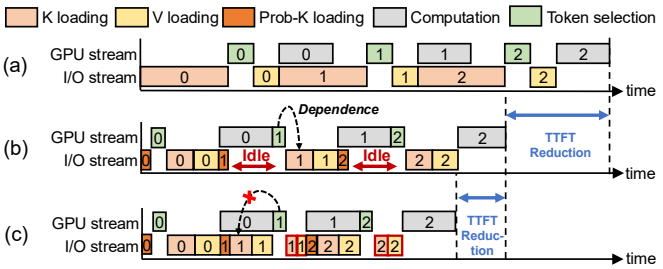


Fig. 10: Timeline breakdown of TTFT. (a) Without similarity-guided important token identification. (b) With similarity-guided important token identification, where black dashed arrows indicate the read-after-compute dependency. (c) With layer-aware KV prefetching, where red-bordered rectangles indicate loading missing KVs. Assume the LLM model consists of three transformer layers. The numbers inside rectangles represent the layer index.

transformer layer, with 32, 36, and 48 layers in the three models, respectively. We have two findings: (1) selecting a larger proportion of important tokens leads to higher similarity; (2) although smaller models and some layers generally show lower similarity, they remain substantially above the random-selection baseline in most cases. (3) the observed similarity consistently holds across both MHA-based models (OPT-6.7B and OPT-30B) and the GQA-based model (Qwen3-8B), suggesting that this property is not tied to a specific attention architecture.

Design. Based on the above observations, we propose the similarity-guided important token identification technique. The core idea is that *owing to the high similarity among attention heads, we can use the important token indices identified from a few selected heads to approximate those of the remaining heads*. For simplicity, we refer to these selected heads as *probe heads*. Since this process involves loading only keys from a subset of the heads instead of all heads, it reduces both I/O data volume and TTFT.

As some layers exhibit less pronounced similarity between important token sets across different heads, applying this technique to these layers may misidentify important tokens in some heads, reducing the model's inference accuracy. To tackle this issue, we introduce a similarity threshold that dynamically determines whether to apply the technique for each transformer layer. The similarity-guided important token identification is enabled only when the measured similarity value from the probe heads is higher than the similarity threshold.

Figure 9 illustrates the process of completing a transformer layer with and without this technique. Assume the prefix contains 4 tokens, with only one being important. Each transformer layer has 32 heads and all the prefix KVs of each head are stored on disks. The number of probe heads is set to three. Without it, all 32 heads must load 128 *key* vectors and later 32 k, v vectors, totaling 160 vectors. With similarity-guided important token identification, only probe-head keys are loaded first; if their similarity exceeds the threshold, the selected token indices are reused for the remaining heads, reducing loaded vectors from 160 to 76 in this example. When the threshold is unmet, the process reverts to the standard mode.

Figure 10 compares the timelines for completing three transformer layers with and without similarity-guided important token identification. Without similarity-guided important token identification, in Figure 10(a), the loading of a large number of keys leads to GPU idle time, prolonging inference process. In contrast, when the technique is enabled in Figure 10(b), and the similarity among probe heads exceeds the threshold, only the keys from the probe heads (in orange) are loaded to identify important tokens. After obtaining the important tokens, the system loads only the crucial values (in pink) and remaining crucial keys from non-probe heads (in yellow), effectively reducing I/O overhead. Additionally, loading only a subset of keys for attention weight calculations reduces the time spent generating important token index sets (in green), leading to a shorter overall TTFT.

Hyperparameter decisions. To make this algorithm practical, we need to determine the number of probe heads and the similarity threshold. Selecting only one probe head to determine the most important token index may introduce bias, affecting model accuracy. Using two probe heads might fail to identify the most important index through voting when disagreements arise. Therefore, we choose to use three probe heads. Increasing the number of probe heads offers minimal improvements in accuracy but increases the keys loading time, thereby extending the TTFT. Additionally, we found that the choice of which three heads to use has no impact on accuracy due to the similarity. Therefore, we simply select the first three heads in each transformer layer as the probe heads to keep the selection process quick. We first compute the similarity of important token sets between each pair of probe heads, and then take the average to measure the overall similarity among the three probe heads. Thus, the similarity is independent of the order of probe heads.

To address this issue, we calculate the expected value based on the proportion of selected important tokens and use it as the similarity threshold. Assume a prefix contains n tokens and k important tokens ($k \leq n$). If we randomly perform this selection twice, we obtain sets A and B. The expected intersection and union are $E(A \cap B) = k^2/n$ and $E(A \cup B) = 2k - k^2/n$, respectively. Thus, $E(\text{Jaccard}(A, B)) = \frac{k/n}{2-(k/n)}$,

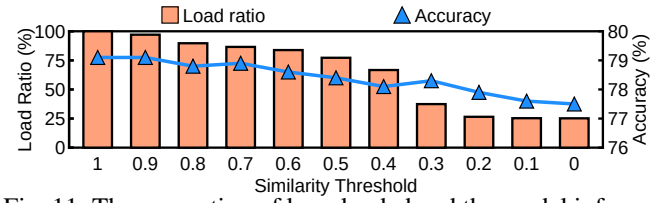


Fig. 11: The proportion of keys loaded and the model inference accuracy under different similarity thresholds.

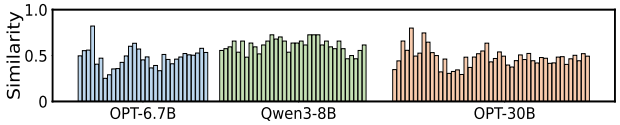


Fig. 12: Similarity of important token index sets between adjacent transformer layers in different models.

denoted as j. We set the threshold $t = j^{\alpha}$, where $\alpha = 0.6$ is empirically chosen to achieve a good balance between model inference accuracy and the amount of keys loaded.

C. Layer-Aware KV Prefetching

Observation III: The sets of important token indices are highly similar between adjacent transformer layers.

To understand the nature of inter-layer consistency in token importance, we examine it from two complementary perspectives. First, we assess the overlap of important token index sets between adjacent layers. For OPT models and Qwen3-8B, we select the top 25% of tokens based on importance in each layer and compute the Jaccard similarity [7] between these sets in neighboring layers. As shown in Figure 12, the similarity consistently falls within a narrow range, e.g., for Qwen3-8B, values span from 0.46 to 0.72 with an average of 0.61, which indicates that adjacent layers largely agree on which tokens are important.

Second, we investigate how relative importance rankings propagate across layers. Within the top 25% important tokens of each layer, we partition positions into finer groups by their percentile rank (e.g., the highest group contains the top 20% most critical tokens in this set). For each group, we measure its recall in the next layer—the fraction of tokens that remain in the top 25%. As shown in Figure 13, tokens with higher importance in the current layer consistently exhibit higher recall, and this trend holds across three models and two datasets. Together, these results show that token importance is not only shared across layers but also largely preserved in rank order, providing a strong foundation for accurate cross-layer prefetching.

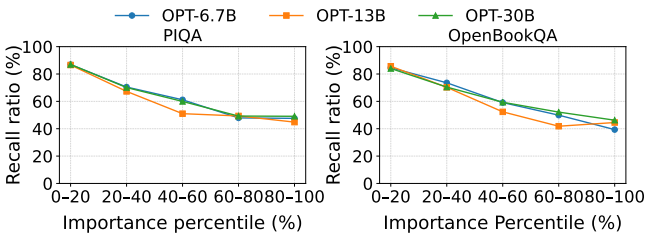


Fig. 13: Recall ratio of next layer important tokens.

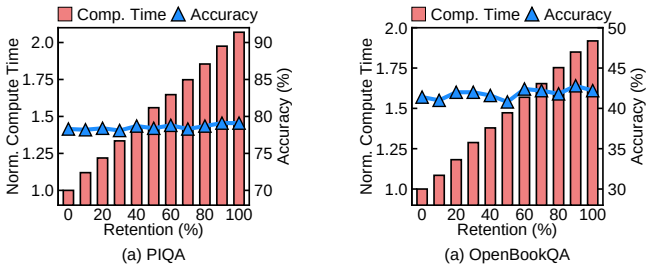


Fig. 14: Impact of retaining unimportant KVs on compute time and model accuracy.

Design. Based on the above observations, we propose the layer-aware KV prefetching mechanism. *The core idea is to use the current layer's important token index sets to approximate those of the next layer, enabling the speculative prefetching of critical KVs prioritized by importance using idle I/O bandwidth during the current layer's computation.* By breaking the read-after-compute dependency, this approach overlaps I/Os with GPU processing, thereby hiding data loading latency and reducing TTFT.

Figure 10 illustrates the impact of this mechanism on inference timelines. As shown in Figure 10(b), without prefetching, the loading of critical KVs for Layer 1 is blocked until token selection completes due to the strict read-after-compute dependency, leaving PCIe bandwidth idle during Layer 0's computation. In contrast, Figure 10(c) demonstrates that by using Layer 0's important token index sets to approximate those of Layer 1, the dependency is broken. Therefore, the idle PCIe bandwidth during Layer 0's computation is utilized to speculatively prefetch Layer 1's important KVs. Upon obtaining the ground-truth importance for Layer 1, only the missing KVs (highlighted in the red boxes) need to be synchronously loaded. Since these missing KVs constitute only a small subset of the total required data, the I/O latency on the critical path is significantly reduced. This pipelined pattern continues for subsequent layers (e.g., Layer 1 prefetching for Layer 2), ensuring continuous I/O hiding throughout the inference process, yielding a shorter overall TTFT.

Implementing this mechanism requires addressing two key issues. First, a robust verification and recovery mechanism is necessitated to identify and load missing important tokens, ensuring correctness without incurring high penalties. This requirement arises because using the current layer's important tokens to approximate those of the next layer is inherently speculative, inevitably leading to the omission of crucial KVs. Second, prefetching I/Os must be carefully scheduled to terminate within the computation window, since any retrieved unimportant KVs after computation result in loading of unused data being exposed on the critical path, which wastes I/O bandwidth and incurs direct latency penalties.

To address the first issue, we implement a *speculate-then-verify* workflow. During the computation of layer i, HyperInfer speculatively loads KVs for layer $i+1$ into a dedicated prefetch buffer sized for important KVs of a single layer, prioritizing tokens with the highest importance. At the start of layer $i+1$, the system performs a lightweight verification step where it executes the similarity-guided identification (§IV-B)

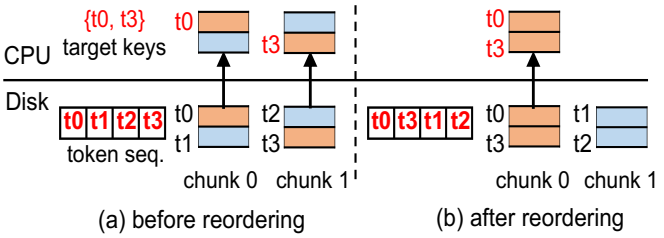


Fig. 15: Comparison of the number of chunks read before and after the token sequence is reordered. The orange (blue) rectangles represent important (unimportant) keys.

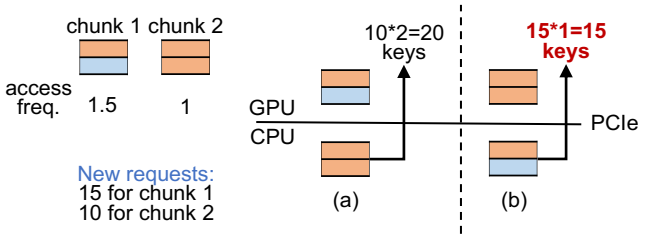


Fig. 16: Comparison of two cache replacement policies.(a) is the frequency-based cache replacement policy and (b) is the score-based cache management policy. The orange (blue) rectangles represent important (unimportant) keys. Only important keys are needed for new requests.

to determine the ground-truth important KVs. HyperInfer then compares the ground-truth with the already prefetched KVs and loads only the missing ones. Simultaneously, we discard any prefetched KVs identified as unimportant, as processing them increases attention latency for negligible accuracy gains.

To justify our discard policy, we analyze the sensitivity of accuracy and latency to the retention rate of unimportant KVs for OPT-30B on PIQA and OpenBookQA. As shown in Figure 14, with the top 25% important tokens fixed and the retention ratio of the remaining unimportant ones varied, we observe that while the normalized computation time scales linearly with the retention rate, the accuracy improvements do not exceed 2%, and in some cases, the accuracy even drops.

To address the second issue of over-prefetching beyond the current layer's compute window, we implement a simple yet effective *reactive truncation* strategy. The prefetching for the next layer starts simultaneously with the computation of the current layer. The CPU continuously submits retrieval requests for KVs while monitoring the GPU status. As soon as the GPU finishes the current layer's computation, the prefetcher immediately stops issuing any new requests. Together, these techniques provide a robust, near-zero-penalty optimization. If prefetching fails, the system falls back to standard demand loading (Figure 10(b)), keeping performance bounded.

D. Importance-Informed KV Placement

1) KV Reordering: We present the KV reordering method to address the unnecessary loading of unimportant KVs during important KV retrieval. KV reordering is performed as an online background task at regular intervals (e.g., every 10 minutes) rather than on the request critical path. It periodically reorganizes and repacks important KVs into denser chunks to improve read efficiency and reduce bandwidth waste. We use a radix tree to record stored prefix tokens [38], [44]. For dynamic prefix updates, when new prefixes or suffixes are inserted into the radix tree, their KVs are first stored in the default contiguous layout and are considered in later reordering rounds. After reordering, HyperInfer atomically updates the token-to-chunk mapping metadata, allowing in-flight requests to continue using the old layout while subsequent requests use the new one.

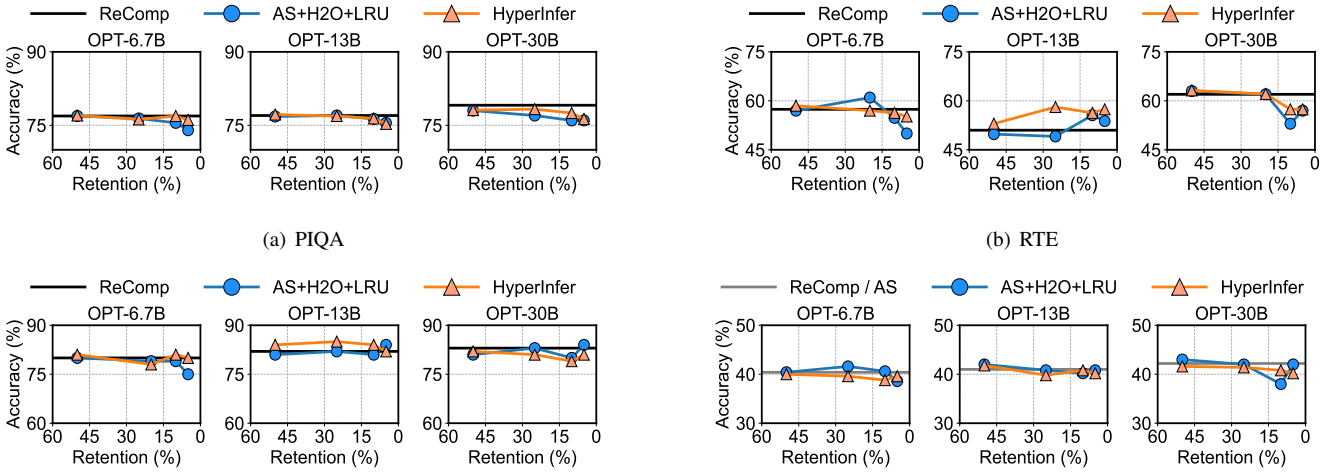
To illustrate, consider a prefix [t0, t1, t2, t3] where the existing system stores keys for two adjacent tokens in one chunk, each containing one important and one unimportant key. Without KV reordering (Figure 15(a)), both chunks must be loaded to retrieve the important keys {t0, t3}, wasting read bandwidth and cache space on unimportant keys. With KV reordering (Figure 15(b)), tokens are reordered by importance, and keys of adjacent reordered tokens ([t0, t3] and [t1, t2]) are packed into one chunk. This enables loading just one chunk to access all important keys, reducing disk read data.

2) Score-Based Cache Management: To cut down on PCIe transfers, after loading a chunk from disk into CPU memory, only the important key and value vectors from that chunk are sent to the GPU memory via PCIe. However, as depicted in Figure 5(b), the presence of important key or value vectors in a chunk does not align with the chunk's access frequency. Consequently, existing systems that base their caching decisions for GPU or CPU memory solely on recency or frequency of chunk access may lower the GPU cache hit ratio for important key-value pairs, thus increasing PCIe traffic.

Score-based cache admission. To enhance cache efficiency, we introduce the importance-aware cache admission policy. This policy assigns a score to each chunk based on its access frequency and the proportion of important keys or values it holds. Chunks with higher scores are preferentially cached in GPU memory, whereas those with lower scores are stored in CPU memory. This approach boosts the GPU cache hit ratio and minimizes data transfers between CPU and GPU. The importance ratio is dynamically calculated as a moving average, updating online after each chunk access.

For instance, suppose key chunk 1 from application A and key chunk 2 from application B are contenders for the limited GPU memory, with application A's request frequency being 1.5 times that of B, making chunk 1 more frequently accessed. However, chunk 1 contains only one important key (50%), while chunk 2 contains two important keys (100%). As depicted in Figure 16(a), traditional systems would cache chunk 1 in the GPU memory due to its higher access frequency, relegating chunk 2 to the CPU. With 15 new requests from A and 10 from B, this approach would necessitate transferring 20 important keys from CPU to GPU. By contrast, factoring in the importance ratio, chunk 1 scores 0.75 ($1.5 \times 50\%$), and chunk 2 scores 1 ($1 \times 100\%$). Thus, our method caches chunk 2 in the GPU memory, as shown in Figure 16(b), cutting the transfer of important keys to just 15.

Dual-cache replacement algorithm. We employ a score-based cache replacement policy to coordinate GPU and CPU caches. Two min-heaps in CPU memory track chunks in each



(a) PIQA (b) RTE

(c) COPA (d) OpenBookQA

Fig. 17: Model generation quality of various systems across four datasets and three models.

cache, with the top entries representing the lowest-scored chunks eligible for eviction. To avoid redundancy, each chunk resides in only one cache, while all replicas are preserved on disk to eliminate I/O latency during eviction.

When a new request arrives, HyperInfer locates chunks containing reusable important tokens. If a chunk is already in the GPU cache, its key/value vectors are used directly and its score is updated. If it resides in the CPU cache, the vectors are transferred to the GPU, and the chunk's score is compared with the lowest in the GPU cache—replacing it only if higher. For chunks on disk, HyperInfer loads them into CPU memory, fetches the required vectors, and evaluates their updated scores to determine whether they should be promoted to GPU memory, remain in CPU cache, or stay on disk.

V. Implementation

We chose to implement HyperInfer on top of FlexGen [32] because its white-box model implementation facilitates the development of our I/O-efficient KV identification method. Specifically, we modified the *mha* function for prefix reuse and used values in *attn_weight* to assess KV importance. For layer-aware KV prefetching, we implemented the *prefetch* function to execute asynchronously from the main computation flow. For KV reordering, we implemented the *PrefixKVLayer* class to store reordered KVs and mapping lists per layer. For cache management, we developed the *TokenCache* class with our score-based policy.

VI. Evaluation

A. Experimental Setup

Models and system configuration. We conduct tests using four open-source LLMs, including three MHA models (OPT-6.7B, OPT-13B, and OPT-30B) and one GQA model (Qwen3-8B). Our experiments are performed on a server with 2 × AMD EPYC 7763 CPUs (64 cores), 128 GB DRAM, one NVIDIA A100 GPU with 80GB HBM, and one 2TB Intel SSD whose measured read throughput is around 5GB/s. The GPU and CPU are connected via PCIe 4.0 × 16.

Datasets and metrics. We select four representative datasets from the standard LM-Evaluation-Harness benchmark [9]: PIQA, RTE, COPA, and OpenBookQA. These datasets are designed for few-shot tasks and structured as multiple-choice questions to evaluate the capabilities of large models in commonsense reasoning, logical inference, causal reasoning, and science question answering, respectively. Due to the lack of open-source, real-world datasets for prefix reuse, we adopt a similar approach to previous work [18] by prepending two to ten few-shot examples as system prompts before each query. These system prompts are shared across different queries, with reuse frequency following a normal distribution.

We measure model generation quality using accuracy, as in prior works [18], [43], and vary the prefix KV retention ratios from 50% to 5% to observe accuracy changes. We further evaluate generation-oriented quality on two document-grounded QA datasets, SQuAD [30] and HotpotQA [36], using Qwen3-8B, and report F1, exact match (EM), and ROUGE-L. Additionally, to test TTFT with long prefixes as in [21] and to prevent runtime out-of-memory errors, we extend the prefixes to a maximum length of 4K for OPT-30B and 10K for the other OPT models. The average number of tokens in the request prefixes across the four datasets ranges from 4.8k to 5.7k.

Baseline systems. We compare HyperInfer with six baselines. (1) *ReComp* [34]: It recomputes all prefix KVs for each request without storing or reusing them. (2) *AS-like* [8]: AttentionStore (AS) asynchronously stores and loads all shared prefix KVs. Since it is not open-source, we reimplement it to the best of our ability based on the paper. To ensure a fair comparison, we additionally add the GPU cache with LRU for prefix KV storage. Besides, we disable its scheduler-aware optimizations to make it more suitable for general scenarios, such as preemptive scheduling environments. (3) *AS+H2O+LRU*: We combine AS-like with one of the state-of-the-art important KV selection systems, H2O [43]. Different from the AS-like, it loads only important values rather than the full values of shared prefixes. (4) *AS+H2O+LFU*: It is similar to (3), but

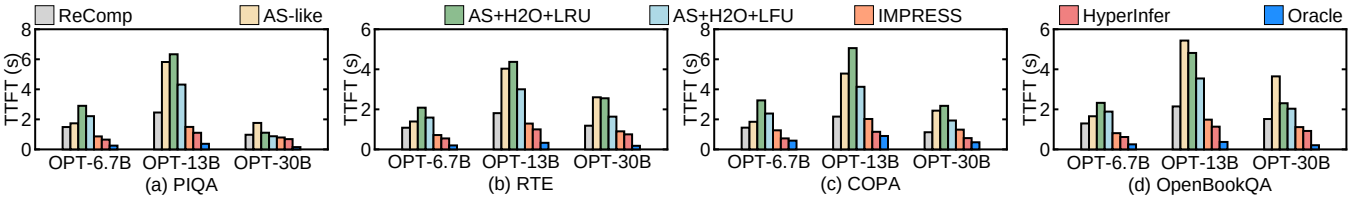


Fig. 18: The average TTFT with various systems across four datasets and three models.

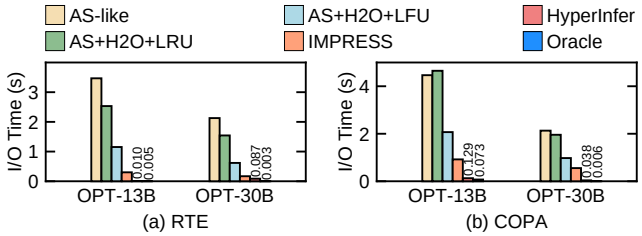


Fig. 19: The average prefix KV loading time on the inference critical path.

Dataset	Method	F1	EM	ROUGE-L
SQuAD	ReComp/AS	0.66	0.52	0.66
	AS+H2O+LRU	0.62	0.46	0.61
	HyperInfer	0.61	0.47	0.61
HotpotQA	ReComp/AS	0.61	0.48	0.61
	AS+H2O+LRU	0.61	0.47	0.61
	HyperInfer	0.59	0.45	0.59

TABLE I: Generation quality on document-grounded QA with Qwen3-8B (25% KV retention).

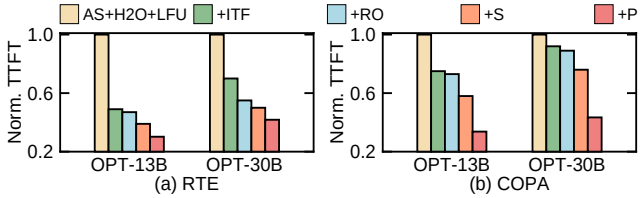


Fig. 20: The performance impact of each optimization.

it uses LFU to manage the cache. (5) *IMPRESS* [3]: It uses the same important-KV identification method and prefix KV storage management as HyperInfer, but lacks the prefetching mechanism. In contrast, *HyperInfer* distinguishes itself by speculatively prefetching the important KVs to overlap I/Os with computation. (6) *Oracle:* We include an Oracle baseline to approximate the theoretical lower bound of TTFT. Oracle uses the same prefetch window as HyperInfer, but prefetches the exact set of important KVs required by the next layer, eliminating both unnecessary prefetches and recovery loading. Therefore, the gap between HyperInfer and Oracle reflects the overhead caused by the mismatch between the estimated and actual important KV sets.

To prevent runtime out-of-memory errors and ensure only a portion of the prefix KVs are cached (the other KVs reside on SSD), we allocate 10GB of GPU cache and 32GB of CPU cache for prefix KVs, leaving the remaining GPU and CPU memory for storing model weights, the KV cache used during the decoding phase, and the input data. On average, PIQA, RTE, COPA, and OpenBookQA require 55GB, 57GB, 64GB, and 65GB of storage for prefix KVs across three models, respectively. Uniformly, each chunk holds keys or values from 64 tokens [38].

B. Overall Performance

Model generation quality. Figure 17 illustrates the impact of different systems on model generation quality at various prefix KV retention ratios. As ReComp and AS-like share the same accuracy, and AS+H2O+LRU and AS+H2O+LFU also show identical accuracy, we present only the accuracy of ReComp, AS+H2O+LRU, and HyperInfer for simplicity. It shows that HyperInfer has a negligible impact on accuracy across these datasets and models, achieving accuracy drop less than 1% compared to ReComp or AS+H2O+LRU. In some cases, HyperInfer even slightly improves accuracy over ReComp, suggesting that focusing on more important tokens can sometimes enhance generation quality. Table I shows that on SQuAD and HotpotQA, HyperInfer maintains generation quality close to AS+H2O+LRU and ReComp at 25% KV retention.

The average TTFT. We pre-warm both CPU and GPU caches for all systems, except ReComp (which doesn't require prefix KV reuse), before evaluating TTFT. We set the KV retention ratio to 50% for COPA and 25% for the other three datasets across all systems. With this setup, HyperInfer shows an average accuracy reduction of 0.2% compared to ReComp. Figure 18 shows the average TTFT per request for different systems. HyperInfer outperforms alternatives, with a 1.28× to 3.90× improvement over AS+H2O+LFU and a 1.16× to 1.75× improvement over IMPRESS. The gains come from lower critical-path I/O for loading prefix KVs into GPU memory: HyperInfer reduces I/O time by 1.74× to 4.4× over IMPRESS, and, in the workloads shown in Figure 19, reduces exposed I/O from 0.62–2.07s under AS+H2O+LFU to 0.01–0.13s. Other systems have longer I/O times, sometimes exceeding ReComp's TTFT. Besides, HyperInfer's performance gains vary across datasets and models due to different prefix KV sizes and computational demands. Notably, OPT-30B has a shorter TTFT than OPT-13B, as it uses shorter prefixes to avoid GPU memory overflow.

The tail latency. HyperInfer achieves the shortest p99 tail TTFT. For example, on the RTE dataset with OPT-30B model, the p99 latencies for ReComp, AS-like, AS+H2O+LRU, AS+H2O+LFU, IMPRESS, HyperInfer are 4.2s, 18.38s, 14.83s, 9.02s, 2.28s and 1.24s respectively. This shows HyperInfer effectively reduces the tail I/O latency when KVs are loaded from SSD.

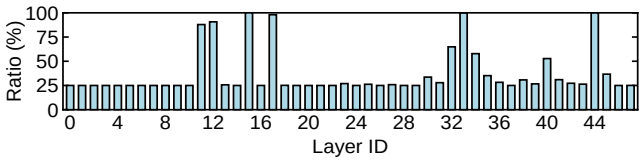


Fig. 21: The retention of KVs per model layer using similarity-guided important token identification.

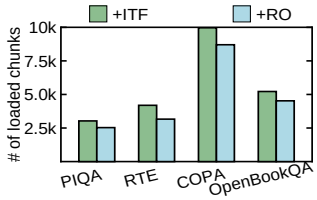


Fig. 22: The impact of KV reordering on the number of loaded chunks.

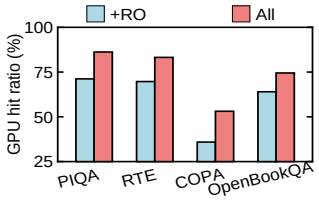


Fig. 23: The impact of score-based cache management on GPU hit ratios.

C. Impact of Individual Techniques

Figure 20 shows the TTFT reduction as optimizations are added step by step. Starting from AS+H2O+LFU, *+ITF* adds importance-guided KV identification, *+RO* further adds importance-informed KV reordering, and *+S* further adds score-based cache management, corresponding to the IMPRESS baseline. Finally, *+P* adds layer-aware prefetching on top of IMPRESS, corresponding to the full HyperInfer design. We observe that each added optimization reduces TTFT, with *+P* achieving the shortest TTFT, showing the effectiveness of HyperInfer's individual techniques. For instance, on OPT-30B with the RTE dataset, the four techniques contribute 51%, 26%, 9%, and 14% of the total TTFT reduction, respectively; on COPA with OPT-13B, their contributions shift to 38%, 3%, 23%, and 37%. In particular, the gap between *+S* and *+P* directly reflects the improvement of HyperInfer over IMPRESS, reducing TTFT by up to 42.9% on COPA with OPT-30B.

Figure 21 shows the average KV loading ratio per layer for OPT-30B on the PIQA dataset using the *+ITF* system, which dynamically adjusts KV loading to optimize the trade-off between accuracy and TTFT. Figure 22 indicates that enabling KV reordering results in an average 1.2× reduction in loaded chunks, as it consolidates important keys and values into fewer chunks. Figure 23 shows that score-based cache management boosts the average GPU hit ratio from 68% to 80% across four datasets, thereby reducing PCIe data transfers.

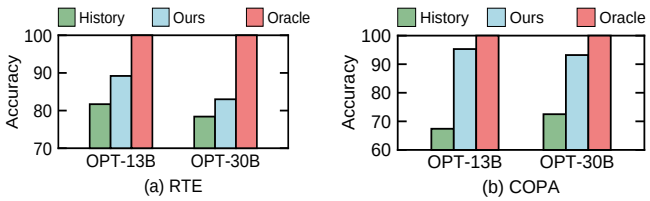


Fig. 24: Prefetch accuracy on (a) RTE and (b) COPA datasets.

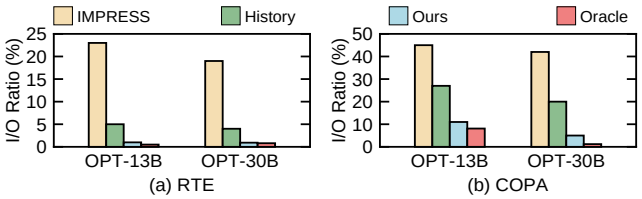


Fig. 25: I/O ratio on (a) RTE and (b) COPA datasets.

D. Prefetching Performance Analysis

HyperInfer employs the layer-aware KV prefetching mechanism described in section IV-C (denoted as *Ours*) to speculatively prefetch important KVs for the next layer. We benchmark it against Oracle and a naive history-based method (denoted as *History*), which prioritizes KVs solely based on historical usage frequency, independent of the current request context, as detailed in §III. We evaluate the two methods along two dimensions: (1) *Prefetch accuracy*, the proportion of prefetched KVs that align with the ground-truth important KVs of the next layer; and (2) *I/O ratio*, defined as the percentage of total I/O time that remains exposed on the inference critical path (i.e., cannot be overlapped with computation). These metrics collectively reveal the precision of KV selection and the effectiveness of latency hiding.

Figure 24 compares the prefetch accuracy of our method against the history-based baseline. Our approach consistently achieves higher accuracy across two models and two datasets, yielding an average improvement of 16%. This result indicates that layer-aware KV prefetching is significantly more effective at predicting next-layer KV importance. Benefiting from this higher precision, the volume of KV loading exposed on the critical path is substantially reduced. As shown in Figure 25, Our method substantially mitigates the I/O bottleneck by minimizing the proportion of data loading exposed on the critical path. Compared to IMPRESS, where I/O is fully exposed due to the read-after-computation dependency, our method reduces the critical path I/O ratio by up to 37%, approaching the performance of the ideal Oracle. Furthermore, by accurately predicting important KVs to effectively hide I/O latency, our approach reduces the critical path I/O ratio by up to 16% compared to the history-based method.

E. Sensitivity Analysis

Alpha value of similarity threshold. Figure 26 depicts the impact of varying the alpha value on inference accuracy and TTFT, ranging from 0 to 2. The findings indicate that while an increased alpha reduces TTFT, it marginally affects inference accuracy due to a lowered similarity threshold, leading to less KV loading. This trend is consistent across datasets. Thus, we optimize alpha at 0.6 for a balance between low TTFT and high inference precision.

Chunk size. Figure 27 compares the TTFT of the leading system and HyperInfer with chunk sizes ranging from 16 to 256. Notably, HyperInfer achieves 2.3× to 3× improvement over the AS+H2O+LFU system across all sizes, underscoring HyperInfer's robustness regardless of chunk dimensions. Accuracy, being unaffected by chunk size, is not depicted.

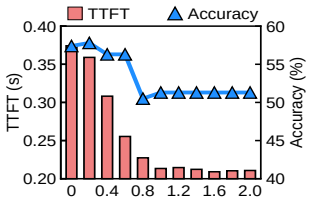


Fig. 26: Results of various alpha values.

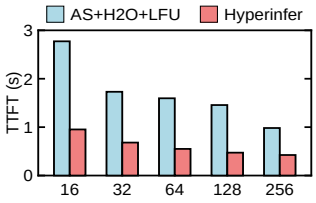


Fig. 27: Results of various chunk sizes.

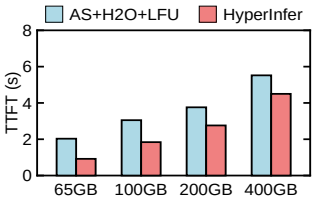


Fig. 28: Results on various dataset sizes.

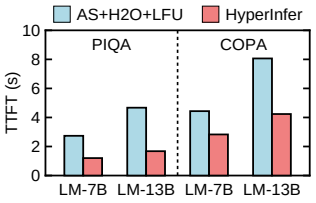


Fig. 29: Results on Llama (LM) models.

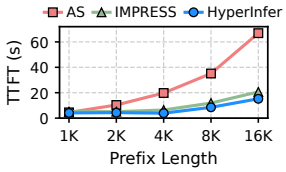


Fig. 30: Results on various prefix lengths.

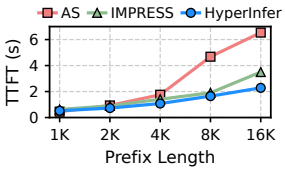


Fig. 31: Results on high-bandwidth host-memory KV storage.

Dataset size. We vary the number of prefixes (i.e., few-shot examples) to create different variants of OpenBookQA, with dataset sizes ranging from 65GB to 400GB. Figure 28 shows that HyperInfer consistently outperforms the leading comparison system, AS+H2O+LFU, achieving speedups ranging from 1.2× to 2.2×.

Model type. Figure 29 shows the average TTFT for Llama2-7B and Llama2-13B models on PIQA and COPA. It demonstrates that HyperInfer achieves a 1.6×-2.9× speedup on Llama models, attributed to similar reasons in section VI-B.

Prefix length. With reusable KVs stored on SSD, we fix the suffix length to 1K tokens and vary only the reusable prefix length from 1K to 16K tokens. Figure 30 shows that HyperInfer consistently reduces TTFT over AS and IMPRESS, achieving 1.10×–5.09× and 1.17×–1.64× speedups, respectively.

High-bandwidth host-memory KV storage. We evaluate HyperInfer on an NVIDIA H800 with PCIe 5.0, with all reusable KVs placed in DRAM. Higher bandwidth reduces the I/O bottleneck for short prefixes, but the KV volume still grows with prefix length. At 16K prefix length, HyperInfer reduces TTFT by 2.86× over AS and 1.53× over IMPRESS, showing that HyperInfer remains effective in this setting.

F. Overhead Analysis

Time overhead. The similarity-guided important token identification technique in HyperInfer loads keys from the probe heads to determine the important token index set. While this adds some I/O and computation overhead, it averages only 6% of our system's overhead due to the limited number of probe heads, resulting in minimal loading and computation. For layer-aware prefetching, verification only performs lightweight per-layer index set-difference operations, costing about 40 μs per layer, less than 0.01% of COPA TTFT on OPT-13B. Recovery then reuses IMPRESS's demand-loading path: while IMPRESS synchronously loads the full important KV set G, HyperInfer loads only the missed subset $G \setminus P$, where P is the prefetched set. Thus inaccurate prefetching only leaves the missed KVs $G \setminus P$ on the critical path, and in the worst case where $P \cap G = \emptyset$, HyperInfer degrades toward IMPRESS with only microsecond-level verification overhead. Furthermore, KV reordering asynchronously sorts tokens by importance and repacks them onto disk, with a total experimental execution time of less than one minute. This process is non-intrusive to TTFT as it operates outside the critical path.

Space overhead. KV reordering adds a mapping list to each chunk's metadata for token position mappings post-reordering. Score-based cache management adds a score per chunk. With a 64-token chunk size, these additions account for less than 0.5% of the chunk's memory. The prefetch buffer is negligible as it stores only one layer of KVs, occupying at most 0.3% of the total GPU memory. Additionally, to avoid loading data from other heads when loading keys from the probe heads, we redundantly store the keys from the probe heads separately. This accounts for 1.2% of the total storage of all prefix KVs, which is a minimal cost considering high-capacity disks.

VII. Related Work

Prefix KV reuse and management. Existing systems reduce redundant KV computation from different angles. vLLM [16] and FlexGen [32] optimize runtime KV-cache management and offloading; prefix-reuse systems [3], [8], [15], [21], [38], [44] store and reuse shared prefix KVs across requests to lower TTFT; and PromptCache [10] and CacheBlend [37] extend reuse to a finer text-segment granularity. More recent KV-cache managers and serving frameworks orchestrate cache reuse, placement, transfer, scheduling, and runtime execution across engines and storage tiers. Among them, TensorRT-LLM [26] targets GPU runtime and kernel execution at a different layer and is orthogonal to HyperInfer, whereas LM-Cache [6], Mooncake [29], FlashGen [14], and Dynamo [27] operate in the same KV-reuse space but none performs importance-based selective loading or cross-layer prefetching. HyperInfer is complementary to these systems and can be integrated to cut the volume of KV data they transfer while preserving output quality.

KV pruning and quantization. Recent studies [18], [22], [32], [43] show LLM inference can achieve similar output quality using only a subset of KVs, proposing various methods to identify these KVs. However, they require full keys during the prefill phase, leading to high I/O latency when combined with prefix KV storage systems. HyperInfer leverages the similarity of important token indices across heads to identify important KVs with minimal I/O, reducing TTFT while maintaining high accuracy. Others [13], [21], [23], [32], [40] focus on KV quantization to reduce bit counts per key and

value element. They can complement HyperInfer to further decrease data load.

LLM serving scheduling and orchestration. Some works optimize serving-level orchestration, including request scheduling [1], [39], model-parallel execution [35], prefill-decoding disaggregation [28], [45], and distributed KV-cache management [20], [29]. These optimizations are orthogonal to HyperInfer, which targets the critical-path I/O volume for reused prefix KVs.

VIII. Conclusion

Existing prefix-KV storage systems often fail to reduce TTFT when disk I/O dominates. We propose HyperInfer, an importance-aware prefix-KV caching and prefetching system that minimizes I/O latency by selectively loading and prefetching important KVs. HyperInfer combines efficient important-KV identification, layer-aware prefetching, and optimized storage management. Experiments show that HyperInfer reduces TTFT by up to 1.75× over state-of-the-art systems while maintaining comparable accuracy.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[2] Toufique Ahmed and Premkumar Devanbu. Better Patching Using LLM Prompting, via Self-Consistency. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[3] Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. IMPRESS: An Importance-Informed Multi-Tier Prefix KV Storage System for Large Language Model Inference. In *23rd USENIX Conference on File and Storage Technologies (FAST)*, 2025.

[4] Weijian Chen, Shuibing He, Yaowen Xu, Xuechen Zhang, Siling Yang, Shuang Hu, Xian-He Sun, and Gang Chen. iCache: An Importance-Sampling-Informed Cache for Accelerating I/O-Bound DNN Model Training. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[5] Weijian Chen, Shuibing He, Ruidong Zhang, Xuechen Zhang, Ping Chen, Siling Yang, Haoyang Qu, and Xuan Zhan. ImPACT: Importance-Informed Prefetching and Caching for I/O-Bound DNN Training. *IEEE Transactions on Computers (TC)*, 2025.

[6] Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference. *arXiv preprint arXiv:2510.09665*, 2025.

[7] Sam Fletcher, Md Zahidul Islam, et al. Comparing Sets of Patterns with the Jaccard Index. *Australasian Journal of Information Systems (AJIS)*, 2018.

[8] 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. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC)*, 2024.

[9] Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, et al. A Framework for Few-Shot Language Model Evaluation, 2024. https://zenodo.org/records/12608602.

[10] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. In *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

[11] Shuibing He, Ping Chen, Shuaiben Chen, Zheng Li, Siling Yang, Weijian Chen, and Lidan Shou. HOME: A Holistic GPU Memory Management Framework for Deep Learning. *IEEE Transactions on Computers (TC)*, 2023.

[12] Amr Hendy, Mohamed Abdelrehim, Amr Sharaf, Vikas Raunak, Mohamed Gabr, Hitokazu Matsushita, Young Jin Kim, Mohamed Afify, and Hany Hassan Awadalla. How Good Are GPT Models at Machine Translation? A Comprehensive Evaluation. *arXiv preprint arXiv:2302.09210*, 2023.

[13] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization. *arXiv preprint arXiv:2401.18079*, 2024.

[14] Jinwoo Jeong and Jeongseob Ahn. Accelerating LLM Serving for Multi-turn Dialogues with Efficient Resource Management. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS)*, 2025.

[15] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *arXiv preprint arXiv:2404.12457*, 2024.

[16] 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 PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[17] Peter Lee, Sebastien Bubeck, and Joseph Petro. Benefits, Limits, and Risks of GPT-4 as An AI Chatbot for Medicine. *New England Journal of Medicine (NEJM)*, 2023.

[18] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[19] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[20] Bin Lin, Tao Peng, Chen Zhang, Minmin Sun, Lanbo Li, Hanyu Zhao, Wencong Xiao, Qi Xu, Xiafei Qiu, Shen Li, et al. Infinite-LLM: Efficient LLM Service for Long Context with DistAttention and Distributed KVCache. *arXiv preprint arXiv:2401.02669*, 2024.

[21] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM Special Interest Group on Data Communication Conference (SIGCOMM)*, 2024.

[22] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[23] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache. *arXiv preprint arXiv:2402.02750*, 2024.

[24] Pan Lu. Chameleon-LLM, 2024. https://github.com/lupantech/chameleon-llm/blob/main/run_tabmwp/demos/prompt_policy.py.

[25] Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-Play Compositional Reasoning with Large Language Models. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[26] NVIDIA. Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM.

[27] NVIDIA. Dynamo Inference Framework. https://developer.nvidia.com/dynamo, 2025.

[28] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient Generative LLM Inference using Phase Splitting. In *Proceedings of ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024.

[29] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran 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)*, 2025.

[30] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know What You Don't Know: Unanswerable Questions for SQuAD. *arXiv preprint arXiv:1806.03822*, 2018.

[31] Vikas Raunak, Amr Sharaf, Yiren Wang, Hany Awadalla, and Arul Menezes. Leveraging GPT-4 for Automatic Translation Post-Editing. In *Proceedings of Findings of the Association for Computational Linguistics (EMNLP)*, 2023.

[32] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-Throughput Generative Inference of Large Language Models with A Single GPU. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.

[33] ShareGPT Teams. ShareGPT, 2023. https://sharegpt.com.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[35] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2024.

[36] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[37] 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 for RAG with Cached Knowledge Fusion. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*, 2025.

[38] Lu Ye, Ze Tao, Yong Huang, and Yang Li. Chunkattention: Efficient Self-Attention With Prefix-Aware KV Cache and Two-Phase Partition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

[39] 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 *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[40] Yuxuan Yue, Zhihang Yuan, Haojie Duanmu, Sifan Zhou, Jianlong Wu, and Liqiang Nie. WKVQuant: Quantizing Weight and Key/Value Cache for Large Language Models Gains More. *arXiv preprint arXiv:2402.12065*, 2024.

[41] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. OPT: Open Pre-Trained Transformer Language Models. *arXiv preprint arXiv:2205.01068*, 2022.

[42] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models. *arXiv preprint arXiv:2309.01219*, 2023.

[43] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[44] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. SGLang: Efficient Execution of Structured Language Model Programs. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[45] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

RuiDdong Zhang is currently working toward the PhD degree at the College of Computer Science and Technology, Zhejiang University, China. His research focus on systems for AI and storage for AI.

Weijian Chen received the PhD degree in computer science and technology from Zhejiang University, in 2025. His research focuses on memory and storage system for AI.

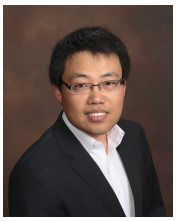
Ping Chen received the PhD degree in computer science and technology from the College of Computer Science and Technology, Zhejiang University, China, in 2025. He is now a researcher with the Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security and the State Key Laboratory of Blockchain and Data Security, Zhejiang University. His research focuses on intelligent computing and memory management for AI systems.

Shuibing He (Member, IEEE) received the PhD degree in computer science and technology from Huazhong University of Science and Technology, in 2009. He is now a tenured professor with the College of Computer Science and Technology, Zhejiang University, China. His research areas include intelligent computing, high-performance computing, and memory and storage systems. He is a member of the IEEE and ACM.

Wenjie Liang is currently a fourth-year undergraduate student in the College of Computer Science and Technology at Zhejiang University. His research interests lie in the system for AI.

Rongchen Liu is currently working toward the PhD degree with the College of Computer Science and Technology, Zhejiang University, China. His research focuses on intelligent computing and memory management for Al systems.

Siling Yang (Graduate Student Member, IEEE) received the PhD degree in computer science and technology from the College of Computer Science and Technology, Zhejiang University, China, in 2025. She is now a researcher with the School of Software Technology, Zhejiang University, China. Her research focuses on intelligent computing and processing-in-memory for AI.



Xuechen Zhang (Member, IEEE) received the M.S. and Ph.D. degrees in Computer Engineering from Wayne State University. He is currently an Associate Professor in the Department of Information Technology at Kennesaw State University, Marietta Campus. His research interests include file and storage systems, operating systems, high-performance computing, and systems for AI. He is a member of the IEEE and ACM.