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

immediate

***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 speculative 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. Leveraging the observed similarity of important token index sets across adjacent transformer layers, it further introduces a speculative prefetching mechanism to hide I/O latency along the critical inference path. Finally, HyperInfer optimizes prefix KV storage and cache utilization through importance-informed KV management, achieving additional reductions in TTFT for end-to-end inference. Our experimental results show that HyperInfer can reduce TTFT by up to ?× compared to state-of-the-art systems, while maintaining comparable inference accuracy.**

I. Introduction

GENERATIVE large language models (LLMs), such as ChatGPT and GPT-4, are increasingly utilized in applications like chatbots [17], [38], document summarization [3], [8], and translation [10], [32] 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 request 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 [6], and examples of question-answer pairs [26] to guide the model's output format.

While longer requests enhance the quality of the model's output, they also significantly increase response-generation latency, particularly the time to first token (TTFT). This occurs because the computational complexity of processing a request grows superlinearly with its length [36]. 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 [9], [13], [40], [46], which can become insufficient for lengthy sequences or large batches, thus restricting TTFT reduction. AttentionStore [6] 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 section II-B.

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 [18], [23], [33], [45]. Based on this insight, we propose an importance-aware, multi-tier prefix KV caching and prefetching system, HyperInfer, which leverages GPU memory, CPU memory, and disks to reduce LLM inference latency while maintaining comparable output quality. GPU and CPU memory serve as caches for data stored on disks. The core idea is to load only the KVs of important prefix tokens, minimizing I/O from slower disks, while prefetching future potentially critical KVs to overlap I/O with computation, further reducing latency. However, building such an efficient system involves three significant 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, since the important tokens of each layer are only revealed after computing that layer's attention scores, they cannot be prefetched during the computation of the previous layer. Existing systems either ignore token importance and randomly prefetch some tokens' KVs, which leads to low accuracy and wastes I/O bandwidth, or they require changes to model parameters, which introduces extra overhead. Hence, a more effective method is required to identify which tokens' KVs to prefetch in order to reduce I/O on the critical path.

Third, existing systems often consolidate consecutive KVs into large objects (e.g., chunks), to optimize disk I/O 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 [13], [39], [46], 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 adress the second challenge, we leverage the observation that important token distributions also exhibit inter-layer similarity across adjacent transformer layers. Building on this property, we design a layer-aware speculative prefetching mechanism that uses the important tokens identified at layer i as a prior to predict those of layer i+1. This enables the system to prefetch the next layer's KVs during the current layer's computation, overlapping I/O with computation and further reducing first-token latency.

To adress the third challenge, we propose importance-informed KV management 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 its performance on three LLM models with various sizes (from 6.7B to 30B) across four datasets. Our experimental results show that HyperInfer achieves up to a 3.8× reduction in I/O time and a 2.8× reduction in TTFT, 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 prefetching and caching system that integrates three storage tiers: GPU memory, CPU memory, and disk.
- We propose a similarity-guided important token identification method to identify important KVs, significantly reducing I/O overhead.
- We propose layer-aware speculative prefetching mechanism to prefetch important KVs, effectively overlapping I/O with computation.
- We devise importance-informed KV management methods including KV reordering and a new score-based cache management policy to further minimize I/O data volume from slower storage media.
- We implement HyperInfer and our evaluation demonstrates that it reduces TTFT by up to ?× compared to the state-of-the-art prefix KV storage systems while maintaining comparable inference accuracy.

II. Background and Motivation

A. Architecture and Workflow of Large Language Models

Model inference. A generative large language model (LLM) consists of an input layer, a stack of transformer layers, and an output layer, as illustrated in Figure 1. Given an input sequence $S = [t_0, t_1, \ldots, t_{l-1}]$ of l tokens, the input layer maps it to a tensor $X_{\text{in}} \in \mathbb{R}^{l \times d}$, where d is the hidden dimension. This tensor is then processed sequentially through n transformer layers. Each layer must complete its computation before passing its output $X_{\text{out}_i} \in \mathbb{R}^{l \times d}$ to the next layer, forming a strictly layer-by-layer dependency. The final layer output $X_{\text{out}_{n-1}}$ is then fed into the output layer to generate the next token t_l. The new token is appended to the input sequence and reprocessed to generate subsequent tokens.This iterative process continues until reaching the maximum token limit or a special end-of-sequence (EOS) token. The generation of the first token is referred to as the *prefill* phase, while the generation of subsequent tokens is termed the *decoding* phase.

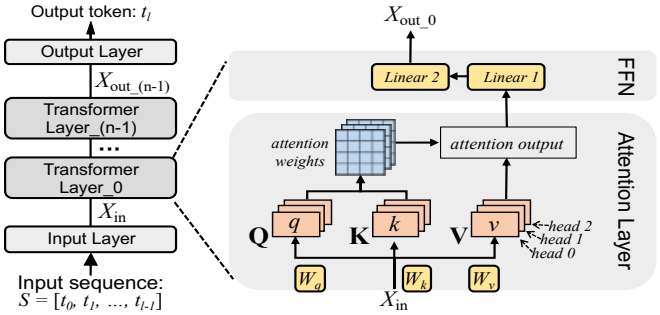


Fig. 1: The LLM model structure.

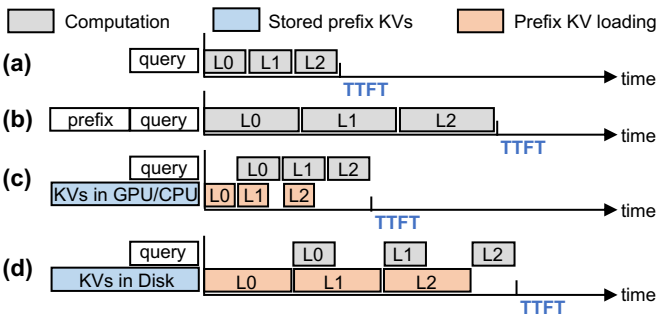


Fig. 2: The TTFTs of various cases. Assume the LLM model consists of three transformer layers, denoted as 'Lx'.

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. Shared Prefixes and Storage System

Long TTFT due to the use of context-rich prefixes. Directly using large models for inference may yield suboptimal results. Specifically, when asked about recent events absent from the training data, the model can produce incorrect or misleading responses due to issues [44]. To enhance response quality, applications often augment user *queries* with context-rich *prefixes*, forming complete *requests* that are fed into the LLM as input sequences. For example, Retrieval-Augmented Generation (RAG) [19] retrieves external documents relevant to the

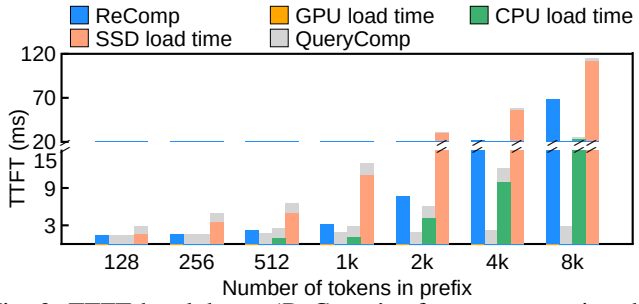


Fig. 3: TTFT breakdown. 'ReComp' refers to not reusing the prefix KV. 'QueryComp' denotes the remaining computation after loading the prefix KV.

user query. Advanced GPT plugins, such as Chameleon [26], embed tool definitions in the system prompt and use few-shot examples to guide complex reasoning. Multi-turn dialogue systems [6] incorporate previous question–answer pairs to better capture user intent, while self-consistency [2] improves accuracy by generating multiple responses and aggregating them through voting.

Figures 2(a) and 2(b) show that although context-rich prefixes enhance response quality, they substantially increase the 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 [25]. Given that an average real-world query contains approximately 750 tokens [35], 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.

Prefix KV storage systems and I/O bottleneck. Researchers have observed that these prefixes are often partially or completely shared across different requests [6], [9], [13], [22], [40], [46]. 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 [6]. 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 Storage Systems

An effective prefix KV storage system must satisfy two key requirements. First, it must provide sufficient storage capacity to hold a large number of prefix KVs. Second, it must ensure low latency for prefetching prefix KVs, as excessive loading delay can become a bottleneck in LLM inference and limit the reduction of time-to-first-token (TTFT). However, no existing system can simultaneously satisfy both requirements across diverse workloads. The current designs can be broadly categorized into three types, as summarized below.

a) GPU/CPU-Memory-Only Storage: Most existing systems, such as PromptCache [9], SGLang [46], RAGCache [13], and ChunkAttention [40], store prefix KVs only in GPU and/or CPU memory to minimize prefetch latency, as illustrated in Figure 2(c). This design achieves low TTFT by avoiding I/O operations. However, GPU and CPU memory are both limited in capacity, and the prefix KV storage space can quickly become exhausted under large-scale or multi-session scenarios. For example....

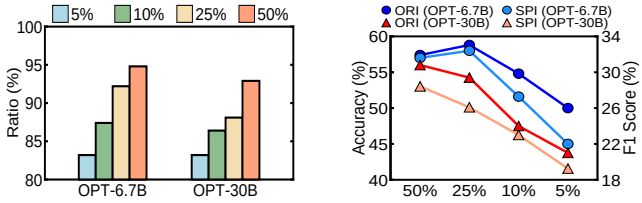
b) Hybrid CPU-and-Disk Storage: To overcome the memory limitation, AttentionStore [6] stores prefix KVs across CPU memory and disk, providing sufficient capacity for long-context workloads. However, this design fails to eliminate loading latency, as disk I/O cannot always be overlapped with computation, especially under heavy loads. As shown in Figure 2(d) and Figure 3, loading prefix KVs from SSD incurs significantly higher TTFT than from GPU or CPU memory, since SSD-to-GPU data transfer latency—accounting for 51%–98% of TTFT—is rarely hidden by computation. Thus, disk access becomes a new performance bottleneck, particularly for longer prefixes.

c) Importance-Aware KV Loading: To mitigate the I/O bottleneck caused by loading prefix KVs from storage, IMPRESS [4] introduces a novel approach that loads only the most important KVs to reduce I/O latency while maintaining comparable model accuracy. However, these selected KVs still lie on the critical path of model inference. Although prior works employ prefetching to overlap data loading with model computation and thereby reduce I/O latency, this strategy fails in IMPRESS. The selection and loading of important KVs for the next layer depend on the computation results of the current layer, making it impossible to determine which KVs to prefetch in advance. Consequently, the I/O latency of these KVs remains exposed on the critical path, limiting end-to-end inference efficiency.

D. Not All KVs Are Equally Important

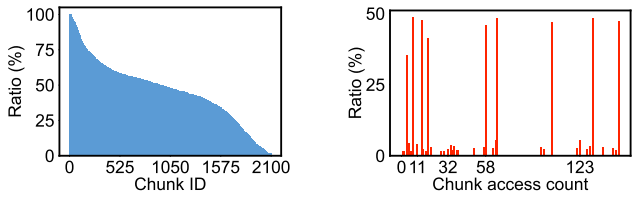
Recent research [18], [23], [33], [45] indicates that not all tokens' KVs are equally important for maintaining LLM inference accuracy. These methods generate and store the full set of KVs during the prefill phase, then identify and discard the less important tokens' KVs during decoding by analyzing the attention weights. This approach reduces the computational load during the decoding phase while maintaining comparable LLM inference accuracy.

Inspired by this, we propose an importance-informed three-tiered prefix KV caching and speculative prefetching system that encompasses GPU memory, CPU memory, and disk storage. Our system addresses the disk I/O bottleneck through a two-pronged approach: selective loading and speculative prefetching. When reusing prefix KVs, we selectively load only important KVs for prefill and decoding computations, while unimportant tokens' KVs are discarded. To further optimize I/O efficiency, we propose a layer-aware speculative prefetching mechanism that uses the current layer's important token indices to predict and prefetch the next layer's



(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.

important KVs during computation, effectively overlapping I/O operations with computation. This integrated approach of selective loading combined with cross-layer speculative prefetching fundamentally alleviates the disk bottleneck by both reducing the total I/O volume and moving critical I/O operations off the inference critical path, thereby significantly reducing TTFT.

III. Design Challenges of HyperInfer

Challenge 1: Directly applying existing important token identification methods on prefix KV loading still has significant I/O overhead. Existing methods must load all prefix keys into GPU memory to compute attention weights and then identify important KVs [18], [23], [33], [45]. Reducing only the loading time of prefix values limits TTFT reduction.

A straightforward approach to avoid loading all prefix keys is to statically record the identified important tokens for queries. Then, when another query with the same prefix arrives, only the KVs of pre-identified important tokens would be loaded, reducing the I/O time. However, this approach has a significant flaw. We observed that the importance of tokens within the same prefix can vary depending on the specific query. We intuitively explain the observation here. For example, in a RAG scenario, different queries might use the same document as the prefix, but the relevant answers could be found in different text segments of the prefix. Thus, the method that pre-identifies important tokens could miss critical ones, substantially degrading the accuracy of LLM inference.

To confirm this limitation, we assess the recall ratio of important tokens and its effect on model generation quality across two models and datasets: OPT-6.7B on RTE [7] and OPT-30B on SQuAD [31]. We evaluate generation quality using accuracy and F1 score [22] for the two datasets respectively. As depicted in Figure 4, even with recall ratios above 80%, the omission of vital KVs results in accuracy and F1 score declines of up to 5% for RTE and 3.3% for SQuAD. Consequently, there is a need for a dynamic method that can discern important tokens within prefixes for various queries, while minimizing I/O overhead.

Challenge 2: The distribution of important tokens is unknown before prefetching. Existing methods determine the important KVs of each layer by computing the attention weights of the current layer. As a result, the distribution of important tokens for the next layer is not available during the current layer's computation. This prevents prefetching of the next layer's important KVs while the current layer is being processed, leaving I/O bandwidth underutilized.

A straightforward alternative is to randomly prefetch KVs for the next layer without considering their importance, but this approach suffers from low accuracy and brings many unimportant tokens' KVs into GPU memory, thereby wasting I/O bandwidth.

To verify this, we measured the recall rate of randomly prefetching KVs across two models and datasets, xxx and xxx. As shown in Figure xxx, when KVs were pre-fetched randomly, the recall rate was only xxx. This approach leads to a large number of unimportant KVs being loaded into CPU memory, which are not used during inference, resulting in wasted I/O bandwidth and GPU memory.

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 [6], [15], [40], [46], 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.

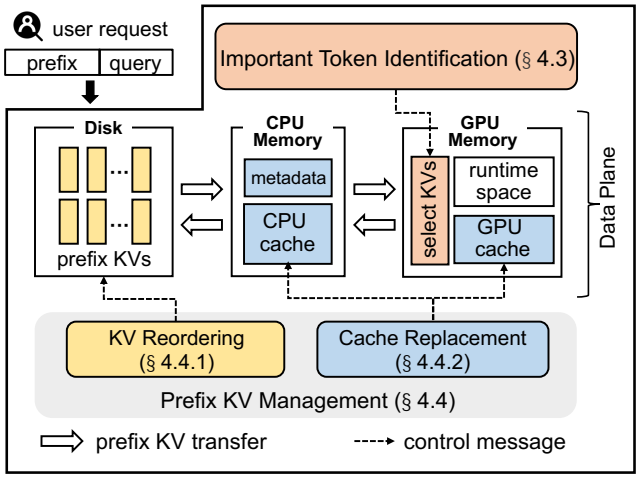


Fig. 6: Overview of the HyperInfer system.

IV. HyperInfer Design

In this section, we present HyperInfer, an importance-informed three-tier prefix KV caching and prefetching system. First, we describe the overall architecture of HyperInfer (section IV-A). Then, we introduce an I/O-efficient technique to identify important tokens (section IV-B). Furthermore, we propose a speculative prefetching technique to mitigate I/O bottlenecks by overlapping computation and I/O (section 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 (section IV-D).

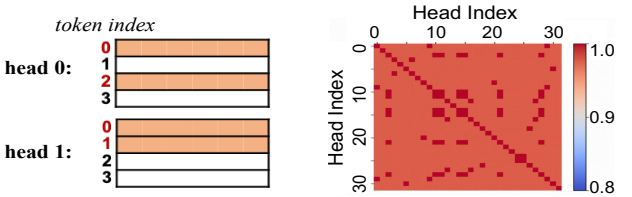
A. Overview

We propose HyperInfer to provide large storage capacity for prefix KVs while ensuring efficient I/O accesses to reduce TTFT. The system is designed based on two principles: (1) Using a minimal number of I/Os to identify the important KVs within a prefix, allowing only the essential KVs to be loaded during the prefill phase; (2) Since loading only important KVs could degrade the efficiency of existing storage and caching systems, we optimize the three-tier prefix KV 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 avoid space wastage. The metadata in the CPU memory is organized using a radix tree [46], which facilitates quick searches for the reusable prefix KVs. The runtime space stores model parameters and intermediate data needed for GPU inference. HyperInfer has two control components including important token identification (ITF) (section IV-B) and prefix KV management (PKM) (section IV-D). ITF identifies important tokens within a chunk by loading only partial keys rather than all of them, reducing the amount of data loaded from disks. PKM manages the storage and data movement of prefix KVs across disks and the two cache spaces in CPU and GPU memory.

Dataflow of HyperInfer. Assume that a request $S = [t_0^p, t_1^p, ..., t_{m-1}^p, t_0^q, t_1^q, ..., t_{n-1}^q]$ arrives. m is the number of tokens in prefix and n is the number of tokens in the query. t^p and t^q denote the prefix and non-prefix tokens respectively. First, given the request, HyperInfer searches the radix tree to find the longest common prefix subsequence from all previous requests [40], [46]. Let's assume the result is $R = [t_0^p, t_1^p, ..., t_j^p]$, whose KVs are stored in GPU memory, CPU memory, or disk. There may also be some prefix tokens $NR = [t_{j+1}^p, ..., t_{m-1}^p]$ that are not in the radix tree, and therefore their KVs do not exist in the system. Next, HyperInfer employs the I/O-efficient ITF method to identify the important tokens within R , assuming $R_{important} = [t_t^p, t_{t+1}^p, ..., t_s^p]$ $(0 \le t \le s \le j)$ is identified. If KVs in $R_{important}$ are not in GPU memory, they are loaded from disk or CPU memory. The KVs of unimportant tokens in R are not reused and do not participate in further inference. Then, the loaded $R_{important}$, the tokens NR, and the tokens $[t_0^q, t_1^q, ..., t_{n-1}^q]$ are sent into LLM model, completing the remaining computations in the prefill phase. Essentially, $R_{important}$ plus NR becomes the defacto prefix used in the LLM inference, replacing the set of $\{t^p\}$ in S. Finally, the newly generated KVs for the prefix token in NR are stored on disk, and the prefix tokens in NR are inserted into the radix tree for future reuse by other requests. The decoding phase remains unchanged, following existing systems [36].

Importance metric. In this paper, we use the sum of values in each column of the attention weight matrix as the token's importance, following the same method in H2O [45]. A higher sum indicates greater token importance. Our system is also compatible with other metrics for measuring token importance [18], [23], [33].



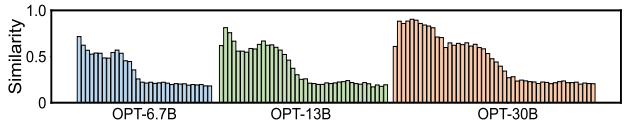
(a) Two Heads of K Tensors (b) Jaccard Scores Between Heads

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.

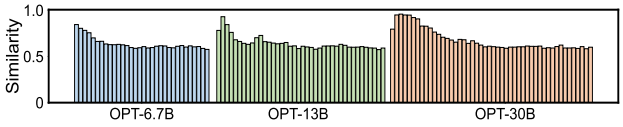
B. Similarity-Guided Important Token Identification

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 described in section II-A, each token has K and V tensors for every head. We found that the set of important token indices is highly similar across different heads within the same layer. This is intuitive because the k or v tensors in different heads are derived from the same large K or V tensors [36], [43]. Therefore, if a token is significantly more important than another in one head, it is highly likely that this importance relationship holds in other heads as well. Note that although we cannot strictly prove these observations mathematically for all LLMs and scenarios, we will demonstrate the generality and

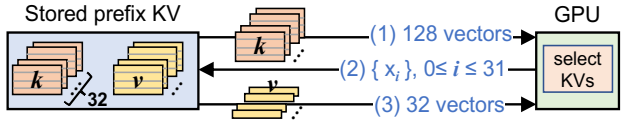


(a) select the top 10% most important tokens

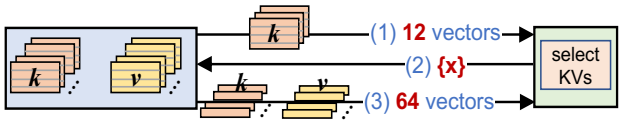


(b) select the top 40% most important tokens

Fig. 8: Similarities of important token index sets across all transformer layers.



(a) w/o similarity-guided token selection



(b) w/ similarity-guided token selection

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.

practicality of these observations through accuracy evaluations (section VI-B) after applying the corresponding techniques on four datasets across three models.

To quantitatively measure the similarity between these index sets, we use the Jaccard index [5]. Assume the sets of the most important token indices selected from two heads are A and B. The jaccard index is defined as the size of the intersection divided by the size of the union of the two sets: $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$. The Jaccard value is 0 when the two important token index sets are completely different and 1 when they are identical. Figure 7(a) shows an example where an input sequence contains four tokens, of which 50% are considered as important tokens. The index set of important tokens for the keys of head 0 is $h_0 = \{0, 2\}$, and for head 1 it is $h_1 = \{0, 1\}$. The similarity between the important token indices in these two heads is $J(h_0, h_1) = \frac{|h_0 \cap h_1|}{|h_0 \cup h_1|} = \frac{1}{3}$. Figure 7(b) shows a real similarity heatmap of the important token sets in the keys produced by the middle transformer layer of the OPT-6.7B model. It demonstrates that these similarities are quite high, 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 gain further insights, we study the average similarity of token index sets across different heads for the OPT-6.7B, OPT-13B, and OPT-30B models, when selecting the top 10% and 40% most important tokens (Figure 8). Each bar in the figure represents the average value for a single transformer layer. These three models contain a total of 32, 40, and 48 transformer layers, respectively. We have two findings. (1) The higher ratio of important tokens selected, the greater the similarity. For instance, when selecting 40% and 10% of the tokens, the average similarity across all layers of OPT-30B is 0.68 and 0.48, respectively. This aligns with intuition, as the similarity reaches 1 when all the tokens (100%) are selected. (2) Although smaller models and deeper transformer layers tend to exhibit lower similarities, they are still significantly higher than the expected value from random selection in most cases.

Based on the above observations, we propose the similarity-guided important token identification technique. The core idea is that *because of the similarity we can leverage the important token index set generated from a few selected heads to approximate the important token index sets for 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 disk. The number of probe heads is set to three. Without the similarity-guided important token identification technique, it involves three steps: (1) loading the keys of all 32 heads (total 32 × 4 = 128 vectors) from disk into the GPU memory; (2) GPU calculates attention weights using the query and all the keys, identifies the most important token index in each head i: $\{x_i\}$, $0 \leq i \leq 31$, and returns them to the CPU memory; The detailed identification algorithm is based on H2O [45]. (3) The CPU then loads the k and v vectors of $\{x_i\}$ from each head (total 32 × 1 = 32 vectors) into the GPU memory to complete the remaining prefill computations. The entire process loads a total of 128 + 32 = 160 vectors.

In contrast, with the similarity-guided important token identification technique, the steps are as follows: (1) only the keys from the three probe heads (total 3 × 4 = 12 vectors) are loaded from disk into the GPU memory; (2) GPU calculates attention weights using the query and the keys from the probe heads, identifies the important token index sets of the three heads, and computes the average Jaccard similarity. If this similarity exceeds the similarity threshold, only one token index deemed most important by all probe heads, $\{x\}$, are returned to the CPU; Then, proceed to step (3). If the threshold is not met, the computation mode of the current layer falls back to the version without enabling the similarity-guided important token identification method. (3) The CPU then loads

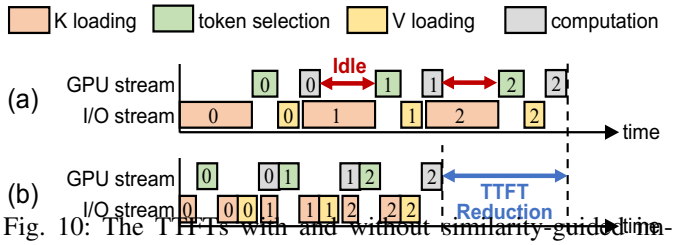


Fig. 10: The TTFTs with and without similarity-guided important token identification. Assume the LLM model consists of three transformer layers. The numbers inside the rectangles represent the layer index.

the k and v vectors of $\{x\}$ from each head (total 32 $\times$ 1 $\times$ 2 = 64 vectors) into the GPU memory for the remaining computations. The entire process loads a total of 12 + 64 = 76 vectors. While the I/O data volume cannot be reduced when the probe heads' similarity does not exceed the threshold, this situation only occurs in less than 20% transformer layers in the HyperInfer system on average (see section VI-C). Moreover, in practical LLM models, each transformer layer typically has dozens of heads (e.g., 32-96 in various sizes of the OPT model) and the prefix contains thousands of tokens [22], [40], making this technique effective in reducing I/O data volume across the entire model.

Figure 10 compares the timeline for completing three transformer layers with and without this technique. 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 probe heads' similarity exceeds the threshold, the time required to load only the keys from the probe heads (in red color) is significantly shorter, thereby reducing GPU wait time. Additionally, loading only a subset of keys for attention weight calculations reduces the time spent generating important token index sets (in green color), 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.

Setting the similarity threshold is complex. A threshold set too high might fail to reduce the number of keys and values across many transformer layers, as the average similarity of

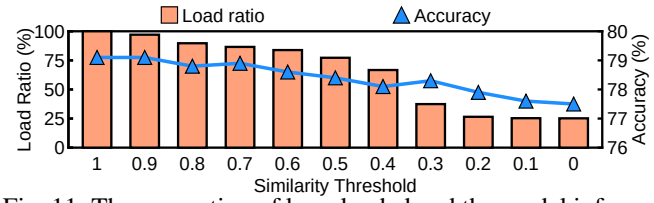
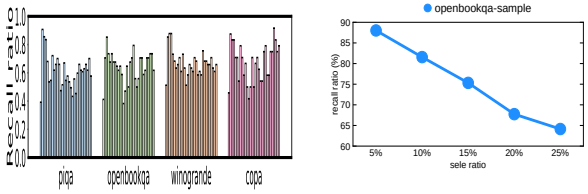


Fig. 11: The proportion of keys loaded and the model inference accuracy under different similarity thresholds.



(a) select the top 25% most important tokens (b) select different percentage most important tokens

Fig. 12: Similarities of important token index sets of adjacent layers.

the probe heads often falls short of the threshold. On the other hand, a threshold set too low can introduce bias in the identified important token index set, compromising model accuracy. Figure 11 illustrates this on the PIQA [7] dataset, where each transformer layer selects the top 25% of the most critical prefix KVs. Although decreasing the threshold from 1 to 0 cuts the number of keys loaded by 4×, it also results in a 1.6% decrease in accuracy, from 79.1% to 77.5%. Similar patterns are observed across other datasets.

To address this issue, we first calculate the expected value based on the proportion of selected important tokens, then slightly increase this value and use it as the similarity threshold. Specifically, suppose we have a prefix containing n tokens and need to select k important tokens ($k \leq n$). If we randomly execute this selection twice, we obtain sets A and B. The probability of each token appearing in both sets is $\frac{k^2}{n^2}$. Given N tokens in total, $E(A \cap B) = n \cdot \frac{k^2}{n^2} = \frac{k^2}{n}$, and $E(A \cup B) = E(A) + E(B) - E(A \cap B) = k + k - \frac{k^2}{n} = 2k - \frac{k^2}{n}$. Therefore, $E(\text{Jaccard}(A, B)) = \frac{E(A \cap B)}{E(A \cup B)} = \frac{k/n}{2-(k/n)}$. Denote this expected value as j. We set the threshold $t = j^{\alpha}$, where we empirically choose $\alpha = 0.6$ in our experiments to achieve a good balance between model inference accuracy and the amount of keys loaded.

C. Layer-Aware Speculative Prefetching

Observation III: Important token indices also exhibit strong consistency across adjacent layers of an LLM

For further investigation of the distribution of important token indices, we study the similarity of the index sets of the top 25% important tokens across adjacent layers for different models. As shown in Figure 12(a), we find that the distribution of important tokens is similar cross layers.

Based on observation III, We propose the layer-aware speculative prefetching mechanism. During the computation of the current layer, we prefetch important KVs into GPU memory for the next layer. This prefetching leverages the similarity of important token index sets across layers to improve prefetching

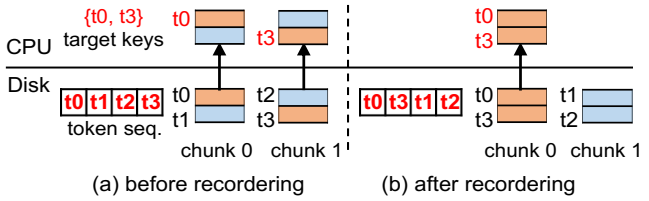


Fig. 13: Comparison of the number of chunks read before and after the token sequence is reordered. The orange (blue) rectangles represent important (unimportant) keys.

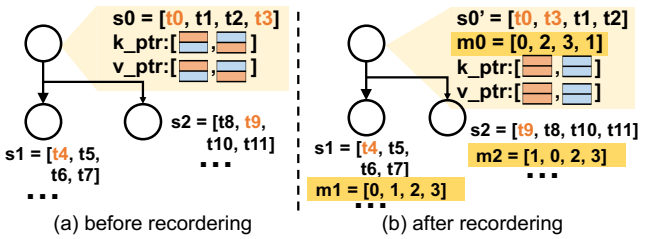


Fig. 14: Comparison of meta structure before and after KV reordering. The orange (blue) rectangles represent important (unimportant) keys.

accuracy. By approximating the important token index set of the next layer with the set from the current layer, we increase the accuracy of prefetching, leading to better performance compared to random prefetching. Once the important token index set for the next layer is identified, any important tokens' KVs that were not prefetched are then loaded into GPU memory for inference.

D. Importance-Informed KV Management

1) KV Reordering: We present the KV reordering method to address the unnecessary loading of unimportant KVs during important KV retrieval. By periodically reorganizing and repacking important KVs into denser chunks, this approach optimizes read efficiency and reduces bandwidth waste. Scheduled at regular intervals (e.g., every 10 minutes), this process is based on the average token importance and operates asynchronously to avoid disrupting the main I/O flow.

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 13(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 13(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.

Metadata adjustment. In LLMs, prefix KVs can only be reused if two prefixes share a common subsequence starting from the first token (i.e., the token order must be the same). Therefore, existing systems typically use a radix tree [40], [46] or its variants [13] to record stored prefix tokens, enabling quick search for reusable stored prefix KVs when a new request arrives. However, KV reordering may destroy the radix tree structure by altering the token order, causing new requests to fail in locating the correct shared prefix KVs. To overcome this, we limit the scope of KV reordering to the tokens within each node of the radix tree. Additionally, we introduce a mapping list within each node to assist the checking operations of new requests. We use an example to explain the details.

Consider two requests that retrieve related document segments as prefixes through RAG. Each prefix contains eight tokens: p0=[t0, t1, t2, t3, t4, t5, t6, t7] for one request and p1=[t0, t1, t2, t3, t8, t9, t10, t11] for the other. t0, t3, t4, and t9 are important data, while the remaining tokens are unimportant. Before applying the KV reordering technique, the radix tree is organized as shown in Figure 14(a), where the common prefix subsequence s0=[t0, t1, t2, t3] is grouped within the same node, enabling the reuse of as many prefix KVs as possible. Assume the chunk size is set to 2, with each chunk containing the keys or values of two consecutive tokens. Each node contains a list of pointers k_ptr (v_ptr) to these key (value) chunks.

After enabling KV reordering, as shown in Figure 14(b), the token sequence within each node is reordered in a descending order of importance, and the keys and values are repacked. In Node 0, for example, the token sequence becomes s0' = [t0, t3, t1, t2], with t0 and t3 now grouped within the same chunk. Consequently, reordering disrupts the token sequence in the radix tree. We add a new mapping list to Node 0 to address this issue. It is denoted as m0 = [0, 2, 3, 1], allowing the original s0 sequence to be recovered using the torch index operation s0'[m0] when search reusable shared prefixes for new requests. This vectorized indexing operation is highly efficient, consuming less than 2% of the TTFT in our experiments.

We explicitly avoid cross-node reordering, such as placing t4 and t9 into s0', for two key reasons. First, it would destroy the radix tree structure since t4 and t9 are not common tokens shared by both prefixes (i.e., p0 and p1), potentially leading to errors in retrieving reusable prefix segments for new requests. Second, it would result in unnecessary read bandwidth consumption by loading t9 when reusing the KVs of p0. The constraint against cross-node reordering prevents packing unshared tokens' KVs from different prefixes together, thereby reducing bandwidth wastage.

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

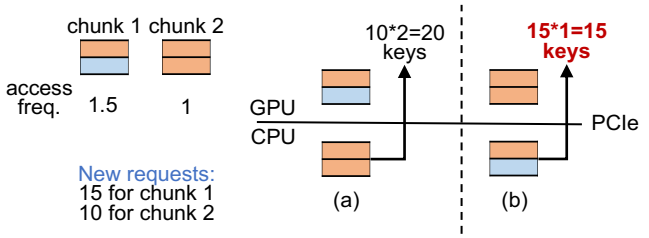


Fig. 15: Comparison of two cache replacement policies.(a) is the frequency-based cache replacement policy and (b) is the score-based cache management policy. The orange (blue) rectangles represent important (unimportant) keys. Only important keys are needed for new requests.

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 15(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 15(b), cutting the transfer of important keys to just 15.

Dual-cache replacement algorithm. We employ score-based cache replacement policy to oversee GPU and CPU caches, utilizing two min-heaps in CPU memory to manage chunks in both caches and facilitate eviction. The heaps' tops indicate the lowest-scored chunks in the GPU and CPU caches, respectively. To optimize data caching, we ensure non-redundancy between the GPU and CPU caches. Additionally, we maintain all chunk replicas on disk, thus eliminating I/O latency when chunks are evicted from CPU to disk.

Upon receiving a new request, HyperInfer first identifies reusable important tokens and locates their associated chunks. If the chunk is already in the GPU cache, HyperInfer utilizes the key or value vectors for inference and updates the chunk's score, retaining it in the GPU cache. If the chunk is in the CPU cache, after transferring the vectors to the GPU, HyperInfer updates and compares the chunk's score with the GPU cache's lowest. If superior, it replaces the lowest-scored chunk in the GPU cache; otherwise, it stays in the CPU cache. Should the chunk reside on disk, HyperInfer loads it into CPU cache, transfers the necessary vectors to the GPU, and updates the chunk's score. This new score is then assessed against the lowest scores in both caches to decide whether the chunk should be promoted to the GPU cache, remain in the CPU cache, or stay on disk.

V. Implementation

We chose to implement HyperInfer on top of FlexGen [33] 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 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 three open-source OPT models of different scales (i.e., OPT-6.7B, OPT-13B, and OPT-30B). Our experiments are performed on a server with 2 $\times$ 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 $\times$ 16.

Datasets and metrics. We select four representative datasets from the standard LM-Evaluation-Harness benchmark [7]: 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], [45], and vary the prefix KV retention ratios from 50% to 5% to observe accuracy changes. Additionally, to test TTFT with long prefixes as in [22] 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 four baselines. (1) *ReComp* [36]: It recomputes all prefix KVs for each request without storing or reusing them. (2) *AS-like* [6]: 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 [45]. Different from the AS-like, it asynchronously loads only important values rather than the full values of shared prefixes. (4) *AS+H2O+LFU*: It is similar to (3), but it uses LFU to manage the cache. In contrast,

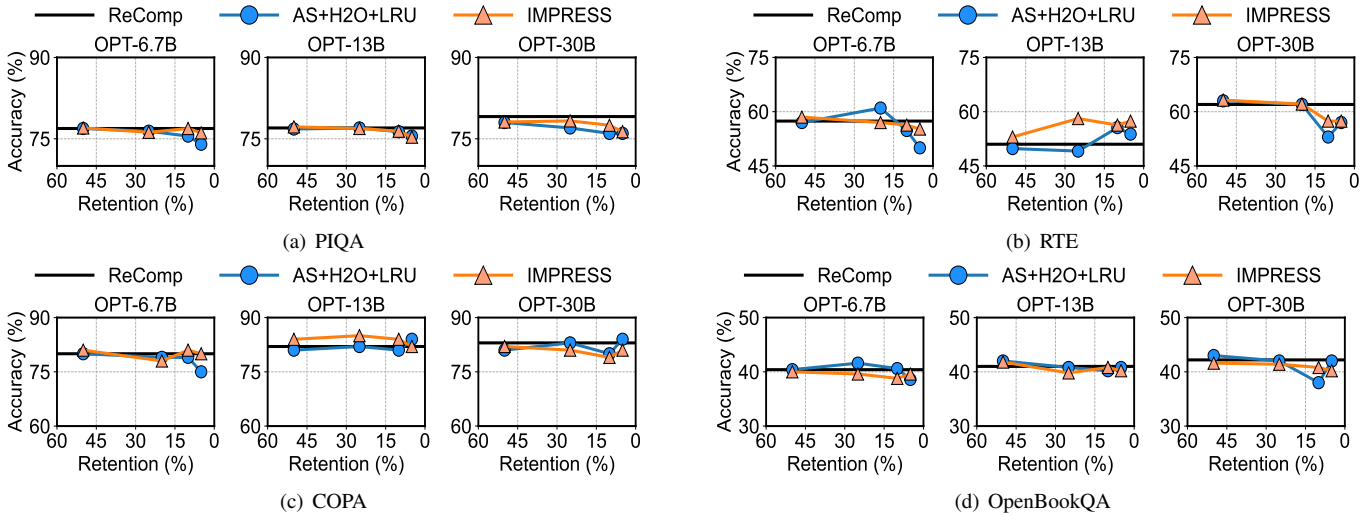


Fig. 16: Model generation quality of various systems across four datasets and three models.

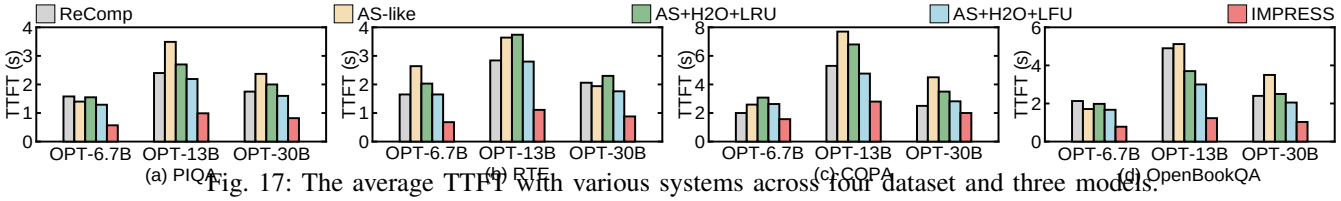


Fig. 17: The average TTFT with various systems across four dataset and three models.

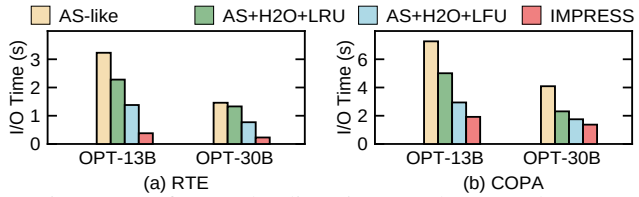


Fig. 18: Prefix KV loading time on the RTE dataset.

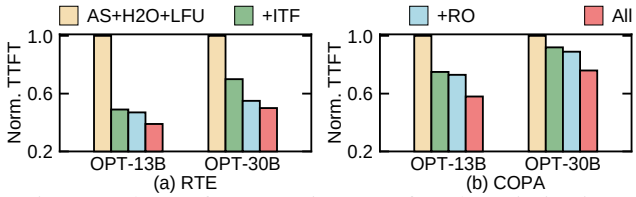


Fig. 19: The performance impact of each optimization.

HyperInfer selectively loads partial keys and values, reorders KVs, and uses a score-based cache management strategy.

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 [40].

B. Overall Performance

Model generation quality. Figure 16 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.

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 17 shows the average TTFT per request for different systems. HyperInfer outperforms alternatives, with a 1.2× to 2.8× improvement over leading solutions, due to a 1.5× to 3.8× reduction in I/O time for loading prefix KVs into GPU memory (Figure 18). 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, and HyperInfer are 3.9s, 9.3s, 6.6s, 5.9s, and 2.95s, respectively. This shows HyperInfer effectively reduces the tail I/O latency when KVs are loaded from SSD.

Fig. 20: The retention of KVs per model layer using similarity-guided important token identification.

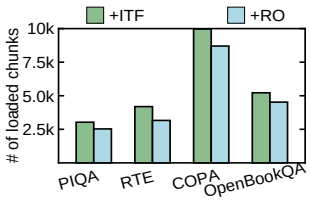


Fig. 21: The impact of KV reordering on the number of loaded chunks.

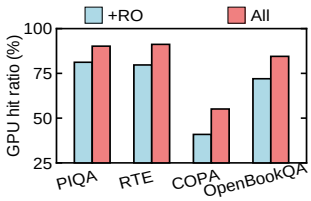


Fig. 22: The impact of scored-based cache management on GPU hit ratios.

C. Impact of Individual Techniques

Figure 19 shows the impact of each optimization. Using the current SOTA system *AS+H2O+LFU* as the baseline (TTFT normalized to one), *+ITF* adds similarity-guided important token identification for loading important KVs, *+RO* enables KV reordering on *+ITF*, and *All* incorporates score-based cache management on *+RO*. We observe that each optimization reduces TTFT, with *All* achieving the shortest TTFT, showing the effectiveness of HyperInfer's individual techniques. Besides, technique contributions vary across models and datasets. For example, in OPT-30B on RTE, the contributions of the three techniques are 60%, 30%, and 10%, while on COPA with OPT-13B, they are 36%, 8%, and 56%.

Figure 20 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 21 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 22 shows that score-based cache management boosts the average GPU hit ratio from 68% to 80% across four datasets, thereby reducing PCIe data transfers.

D. Sensitivity Analysis

Alpha value of similarity threshold. Figure 23 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 24 compares the TTFT of the leading system and HyperInfer with chunk sizes ranging from 16 to 256. Notably, HyperInfer achieves 2.2× to 2.4× 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.

Dataset size. We vary the number of prefixes (i.e., few-shot examples) to create different variants of OpenBookQA, with

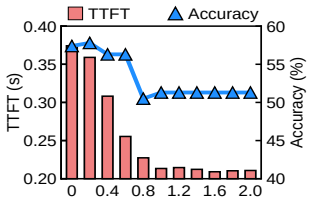


Fig. 23: Results of various alpha values.

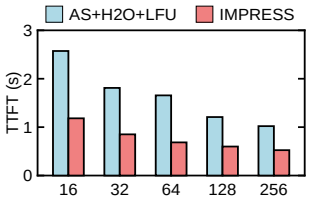


Fig. 24: Results of various chunk sizes.

dataset sizes ranging from 65GB to 400GB. Figure 25 shows that HyperInfer consistently outperforms the leading comparison system, AS+H2O+LFU, achieving speedups ranging from 1.2× to 2.0×.

Model type. Figure 26 shows the average TTFT for Llama2-7B and Llama2-13B models on PIQA and COPA. It demonstrates that HyperInfer achieves a 1.7×-2.7× speedup on Llama models, attributed to similar reasons in section VI-B.

E. 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. 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. 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

KV cache reuse. Some works [16], [29], [33], [36] accelerate the decoding phase by reusing KVs across iterations within a request. These are orthogonal to HyperInfer, which targets the prefill phase. Recent studies [6], [13], [15], [22], [40], [46] reuse shared prefix KVs across requests to reduce prefill latency (i.e., TTFT), but load entire prefix KVs, causing high I/O latency when KVs are on disk. In contrast, HyperInfer prefetches only important prefix KVs, reducing I/O latency. Other efforts [9], [39] explore finer-grained reuse of prefix KVs at the text segment level, which is orthogonal to HyperInfer and can be combined to enable more KV reuse.

KV pruning and quantization. Recent studies [18], [23], [33], [45] 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

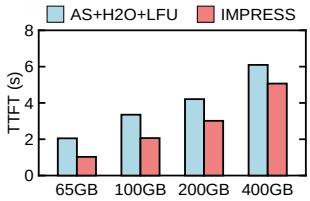


Fig. 25: Results on various dataset sizes.

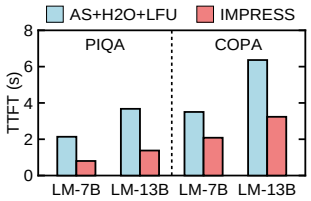


Fig. 26: Results on Llama (LM) models.

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 [11], [22], [24], [33], [42] focus on KV quantization to reduce bit counts per key and value element. They can complement HyperInfer to further decrease data load.

Other efficient LLM serving systems. Some works optimize other aspects of inference systems, such as request scheduling [1], [41], model parallelism strategies [20], [37], prefill-decoding decoupling [12], [14], [28], [34], [47], and distributed KV cache [21], [30]. These optimizations are also orthogonal to HyperInfer and can complement its improvements.

VIII. Conclusion

Existing prefix KV reuse systems do not always reduce TTFT, especially when disk I/O latency is involved in large-scale LLM services. We propose HyperInfer, a multi-tier prefix KV storage system to minimize I/O delay by only loading important KVs. Simply applying existing important token identification algorithms is suboptimal, as the reduction in I/O is limited. Therefore, we first introduce the I/O-efficient similarity-guided important token identification algorithm to identify important KVs with minimal I/O. Then, we propose importance-informed KV management to optimize storage and caching, further reducing TTFT. Our experiments show that HyperInfer reduces TTFT by up to 2.8× compared to state-of-the-art systems, while maintaining comparable inference accuracy.

Acknowledgments

We sincerely thank the anonymous reviewers for their constructive suggestions. This work was supported in part by the National Key Research and Development Program of China (2023YFB4502100), the National Science Foundation of China (62172361), the Major Projects of Zhejiang Province (LD24F020012), the Open Project Program of Wuhan National Laboratory for Optoelectronics (2023WNLOKF005), and the Pioneer and Leading Goose R&D Program of Zhejiang Province (2024SSYS0002).

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)*, pages 117–134, 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)*, pages 1742–1746. IEEE, 2023.

[3] G Bharathi Mohan, R Prasanna Kumar, Srinivasan Parathasarathy, S Aravind, KB Hanish, and G Pavithria. Text Summarization for Big Data Analytics: A Comprehensive Review of GPT 2 and BERT Approaches. *Data Analytics for Internet of Things Infrastructure*, pages 247–264, 2023.

[4] 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 25)*, pages 187–201, Santa Clara, CA, February 2025. USENIX Association.

[5] Sam Fletcher, Md Zahidul Islam, et al. Comparing Sets of Patterns with the Jaccard Index. *Australasian Journal of Information Systems*, 22, 2018.

[6] 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.

[7] 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.

[8] Alireza Ghadimi and Hamid Beigy. Hybrid Multi-Document Summarization using Pre-Trained Language Models. *Expert Systems with Applications*, 192:116292, 2022.

[9] 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)*, volume 6, pages 325–338, 2024.

[10] 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.

[11] 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.

[12] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. *arXiv preprint arXiv:2401.11181*, 2024.

[13] 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.

[14] Yibo Jin, Tao Wang, Huimin Lin, Mingyang Song, Peiyang Li, Yipeng Ma, Yicheng Shan, Zhengfan Yuan, Cailong Li, Yajing Sun, et al. P/D-Serve: Serving Disaggregated Large Language Model at Scale. *arXiv preprint arXiv:2408.08147*, 2024.

[15] Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-Throughput LLM Inference with Shared Prefixes. *arXiv preprint arXiv:2402.05099*, 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)*, pages 611–626, 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*, 388(13):1233–1239, 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)*, pages 155–172, 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 the Conference on Neural Information Processing Systems (NeurIPS)*, volume 33, pages 9459–9474, 2020.

[20] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 663–679, 2023.

[21] 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.

[22] 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)*, pages 38–56, 2024.

[23] 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 the Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pages 52342–52364, 2023.

[24] 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.

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

[26] 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 the Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pages 43447–43478, 2023.

[27] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. AttAcc! Unleashing the Power of PIM for Batched Transformer-based Generative Model Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 2, pages 103–119, 2024.

[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)*, pages 118–132, 2024.

[29] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently Scaling Transformer Inference. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 5, pages 606–624, 2023.

[30] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A KVCache-Centric Disaggregated Architecture for LLM Serving. *arXiv preprint arXiv:2407.00079*, 2024.

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

[32] 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)*, pages 12009–12024, 2023.

[33] 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 International Conference on Machine Learning (ICML)*, pages 31094–31116, 2023.

[34] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. DéjàVu: KV-Cache Streaming for Fast, Fault-Tolerant Generative LLM Serving. *arXiv preprint arXiv:2403.01876*, 2024.

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

[36] 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 the Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.

[37] 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. *arXiv preprint arXiv:2404.09526*, 2024.

[38] Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. A Brief Overview of ChatGPT: The History, Status Quo and Potential Future Development. *IEEE/CAA Journal of Automatica Sinica*, 10(5):1122–1136, 2023.

[39] 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.

[40] Lu Ye, Ze Tao, Yong Huang, and Yang Li. Chunkattention: Efficient Self-Attention With Prefix-Aware KV Cache and Two-Phase Partition. *arXiv preprint arXiv:2402.15220*, 2024.

[41] 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)*, pages 521–538, 2022.

[42] 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.

[43] 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.

[44] 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.

[45] 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 the Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pages 34661–34710, 2023.

[46] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. SGLang: Efficient Execution of Structured Language Model Programs. *arXiv preprint arXiv:2312.07104*, 2023.

[47] 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)*, pages 193–210, 2024.