

CatKV: Accelerating Position-Independent Caching with Context-Adaptive KV Cache Compression

JIE XU, Southern University of Science and Technology, China

RENJIE LIU, Southern University of Science and Technology, China

YI LI*, Beijing Institute of Technology, Zhuhai, China

BO TANG[†], Southern University of Science and Technology, China

Context caching is widely adopted to reduce the dominant prefill cost for long prompts across various large language model (LLM) applications. Recently, position-independent caching (PIC) has emerged to improve KV cache reuse beyond strict prefix matching. However, the required KV caches often reside in a persistent remote store in realistic deployments, making KV transfer the new bottleneck on time-to-first-token (TTFT). Existing KV compression techniques are primarily designed for long contexts. Since text chunks in PIC are much smaller, they fail to deliver sufficient compression and/or lead to substantial generation quality degradation.

In this work, we propose CatKV, an end-to-end PIC serving system for efficient KV cache reuse. Specifically, we devise a context-adaptive mixed-precision compression scheme in CatKV, which combines SVD factorization with adaptive scalar quantization in latent space. Given a target compression ratio, CatKV automatically determines the SVD and quantization configurations by analyzing the singular-value spectrum using the elbow method. To improve system efficiency, CatKV offloads online compression of cold text chunks to the CPU for asynchronous execution and exploits grouped factorization across chunks to further reduce storage consumption. Furthermore, CatKV introduces a six-stage layer-wise pipeline to reduce end-to-end latency. Extensive experiments demonstrate the superiority of CatKV across 3 PIC algorithms, 4 benchmarks, and 3 LLMs. In particular, CatKV achieves 2–5× faster TTFT and 6–10× higher compression ratios compared to baselines, all while preserving model accuracy. CatKV is open-source at <https://github.com/AlayaDB-AI/CatKV.git>.

CCS Concepts: • **Information systems** → **Data management systems**; • **Computing methodologies** → **Natural language generation**.

Additional Key Words and Phrases: KV Cache Compression; Position-Independent Caching; Inference Acceleration; Large Language Models;

ACM Reference Format:

Jie Xu, Renjie Liu, Yi Li, and Bo Tang. 2026. CatKV: Accelerating Position-Independent Caching with Context-Adaptive KV Cache Compression. *Proc. ACM Manag. Data* 4, 4 (SIGMOD), Article 266 (September 2026), 27 pages. <https://doi.org/10.1145/3837104>

1 Introduction

Large Language Models (LLMs) are widely deployed in real-world applications, e.g., personal assistants, question answering and healthcare [30, 31, 40, 55, 56, 62], but they often hallucinate when processing domain-specific queries beyond their internal training data. Retrieval-Augmented

*Work done during internship at AlayaDB.AI.

[†]Dr. Bo Tang is the corresponding author.

Authors' Contact Information: Jie Xu, Southern University of Science and Technology, Shenzhen, China, 12432713@mail.sustech.edu.cn; Renjie Liu, Southern University of Science and Technology, Shenzhen, China, liurj2023@mail.sustech.edu.cn; Yi Li, Beijing Institute of Technology, Zhuhai, China, 3120252145@bit.edu.cn; Bo Tang, Southern University of Science and Technology, Shenzhen, China, tangb3@sustech.edu.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2836-6573/2026/9-ART266

<https://doi.org/10.1145/3837104>

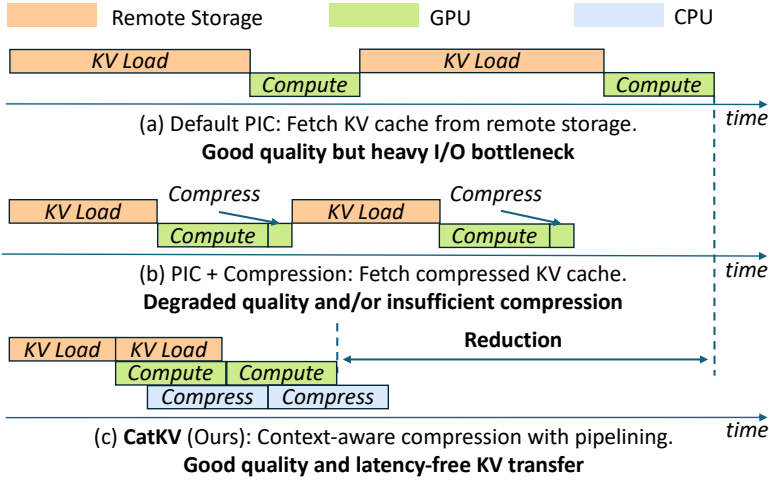


Fig. 1. Execution patterns of default PIC [2, 17, 21, 63, 65] with existing compression methods [5, 20, 36, 37, 64] and CatKV, where *KV Load* is the loading of remote KV caches, *Compute* is the prefill computation, and *Compress* is the compression of newly generated KV caches.

Generation (RAG) [15, 23, 24, 33, 42] addresses this by fetching external knowledge chunks, yet this incurs substantial computation in the prefill stage, where the generation of Key-Value (KV) cache becomes the dominant bottleneck for time-to-first-token (TTFT). While prefix caching techniques [9, 11, 27, 70] attempt to mitigate this by skipping redundant prefill tasks, their benefits are severely constrained by the strict requirement of identical token prefixes, a condition that rarely holds in dynamic RAG workloads characterized by multiple, unordered text chunks [17, 65].

To enhance KV cache reuse, emerging algorithmic approaches focus on relaxing matching constraints. Specifically, recent systems [2, 17, 21, 63, 65] propose *position-independent caching* (PIC), which directly reuses the KV cache of text chunks without requiring identical token prefixes, and selectively recomputes high-deviation KV caches to ensure high LLM generation quality (which we refer to as PIC correction). Empirical results demonstrate that this PIC scheme speeds up the prefill stage by $2\text{--}4\times$ while maintaining comparable generation quality to prefix caching.

While PIC increases the KV cache reuse ratio, this approach often requires persistently storing the historical KV caches of retrieved text chunks, substantial storage demands. This storage overhead is further amplified in industrial RAG deployments, where the massive scale of enterprise knowledge bases necessitates offloading KV caches to persistent remote storage tiers [4, 26, 48, 49]. For example, the entire MSMARCO[39] dataset contains ~ 600 million tokens and the total KV cache size reaches ~ 50 TB with Mistral-7B model. This far exceeds the capacity of a single machine. In this scenario, the KV cache required by a single user request, typically involving 10K to 30K context tokens, needs to be fetched from the remote storage before model generation can proceed.

This extra KV cache transfer from remote storage has not yet received much attention, as previous efforts primarily assume the KV caches of text chunks are always kept in the same GPU memory sharing the same context [27, 70] or can be quickly delivered via fast interconnects [8, 71]. Nevertheless, as subsection 2.2 will detail, this data movement overhead can be non-trivial for default PIC execution. As shown in Figure 1(a), when KV caches are stored in remote storage such as S3 [47] and EBS [46], KV cache loading becomes the clear bottleneck during LLM prefill and PIC correction stage due to the limited network bandwidth, thus slowing down the TTFT in LLM generation. As a result, PIC methods fail to deliver their desired benefits in real-world deployments.

To reduce the KV cache footprint, a common practice is to apply KV cache compression [5, 12, 20, 36, 37, 61, 64] by storing the KV tensors in low-precision representations. Existing compression approaches fall into three categories: (i) scalar quantization (SQ) [20, 37] which represents raw KV tensors in low-bit representations such as 4-bit and 2-bit precision; (ii) singular value decomposition (SVD) [5, 64] that employs SVD-based low-rank factorization to KV matrices and truncates along the latent channels; and (iii) bit compression (BC) [36] that applies quantization to delta-transformed cache values. However, these methods are primarily designed for long-context scenarios, whereas PIC workloads typically involve short, independent text chunks. In particular, SQ approaches are observed to degrade LLM generation quality when using a quantization bit-width of 4 bits or less [20, 37, 64]. Similarly, SVD and BC compression cause a performance drop beyond a compression ratio of $4\times$ [57, 69]. As a result, they are suboptimal under the new data distribution and can lead to substantial compression errors when facing outlier values. As depicted in Figure 1(b), they lack a sufficient compression ratio, making the LLM prefill still I/O-bound, and/or downgrade the model generation quality, failing to strike a good balance in between. Furthermore, as pre-computing KV caches for all text chunks in advance is impractical for a large knowledge base, they require to compress the KV caches online for *cold text chunks* (chunks that are not used before) and introduce extra GPU computation overhead.

Our solution CatKV. In this work, we systematically explore the correlation between compression ratio and model performance in PIC workloads. We analyze the limitations of pure SVD methods under small KV chunks and explore the potential opportunities for combining SVD with SQ techniques. Through extensive experimental study on the data distribution of SVD-decomposed factors, we empirically find that applying SQ on top of SVD with a few high-precision values can achieve a superior compression ratio while preserving model accuracy. Inspired by this insight, we propose CatKV, an end-to-end PIC-based system for efficient KV cache reuse from a remote KV store. To eliminate the I/O transfer bottleneck (see Figure 1(c)), CatKV includes a context-adaptive compression approach to represent KV cache tensors in mixed precision. Specifically, CatKV first decomposes the KV tensors by SVD factorization and applies adaptive SQ in latent space. A novel two-stage approach is devised to determine compression-related hyper-parameters based on the statistical distribution of each context. Given a target compression ratio, CatKV first determines the SQ groups by identifying the elbow-point of the singular value spectrum. Then, CatKV selects an appropriate bit width for each SQ group. With a concrete SQ configuration, CatKV finally determines the number of retained ranks for SVD.

We implement CatKV on top of vLLM [27], one of the most popular LLM serving frameworks. Based on this new compression scheme, CatKV dispatches the online compression task to the CPU for asynchronous execution, preventing it from blocking the LLM prefill kernels on GPU. Moreover, we employ grouped SVD factorization across different text chunks to further reduce the remote storage consumption. During end-to-end execution, CatKV also adopts a deliberate six-stage pipeline to squash the bubbles by overlapping data movement with CPU and GPU computation.

We conduct extensive experiments to verify the superiority of CatKV. The experimental results show that CatKV constantly speeds up the TTFT by $2\text{--}5\times$ and reduces the KV cache size by $6\text{--}10\times$, all while preserving the same generation quality across 4 benchmarks, 3 LLMs and 3 PIC algorithms.

In summary, the key technical contributions of this work are:

- We observe that existing compression methods lack sufficient compression and/or degrade the LLM generation quality in PIC workloads, and we systematically study the correlation between compression configurations and generation quality.
- We propose an end-to-end PIC serving system CatKV for efficient KV cache reuse from remote object storage.

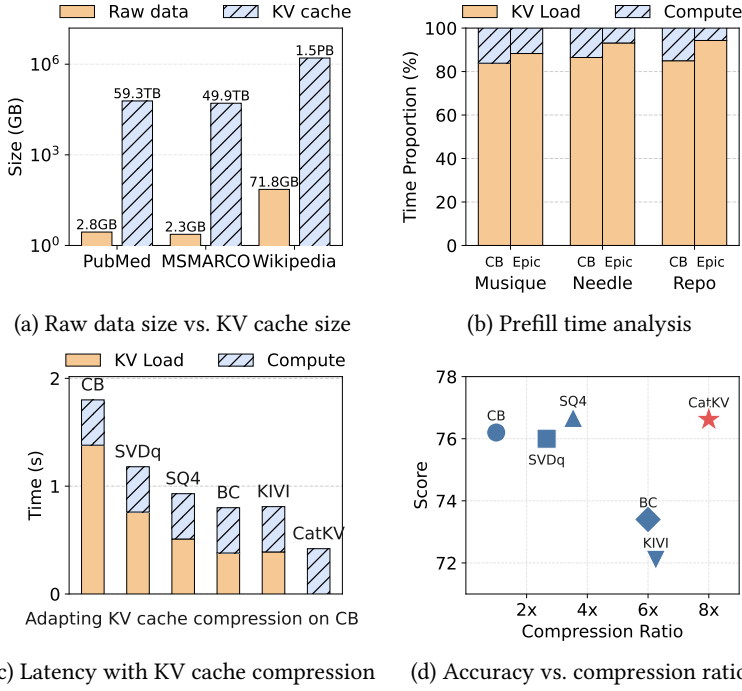


Fig. 2. Empirical analysis of storage and I/O bottlenecks in PIC serving systems, using Mistral-7B.

- We design a novel context-adaptive KV cache compression scheme with superior compression ratio to reduce KV cache transfer overhead while preserving the model accuracy.
- We devise a suite of system optimizations to further boost the performance of CatKV, including CPU asynchronous compression, grouped SVD factorization, and six-stage pipelining.

2 Background and Motivation

In this section, we first introduce the preliminaries of LLM in subsection 2.1, then we identify an overlooked bottleneck in existing PIC serving systems in subsection 2.2.

2.1 Preliminaries of LLM

2.1.1 Large Language Model Basics. Large Language Models (LLMs), such as GPT [1], Llama [18], and Qwen [62], are predominantly built upon the decoder-only Transformer architecture. Through auto-regressive generation, the inference process generates the next token t_{n+1} based on a sequence of input tokens $S = \{t_1, \dots, t_n\}$. Specifically, the model estimates the probability distribution of the next token t_{n+1} by propagating hidden states $H \in \mathbb{R}^{n \times d}$ through L transformer layers. At each layer l , $H^l \in \mathbb{R}^{n \times d}$ goes through an attention block and a feed-forward network, enabling the model to weight antecedent tokens and capture dependencies within a high-dimensional latent space.

2.1.2 KV Cache Reuse Strategy. Modern LLM serving systems reuse previously computed KV caches across requests to avoid redundant prefill computation and reduce TTFT. Based on their requirements for prefix matching, existing approaches can be categorized into (i) *Prefix Caching* and (ii) *Position-Independent Caching*.

Prefix Caching. Most systems such as vLLM [27], SGLang [70] directly use the precomputed KV cache of a text chunk to skip its prefill computation if the text chunk appears at the prefix of the LLM

input. This prefix matching follows the original mechanism in K and V generation, i.e., preserving the same positional encoding and cross-attention with previous tokens. As such, prefix caching does not affect LLM generation and the subsequent tokens will be identical to directly prefill on the entire prompt. Yet, due to this strict requirement, its reuse ratio is limited in long-context scenarios. For example, in RAG workloads that incorporate multiple text chunks into a single prompt, the KV caches for all chunks except the first cannot be reused, as they fail to satisfy the prefix-matching requirement for subsequent inputs [2, 65].

Position-Independent Caching. The strict prefix requirement in prefix caching substantially limits KV reuse in dynamic long-context workloads (e.g., RAG), where the same retrieved chunks may appear under different preceding tokens or in different orders. To reuse KV caches with different prefixes, recent systems [2, 17, 21, 65] adopt position-independent caching (PIC). However, PIC introduces incorrect cross-attention between text chunks, degrading the LLM generation quality. To eliminate the impact of incorrect positional encoding, they apply inverse transformation to remove the positional encoding when storing the precomputed KV caches and re-apply them during prefill. To tackle the loss of cross-attention, they propose a series of algorithms to identify the K , V matrices with high value deviation and apply dynamical recomputation them to restore correct attention scores.

2.1.3 KV Cache Compression. While reusing KV caches significantly reduces computation overhead, the accumulated K and V caches grow linearly with context length and quickly dominate the memory and storage capacity. To mitigate this massive memory footprint, compressing KV cache becomes a common practice. We classify the compression methods into three categories as follows.

Scalar Quantization (SQ). This method is widely used in KIVI [37] and KVQuant [20]. Round-to-nearest quantization is a standard form of uniform scalar quantization [20, 37]. Considering a n -dimensional KV tensor X , the B -bit integer quantization and dequantization process of $X[i]$ is formalized as:

$$Q(X[i]) = \left\lfloor \frac{X[i] - z_X}{s_X} \right\rfloor, \quad X'[i] = Q(X[i]) \cdot s_X + z_X, \quad (1)$$

where $z_X = \min X = \min\{X[1], \dots, X[n]\}$ is the zero-point, and $s_X = (\max X - \min X)/(2^B - 1)$ is the scaling factor, and $\lfloor \cdot \rfloor$ denotes the rounding operation.

KV cache quantization is typically performed in a group-wise manner, where elements sharing similar numerical characteristics are quantized using a shared scaling factor and zero-point. Prior work has shown that the choice of grouping dimension plays a critical role in preserving attention accuracy, as different dimensions exhibit distinct value distributions (especially ranges) and error sensitivity [37]. Specifically, *per-channel quantization* groups elements along the channel dimension, assigning one quantization scale to each channel. *Per-token quantization* groups elements along the token dimension, using a quantization scale for each token.

Singular Value Decomposition (SVD). It is commonly used to construct low-rank approximations of matrices, e.g., SVDq [64] and xKV [5]. Formally, for a given matrix $W \in \mathbb{R}^{m \times n}$, the decomposition is given by $W = U\Sigma V^\top$. Here, U and V denote the orthogonal matrices of left and right singular vectors, respectively, while Σ contains the corresponding singular values in descending order. By retaining only the top- r singular values, the approximation can be formulated as

$$W \approx AB, \quad A = U_r, \quad B = \Sigma_r V_r^\top, \quad (2)$$

where $U_r \in \mathbb{R}^{m \times r}$ and $V_r \in \mathbb{R}^{r \times n}$ correspond to the top- r singular vectors and $\Sigma_r \in \mathbb{R}^{r \times r}$ contains the top- r singular values. By applying it to the K and V matrices, this low-rank factorization reduces

the KV cache storage cost from mn to $r(m + n)$, where m and n denote the token count and total hidden dimension, respectively.

Bit Compression (BC). Recent studies, e.g., CacheGen [36] and LLM.265 [61], treat the KV cache as a continuous bitstream, optimizing for minimal bitrate at a given distortion threshold. These methods employ a non-linear or multi-stage encoding pipeline involving transformation, quantization, and entropy coding. Formally, given the input KV tensor X , an encoder $\mathcal{E}$ maps it into a compact bitstream S by exploiting statistical redundancies:

$$S = \mathcal{E}(X), \quad \hat{X} = \mathcal{D}(S), \quad (3)$$

where $\mathcal{D}$ is the corresponding decoder that reconstructs the approximation $\hat{X}$ from the stored bitstream S . These approaches typically leverage the data distribution and token-wise locality of KV caches to eliminate redundancy when encoding tensors into bitstream.

2.2 The Overlooked Performance Bottleneck

Persistent KV store in PIC. While PIC can significantly reduce the complexity in prefill computation, it requires a persistent KV cache store which does not fit in the storage capacity of a single machine. As shown in Figure 2(a), the storage demand for storing the KV cache of all 71.8 GB text chunks in the Wikipedia corpus reaches over 1.5 PB. Even on a smaller dataset, PubMed with raw 2.8 GB text size, its all KV caches require 59.3 TB storage. This is because, using Mistral-7B, the size of the KV cache for a single token is $2(KV) \times 32(Layer) \times 8(head) \times 128(dim) \times 2(BF16) = 128KB$. As a single token in text is mainly around 4 to 10 bytes, storing KV caches expands the data footprint by over 20K times. This large storage requirement far exceeds what a single machine can hold, even with high-capacity SSDs. As Wikipedia-scale dataset is common in industry, a more practical solution is to store them in persistent remote storage layer such as S3 [47] and EBS [46].

I/O bottleneck analysis in PIC. However, fetching the KV caches of matched text chunks from remote storage causes substantial I/O overhead. We tested two representative PIC algorithms, i.e., CacheBlend [65] (denoted as CB) and EPIC [21], on 3 benchmarks (Musique [54], NeedleBench [28], and RepoBench [34]), and reported the result in Figure 2(b). The KV caches are stored in a remote S3 storage with 10 Gbps bandwidth. Note that the KV loading time includes both KV transfer from remote to CPU memory, and the subsequent PCIe transfer to GPU memory. Yet, the PCIe time only contributes to less than 5% of the total KV loading time as the PCIe has a much higher bandwidth than inter-server network. The reported results confirm that remote KV loading contributes over 80% to the overall prefill latency, and the fraction can exceed 90% with EPIC. This is because EPIC further reduces prefill overhead via lightweight correction, thereby making the system heavily I/O-bound. Therefore, it is challenging to make PIC algorithms feasible as the I/O cost dominates the end-to-end prefill time.

Adapting KV cache compression methods in PIC. As a well-studied practice, KV cache compression methods can reduce the KV cache size, leading to less I/O traffic for fetching KV caches from remote. Inspired by them, we adapt existing KV cache compression methods to the PIC algorithm CacheBlend [65] (CB) in this testing experiment. Interestingly, we observed all existing compression methods have insufficient compression ratio and/or degrade the LLM generation quality, failing to strike a good balance between compression and accuracy. In Figures 2(c) and (d), we report the time decomposition, accuracy scores and compression ratio of different compression methods with CacheBlend (CB), evaluated on NeedleBench using Mistral-7B. In particular, we adapt 4-bit and 2-bit scalar quantization into CB, which are denoted as SQ4 and KIVI, respectively. For SVD-based compression methods and bit compression methods, we adapt SVDq [64] and CacheGen [36]

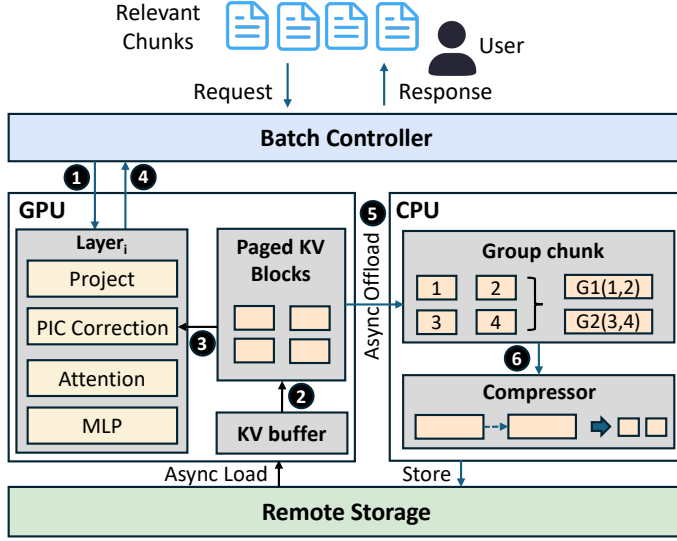


Fig. 3. Architecture overview of CatKV.

(denoted as BC in this experiment) into CB, respectively. The results in Figures 2(c) and (d) show that although KIVI and BC can reduce KV loading to approximately 50% of the total prefill latency, they degrade the model accuracy by 3% and 8%, respectively, which is typically unacceptable in production environments. Meanwhile, though SQ4 and SVDq do not pose significant impacts to the model accuracy, they have insufficient compression ratio (e.g., almost 4 $\times$), making I/O still a clear performance bottleneck.

Cracked GPU model computation. Apart from the unsolved I/O bottleneck by existing compression methods, they also face significant bubbled GPU model execution. This is mainly caused by online KV compression and multi-stage data transfers. Specifically, when a cold text chunk is used for the first time, its freshly generated KV cache will need to be compressed before being sent back to remote storage. A default way to do this is to compress the KV cache online after it is produced. However, as shown in Figure 1(b), this makes the GPU compression on the critical path, thereby blocking subsequent prefill kernels. Moreover, as previous solutions do not explore resource overlap under remote KV store, they sequentially trigger remote I/O, PCIe transfers and GPU kernels, incurring many fragmented operators blocking the prefill computation. As a result, they lead to low resource utilization and high latency.

3 CatKV Architecture Overview

To overcome the limitations of adapted solutions, we propose CatKV in this work, it is an end-to-end PIC serving system for efficient KV cache reuse. Figure 3 depicts the architecture overview of CatKV. It includes (i) a *context-adaptive mixed-precision compression* method combining SVD and SQ to eliminate I/O bottleneck; (ii) *CPU asynchronous compression* and *layer-wise pipelining* to avoid computation bubbles; and (iii) *grouped compression* to further reduce remote storage usage.

Specifically, there are two separate dependency paths in CatKV: $\textcircled{1} \rightarrow \textcircled{2} \rightarrow \textcircled{3} \rightarrow \textcircled{4}$, and $\textcircled{1} \rightarrow \textcircled{2} \rightarrow \textcircled{5} \rightarrow \textcircled{6}$. On the first path, incoming user requests and relevant text chunks are scheduled by the *Batch Controller*, which is implemented on top of vLLM [27] to support continuous batching [66]. When a request is scheduled for execution ($\textcircled{1}$), its required KV caches are fetched from remote storage, decompressed and stored in KV pages on GPU ($\textcircled{2}$). Then, the PIC correction

```

1 import CatKV
2
3 def inference(tokens, PIC_alg):
4     kv_state, cold_tokens = CatKV.fetch_kv(tokens)
5     input = embedding(cold_tokens)
6     for layer in layers:
7         q, k, v = project_kv(input)
8         k, v = CatKV.concat((k, v), kv_state)
9         k, v = CatKV.correct_kv(k, v, PIC_alg)
10        o = attention_func(q, k, v)
11    ...

```

Fig. 4. Three key APIs in CatKV.

module updates positional information and corrects biased KV tensors before sending them to the attention kernel (③). After completing generation for the current request (④), on the second path, CatKV offloads the KV cache of cold text chunks to CPU and executes our proposed compression algorithm in the backend (⑤). Grouped compression is used to further reduce the storage footprint at remote (⑥). Finally, compressed cold KV cache chunks are sent back to remote for future reuse.

PIC compatibility of CatKV. To support different PIC algorithms [2, 17, 21, 65] in a unified framework, we conduct an in-depth analysis on their execution pattern. In particular, we observe that these systems follow a similar workflow: they first concatenate the reused KV cache with freshly generated KV, then recompute a portion of the KV tensors to recover cross-attention, and finally use the corrected KV for attention computation.

Based on this, we provide three general APIs as shown in Figure 4. Specifically, upon receiving a request, CatKV first invokes `fetch_kv` to retrieve cached KV chunks from remote storage and determines the remaining cold text chunks. After the cold KV chunks are generated at each model layer, CatKV concatenates them and the cached KV chunks with the `concat` operator. Subsequently, the `correct_kv` performs KV recomputation to recover cross-attention and the corrected KV caches are fed to the subsequent attention kernel. Based on this unified paradigm, CatKV is extensible to all popular PIC algorithms, while providing unified system optimization on the backend, and we will verify it in section 6.

4 Context-adaptive KV Cache Compression

In this section, we first analyze the core reasons why SVD-based KV cache compression methods have insufficient compression ratio and degrade the LLM generation quality for PIC workloads in subsection 4.1, then we propose our context-adaptive compression scheme in subsection 4.2, and last we discuss the overhead, robustness and limitations of the proposed compression scheme in subsection 4.3.

4.1 SVD Factorization in PIC

The straight-forward way to use SVD for KV cache compression is to factorize the K and V matrices into latent space and retain only the top- r singular values. It is a common practice to compress KV cache under long-context scenarios [6, 51, 64, 69]. However, directly applying SVD factorization in PIC workloads is insufficient as it cannot achieve both good compression ratio and high LLM generation quality, simultaneously. The core reasons are as follows.

R1: Reduced token size per chunk. Existing SVD methods primarily focus on long-context inputs, i.e., $m \gg n$ as shown in Equation 2. However, in PIC, the number of tokens in a text chunk is similar to the channel dimension, i.e., $m \approx n$. This change leads to insufficient compression ratio using the same SVD rank configuration. For instance, when $m \gg n$, retaining 50% of the rank

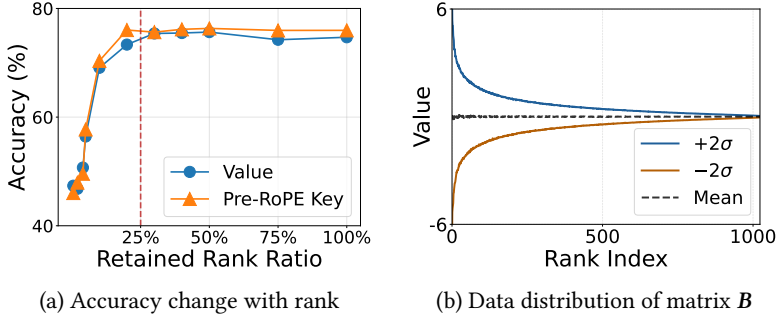


Fig. 5. Analysis of SVD factorization in PIC workloads, NeedleBench with Mistral-7B.

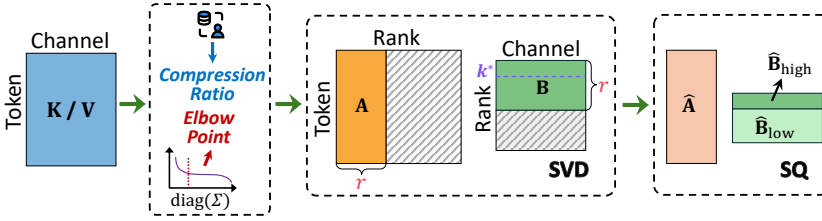


Fig. 6. The end-to-end compression workflow of CatKV.

brings around 50% reduction in size, as the size of matrix B is negligible to the size of A . However, when $m \approx n$, retaining 50% of the rank leads to negligible reduction in size because the size of B is almost the same as the size of A . It means that directly applying SVD leads to inadequate compression ratio in PIC workloads.

R2: Low-rank potential of KV cache. In Figure 5(a), we visualize the change in accuracy by varying the number of retained singular values of the V matrix and the pre-RoPE K matrix (i.e., the K matrix removing the positional encoding, RoPE [50]). The results present that in order to preserve similar model accuracy, the number of retained rank should be over 25%. In other words, to provide a good LLM generation quality, SVD factorization would only have up to $2\times$ compression ratio. Unfortunately, this is insufficient in practice and still incurs significant I/O bottleneck.

R3: Exponential decay along latent channels. As shown in Figure 5(b), both the value range and variance decay exponentially along the latent channels of the matrix B . It is reasonable because the Σ matrix has exponentially decaying singular values along the diagonal. When Σ_r multiplies with V_r^T , every singular value is multiplied along each row of V_r^T , propagating its distribution. The decay in the magnitude of B across latent channels reveals non-uniform channel importance, highlighting the suboptimality of vanilla SVD-based compression.

4.2 Context-Adaptive Compression Scheme

In CatKV, we propose a context-adaptive compression scheme to achieve both good compression ratio and high LLM generation quality, simultaneously. As illustrated in Figure 6, SVD is first applied to the original KV cache to obtain the A and B matrices, and then SQ is used to further compress the dense values into low-bit representations. This extra SQ stage provides two main advantages to KV cache compression: (i) it adds another layer of compression, further increasing the compression ratio; (ii) it relaxes the SVD rank configuration under the same memory budget, namely retaining more ranks, thereby better balancing compression ratio and model accuracy. To achieve this optimal balance, we first formally formulate the relationship between the target

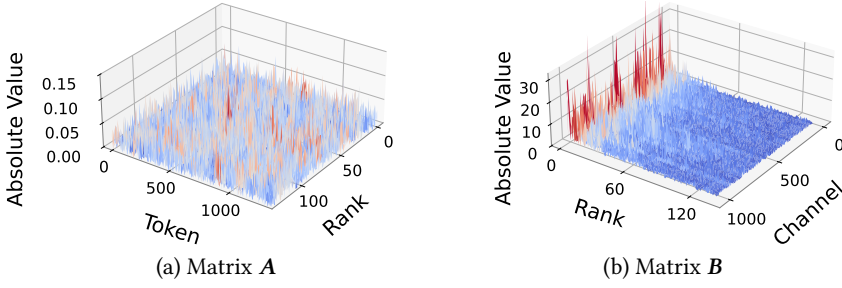


Fig. 7. Data distribution of low-rank factors for key cache.

Table 1. Accuracy and relative error under different SQ configurations using Mistral-7B. $\mathbb{C}$, $\mathbb{T}$ and $\mathbb{R}$ denote per-channel, per-token and per-rank quantization. Vanilla SVD and SVD+SQ use the same retained rank.

configuration	Relative Error ($\downarrow$, %)		Accuracy ($\uparrow$)
	Key	Value	NeedleBench
Vanilla SVD	3.42	12.89	76.96
SVD+SQ ($\mathbb{AT}$, $\mathbb{BC}$)	49.85	31.88	0.19
SVD+SQ ($\mathbb{AR}$, $\mathbb{BC}$)	48.97	32.25	0.35
SVD+SQ ($\mathbb{AT}$, $\mathbb{BR}$)	28.06	25.49	70.92
SVD+SQ ($\mathbb{AR}$, $\mathbb{BR}$)	26.09	25.89	72.21

compression ratio and its underlying parameters. Let the original KV cache be stored in b_0 bits per element, and the average bit-widths of the two quantized factors be $\bar{b}_{A_l}$ and $\bar{b}_{B_l}$. Discounting negligible metadata overhead, the aggregated compression ratio (CR) at layer l is formulated as:

$$CR_l = \frac{T \cdot D \cdot b_0}{(T \cdot \bar{b}_{A_l} + D \cdot \bar{b}_{B_l}) \cdot r_l}, \quad (4)$$

where T is the prompt token count, D is the total hidden dimension, and r_l is the number of retained ranks at layer l . Hence, maximizing generation quality for a target CR_l requires minimizing quantization error under this constraint. This is non-trivial, as it raises two key research challenges: (i) *along which dimensions should A and B be quantized to preserve subspace distributions?* and (ii) *how to allocate precisions ($\bar{b}_{A_l}$ and $\bar{b}_{B_l}$) to maximize fidelity?*

4.2.1 Determining the Optimal Quantization Dimension. To determine along which dimensions the A and B matrices should be quantized, we analyze their value distribution along different dimensions and visualize the results in Figure 7.

Regarding the A matrix, as depicted in Figure 7(a), we find that the value distribution is near-isotropic with no obvious pattern for outlier values, indicating that magnitude-based heuristics are insufficient. Therefore, we determine the SQ direction by evaluating the relative error of matrix reconstruction and end-to-end generation quality on PIC workloads. The results are reported in Table 1. It shows that applying *per-rank quantization* to A yields both lower relative error and higher accuracy compared to per-token quantization. This motivates us to use per-rank SQ for the A matrix to effectively preserve the original subspace distribution.

For the B matrix, as shown in Figure 7(b), the multiplication with singular values induces a significant reduction of the value magnitudes along the rank dimension. In this case, the leading ranks of V_r^T are multiplied with large singular values, see Equation 2, while other ranks form a

Table 2. Accuracy and relative error under different SQ configurations using Mistral-7B. (A) means SQ is applied to matrix A, same for B. The SQ direction is aligned with subsubsection 4.2.1.

configuration	Relative Error ($\downarrow$, %)		Accuracy ($\uparrow$)
	Key	Value	NeedleBench
Vanilla SVD	3.42	12.89	76.96
SVD+SQ (A)	7.64	18.42	75.66
SVD+SQ (B)	25.15	22.24	72.33
SVD+SQ (A, B)	26.09	25.89	72.21

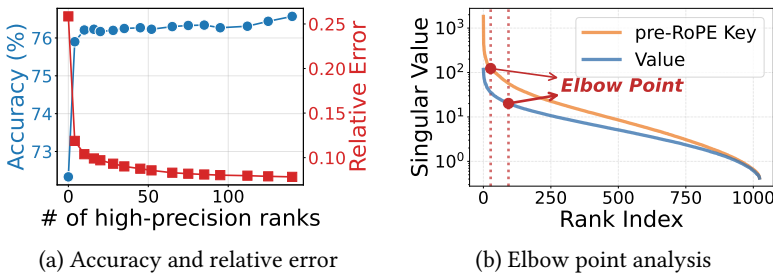


Fig. 8. Precision allocation on B.

long tail with much smaller values. As such, we adopt *per-rank quantization* on the B matrix to prevent the few outliers in the leading rank from affecting all other normal ranks. The relative reconstruction error and accuracy analysis of the B matrix in Table 1 also confirms our observation.

4.2.2 Automatic Precision Allocation. Having determined the quantization dimensions for both A and B, identifying the optimal $\bar{b}_{A_l}$ and $\bar{b}_{B_l}$ in Equation 4 remains challenging, as it requires maximizing the compression ratio without compromising model accuracy. A straight-forward way is to allocate a fixed precision like 4 bits to all elements in the A and B matrices. However, this simple allocation scheme incurs a significant accuracy drop, as shown in Table 2.

Opportunities for mixed precision. We conduct an in-depth analysis of the root causes of quantization error in Table 2. Specifically, the results present that it is the fixed quantization on the B matrix that leads to the high relative error after reconstruction. We find that the cause comes from biased estimation of the first few ranks in B. Following previous work [64], we assume the elements in each rank of B follow a uniform distribution. Then the expectation of the square absolute error ε for a given rank is $\mathbb{E}(\varepsilon^2) = \frac{1}{12} \frac{R^2}{2^{2b}}$, where b is the bit-width and R is the value range of all elements in a rank. Meanwhile, we observe that the variance of the B matrix along the rank dimension decreases dramatically after the first few ranks, as shown in Figure 5(b). This indicates that the leading ranks exhibit a significantly larger dynamic range, thereby contributing to the majority of the estimation errors. Therefore, it inspires us to preserve original precision for the first few ranks in the B matrix, thus, to reduce the error associated with SQ along the rank dimension.

Figure 8(a) illustrates the effect of mixed precision on model accuracy, which exhibits a clear convexity. In particular, we observe that keeping just the first few dominant ranks in high precision helps reduce the reconstruction error significantly and recover most of the model accuracy loss. Moreover, with more ranks being stored in the original precision, the benefit starts to shrink. This diminishing return suggests the existence of a point which is good enough to compensate

Algorithm 1: Context-Adaptive Compression in CatKV

Input : Key/Value matrix X , Target compression ratio CR, bits $b_{\text{high}}, b_{\text{low}}$, Chunk length T , Hidden size D

Output: $\hat{A}, \hat{B}_{\text{high}}, \hat{B}_{\text{low}}, k^*$

$U, \Sigma, V^T \leftarrow \text{SVD}(X); S \leftarrow \text{diag}(\Sigma)$

$k^* \leftarrow \text{KNEEDLE}(S)$ // Find maximum curvature point

$r \leftarrow \text{Equation 4}$ // Calculate retained rank

$A \leftarrow U_r; B \leftarrow \Sigma_r V_r^T$ // Truncate to rank r

$\hat{A} \leftarrow \text{QUANTIZE}(A, b_{\text{low}})$

$\hat{B}_{\text{high}}, \hat{B}_{\text{low}} \leftarrow \text{QUANTIZE}(B, b_{\text{high}}, b_{\text{low}}, k^*)$

return $\hat{A}, \hat{B}_{\text{high}}, \hat{B}_{\text{low}}, k^*$

compression ratio for model accuracy. As such, this motivates us to find a proper configuration on the precision allocation to achieve a desired tradeoff in between.

Precision allocation algorithm. The point starting from which the curve starts to stabilize is often referred to as the *Knee* or *Elbow* point of a curve. The brute force way to search this point is to evaluate the validation accuracy for every possible rank index and select the point with the largest diminishing return. However, it is impractical due to its prohibitively high computational and latency cost. To efficiently identify this point, we instead search for the elbow point on the singular value spectrum $S = \{\sigma_1, \dots, \sigma_R\}$ across all ranks, see Figure 8(b). Specifically, as discussed before, the quantization error is directly related to the magnitude of the singular value. Therefore, the elbow point of the singular value curve can serve as an effective approximation for the optimal configuration on the model accuracy curve. We employ the Kneedle algorithm [45] for a lightweight implementation to automatically detect this elbow point k^* on the singular value spectrum. As a result, the selected k^* partitions the ranks into two groups: the leading ranks $\{1, \dots, k^*\}$ are quantized with full precision b_0 to preserve dominant signals, while the subsequent ranks $\{k^* + 1, \dots, k_R\}$ are quantized to b_s to maximize compression. Empirically, we find that using 4-bit quantization for the b_s is sufficient for compression while not affecting the model accuracy.

Pseudo-code of Context-adaptive compression in CatKV. Algorithm 1 summarizes the end-to-end workflow of our proposed context-adaptive compression approach in CatKV. Specifically, upon obtaining the KV cache of a text chunk, it first performs approximate SVD to get U, Σ and V^T via `torch.svd_lowrank()`. The initial rank is determined by the target compression ratio with an aggressive 4-bit quantization. The elbow point k^* is then identified using the Kneedle algorithm on Σ to separate high-precision and low-precision rank groups. Combining the target compression ratio in Equation 4, the final rank r is then computed to generate the A and B matrices. Last, the two matrices are quantized with mixed precision. In online decompression, the compressed matrices are dequantized and multiplied to reconstruct the original KV cache.

4.3 Discussions

Relations to SVDq. Although SVDq also uses mixed-precision SQ on top of SVD, our method differs from SVDq in two ways: (i) SVDq compresses only the product of U_r and Σ_r and lacks analysis on the V_r matrix, which leads to insufficient compression in PIC scenarios; and (ii) SVDq adopts a manually fixed precision allocation to control quantization error, which in practice can not achieve the desired compression-accuracy tradeoff. In contrast, our approach determines the

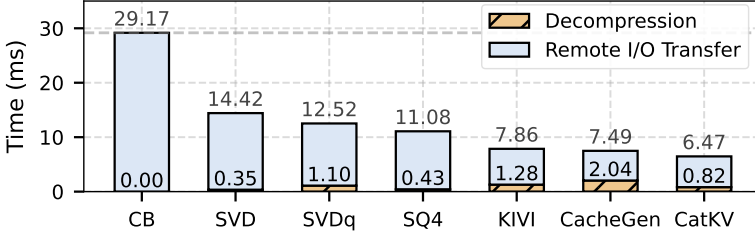


Fig. 9. Single-layer latency comparison using Mistral-7B.

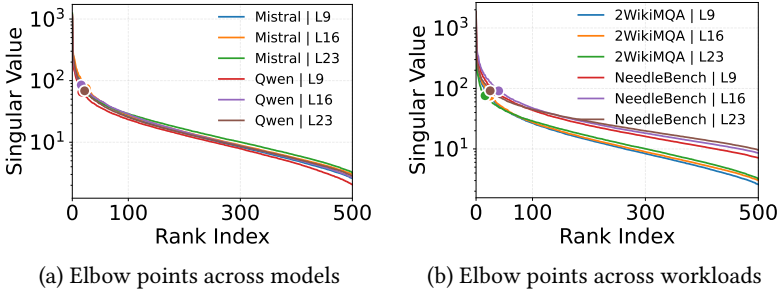


Fig. 10. Elbow points across models, layers, and workloads.

precision boundary adaptively based on the singular value spectrum of the specific prompt, enabling a more flexible and tailored allocation of quantized precisions.

Overhead of mixed-precision quantization. We compare the single-layer latency, including both remote I/O transfer cost and decompression cost, across different solutions. Each layer has 10 text chunks and each chunk has 512 tokens, and all decompression process are conducted on GPU. The results are depicted in Figure 9. While CatKV has slightly higher decompression time (0.82 ms) compared to uniform-precision baselines, i.e., SQ4 (0.43 ms) and SVD (0.35 ms), its superior compression ratio effectively reduces the remote I/O transfer cost. Consequently, the reduction in I/O transfer time fully offsets the decompression overhead, allowing CatKV to achieve the lowest overall latency.

Robustness and stability of elbow. Figure 10 reports the elbow points of the singular value spectrum across different models, layers, and workloads. The results show that the k^* remains stable across various configurations, suggesting the strong generalizability and robustness of our proposed elbow-based method. Note that in extreme cases such as ultra-high compression requirements where the singular value spectrum lacks a valid knee, practices like setting a predefined threshold could be adopted as practical fallbacks.

Limitations of context-adaptive compression in CatKV. While our proposed context-adaptive compression scheme in CatKV is effective in general, its superior compression performance depends on the inherent low-rank structure of the KV caches. For emerging model architectures where this low-rank property possibly could be weak, CatKV may yield diminishing returns, limiting the achievable compression ratio.

5 CatKV System Optimizations

In this section, we improve the performance of CatKV by devising a suite of system optimization techniques.

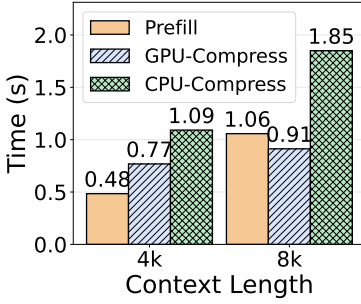


Fig. 11. Execution time of CPU vs GPU compression across context lengths.

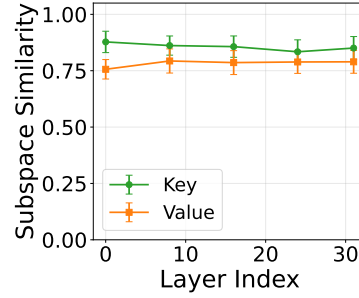
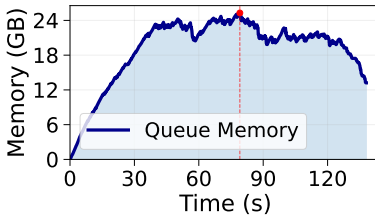
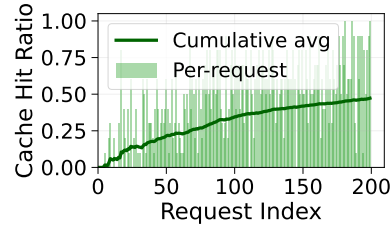


Fig. 12. Subspace similarity between local chunks and global KV matrices.



(a) Queue memory footprint



(b) Cache hit ratio

Fig. 13. CPU Compression queuing behavior in CatKV.

5.1 CPU Asynchronous Compression

In a large external knowledge base, it is usually impractical to pre-compute the KV caches of all text chunks beforehand due to substantial preprocessing overhead. A more feasible approach is to instantiate the KV cache on demand when a cold text chunk is included in the prompt, and send the compressed KV caches back to the persistent remote storage for future reuse. Here we denote the cold text chunks as the chunks that have *not* been used by any request before. As cold text chunks trigger online KV cache compression after the prefill computation, it blocks the execution of subsequent GPU model computation kernels and introduces extra latency on the critical path. As shown in Figure 11, the prefill computation takes 1.06 seconds on an 8K-token context, whereas our context-adaptive compression incurs an additional 0.91 seconds time cost, slowing down the end-to-end model computation by 90%.

To eliminate this extra overhead, we propose to decouple the online compression from the critical path of model computation by offloading the compression operations to CPU. As illustrated by the path ① → ② → ⑤ → ⑥ in Figure 3. Specifically, CatKV maintains a $(start, end)$ index tuple for each cold text chunk, marking out the index range of its KV cache in all tokens' large KV matrices. Upon completion of each model layer's prefill stage, CatKV slices the large KV matrices using the index tuples to obtain a list of generated KV caches for each cold text chunk. Then the sliced KV caches are asynchronously offloaded to CPU memory through PCIe in a separate CUDA stream. This allows CatKV to seamlessly launch the prefill computation of the next layer without blocking. On the CPU side, sliced KV caches are captured by a *producer* thread which pushes each KV cache chunk into a queue. To maximize CPU utilization, a pool of *consumer* workers fetches the KV caches from the producer queue and conduct online compression. By compressing KV caches of

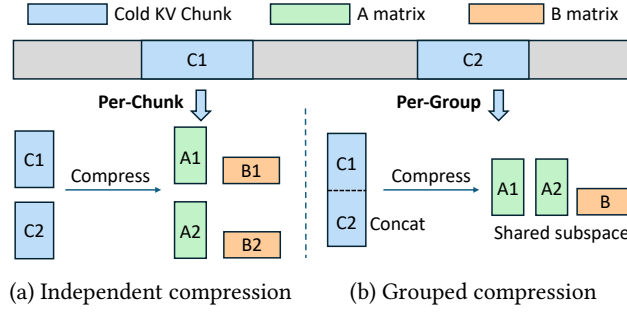


Fig. 14. Grouped KV cache compression in CatKV.

cold text chunk asynchronously on the CPU, we prevent it from blocking the GPU computation and thus speed up the end-to-end latency in the critical path.

Another advantage of the asynchronous CPU compression is that it can serve as a layer of local KV cache to mitigate the I/O overhead of loading KV cache from remote storage for future prompts. In particular, we maintain a local hash map on the CPU to determine whether a requested KV cache chunk exists in current GPU memory waiting for compression. If there is a cache hit, we directly return the KV cache in memory as the result without sending the actual I/O request, further reducing the I/O traffic and latency. However, compressing the KV cache of a text chunk on CPU takes more time than the model computation, as shown in Figure 11. It means that compression tasks may accumulate in the task queue on CPU and possibly exceed the available CPU memory capacity. To investigate whether this could become a system bottleneck, we test 200 requests from the 2WikiMQA dataset at a request rate of 2 req/s with the Mistral-7B model, and we use 32 CPU cores for asynchronous compression. Figures 13(a) and (b) show the memory footprint and cache hit ratio on CPU, respectively. It confirms that as more compression tasks reside on CPU, the cache hit ratio increases and thus fewer new cold chunks come in. This allows the CPU workers to gradually keep pace with the workload over time and maintain the queue length within a stable, bounded range. As such, the hash map-based caching significantly reduces the memory footprint of CPU compression and prevents it from blocking the GPU inference kernels. In extreme scenarios, e.g., constrained CPU memory or degraded cache hit rates, gracefully aborting compression jobs upon out-of-memory (OOM) and re-triggering compression on future cache misses serves as a viable fallback mechanism.

5.2 Grouped KV Compression

As discussed in subsection 5.1, a single request may include multiple cold text chunks for generation, and their generated KV caches need to be compressed before sending to the remote storage. Inspired by the fact that retrieved text chunks within a single RAG prompt exhibit semantic coherence, we explore whether they also share a common latent space after SVD factorization. Note that we use KV chunk to indicate the KV caches of a text chunk hereafter. In Figure 12, to evaluate the subspace similarity, we compute the principal angles between the B matrix of a single KV chunk and those of all KV chunks across model layers. The results show that both the B matrices of the K and V matrices in a single KV chunk show a high similarity with the B matrix generated using all KV chunks. This motivates us to compress the cold KV caches in a group to further reduce the storage demand.

Figure 14 depicts the proposed *grouped compression* and its comparison with the *independent compression*. Specifically, independent compression applies SVD factorization to each cold KV chunk separately, generating A and B matrices for each KV chunk. Considering the similarity of the B

Table 3. CPU compression time w/ and w/o chunk grouping.

Length	Method	Layer	Chunk	Request
4K (4×1024)	w/o Group	0.06s	0.28s	1.01s
	w/ Group	0.08s	—	1.06s
8K (8×1024)	w/o Group	0.08s	0.29s	1.84s
	w/ Group	0.14s	—	1.80s

Table 4. Average relative error using Mistral-7B.

Dataset	Matrix	w/o GC	w/ GC
NeedleBench	Key	7.93	8.94
	Value	18.40	22.02
2WikiMQA	Key	7.87	9.70
	Value	16.86	24.40

matrix across KV chunks in a single request, independent compression leads to redundant matrices and thus requires a higher storage demand. As such, to utilize the latent space similarity, grouped compression first concatenates all cold KV chunks, for example a group of s chunks, along the token dimension, leading to a large $\mathbf{K} \in \mathbb{R}^{(m \cdot s) \times n}$ and $\mathbf{V} \in \mathbb{R}^{(m \cdot s) \times n}$. On the large concatenated matrix, SVD factorization is applied to generate two matrices $\mathbf{A} \in \mathbb{R}^{(m \cdot s) \times r}$ and $\mathbf{B} \in \mathbb{R}^{r \times n}$ by retaining only the top- r singular values. Then, as previously introduced in subsubsection 4.2.2, mixed-precision SQ is applied to each $\mathbf{A} \in \mathbb{R}^{m \times r}$ in $\{\mathbf{A}_1, \dots, \mathbf{A}_s\}$ and one $\mathbf{B}$ matrix with automatic precision allocation to form the final compressed matrices. By this means, we only need to store one $\mathbf{B}$ matrix for all $\mathbf{A}$ matrices in the group. Considering $m \approx n$ for each KV chunk, this grouped compression leads to $2s/(s+1)$ times storage reduction compared to independent compression.

This grouped compression also reduces the I/O traffic during online KV loading. If a future prompt requires multiple KV chunks that share the same $\mathbf{B}$ matrices, the $\mathbf{B}$ matrix only needs to be loaded once from remote storage. Empirical results on the 2WikiMQA dataset show that among cache-hit requests (i.e., requests that reuse at least one cached KV chunk), 8% share at least two KV chunks, suggesting opportunities to share the retrieval of their $\mathbf{B}$ matrices. To this end, we incorporate *I/O request deduplication* in CatKV. Specifically, before sending each batch's I/O request, CatKV performs a set operation on the I/O read requests for $\mathbf{B}$ matrices, and by using the matrix id as identifier, multiple requests to the same $\mathbf{B}$ matrix will be combined into one single request. As such, fewer I/O requests will be sent, thus further reducing the online I/O overhead. Moreover, CatKV maintains a reverse map on the GPU to link each $\mathbf{B}$ matrix with its corresponding $\mathbf{A}$ matrix to achieve accurate online decompression.

Grouped compression overhead. Table 3 reports the compression latency with and without grouped compression. The results show that, though the per-layer compression time increases with chunk grouping, the overall request-level latency remains stable as the total workload is unchanged. That is to say, while grouped compression leads to a heavier load of one single compression task, the number of tasks decreases and the total computation cost for all layers remains identical. As such, grouped compression does not impose extra overhead to CPU execution.

Impact on accuracy. To explore whether grouped compression introduces additional estimation error and potentially influences LLM generation, we evaluated the relative error of the reconstructed KV matrices for both grouped and independent compression. As reported in Table 4, the results

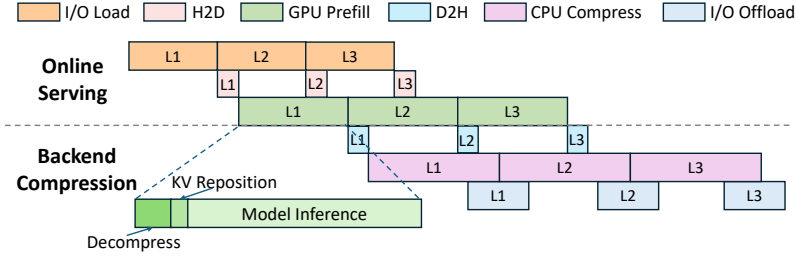


Fig. 15. Six-stage layer-wise serving pipeline in CatKV.

reveal distinct behaviors for the Key and Value caches. Specifically, we observe that grouped compression maintains a high fidelity for the Key cache, with a reconstruction error of approximately 10%. In contrast, it has a more pronounced impact on the Value cache, where the relative error increases to 10–30%. This indicates that the Key cache is more robust than the Value cache to our proposed grouped compression. Therefore, to preserve similar LLM generation quality, we only apply grouped compression to the Key cache while still using independent compression to the Value cache. However, we note that in scenarios facing extreme storage bottlenecks, extending grouped compression to the Value cache remains a practically useful trade-off at the cost of a marginal increase in reconstruction error.

5.3 Layer-wise Asynchronous Pipeline

The default method to conduct model serving is to process different batches and layers sequentially; for each input prompt, the system first loads its required KV cache chunks from remote storage, then transfers the concatenated KV caches to GPU, and finally performs prefill and decode computation to generate output tokens. After the final answer is generated, the system offloads the cold KV cache to CPU for asynchronous compression and meanwhile starts processing the next prompt. This is inefficient because the GPU model computation needs to wait for its slow remote I/O and PCIe transfer. To enable fine-grained scheduling to fully saturate GPU on the critical path, we implement a deliberate pipeline to overlap I/O and computation across consecutive LLM layers. The key difference from previous layer-wise pipelines [9, 35, 43] is that we simultaneously handle both the receiving I/O for compressed KV caches from remote storage and the sending I/O for cold KV chunks to CPU memory.

Figure 15 depicts the six-stage serving pipeline of CatKV. Specifically, for one batch of prompts, on receiving the remote KV cache chunks of the first layer, the CPU launches the model computation kernels on the GPU and meanwhile sending I/O requests and host-to-device (H2D) transfers for the next layer. Since we empirically observe that the KV decompression and repositioning procedures are lightweight on GPU (incurring less than 10% overhead, see the highlighted green in Figure 15), we place them sequentially right before the transformer layer computation in the same CUDA stream. After generating this layer’s full KV caches, the cold KV chunks are sent to CPU in a separate stream while the GPU continues to perform the decomposition and model computation of the next layer. When complete cold KV chunks are received on CPU memory, the asynchronous CPU compression engine then performs at the backend and then sends them back to the remote storage as discussed in subsection 5.1.

This fine-grained six-stage pipeline ensures concurrent processing for different components across transformer layers. For example, when the model computation is executing layer l , the KV cache of layer $l + 1$ is transferred to GPU through PCIe, the KV cache of layer $l + 2$ is fetched from remote, and the generated cold KV cache chunks of layer $l - 1$ is sent back to CPU memory for asynchronous compression. Since the network and PCIe transfers are full-duplex, CatKV utilizes

the full bandwidth of both downloading and uploading without interference with each other. As such, this pipeline allows CatKV to overlap remote I/O, PCIe data transfer and GPU computation, by which the model computation can more effectively saturate GPU resources throughout the whole serving process.

6 Experimental Evaluation

In this section, we conduct extensive experimental evaluation on CatKV to verify its superiority over the competitors. In particular, there are three questions to answer:

- *Does CatKV affect model accuracy in various PIC workloads?*
- *Can CatKV achieve better compression ratio than baselines?*
- *Can CatKV outperform existing solutions in terms of TTFT?*

6.1 Experimental Setup

Models and hardware settings. We evaluate CatKV across 3 LLMs: Mistral-7B [53], Llama 3.1-8B [18], and Qwen 2.5-14B [62]. All experiments were conducted on a server equipped with two NVIDIA RTX 4090 GPUs, 1TB memory and two AMD EPYC 9554 64-Core CPUs. The server connects to remote S3 [47] storage with 10 Gbps Ethernet and the PCIe bandwidth connecting CPU and GPU is 22 GB/s.

Datasets and workloads. Our evaluation covers the following types of workloads and benchmarks:

- *Multi-hop QA:* This workload requires combining facts from multiple chunks to generate the answer. We sample 200 questions from 2WikiMQA [19] and 150 questions from MuSiQue [54] for the complex reasoning scenario.
- *Long-context Retrieval:* This workload evaluates the model's ability to locate information in extensive contexts. We sample 200 tasks with varying levels of information density from NeedleBench [28] to simulate the needle-in-a-haystack scenario.
- *Repository-level Code Intelligence:* We sample 300 tasks from RepoBench [34] to evaluate the model in large software projects. This requires the model to leverage the contexts of multiple files and understand their dependencies for code completion.

Metrics. We compare CatKV with alternative systems by:

- *Time-To-First-Token (TTFT):* TTFT measures the latency from the input prompt to the generation of the first token. It is crucial in real-time interactions for a responsive user experience.
- *Compression Ratio (CR):* CR is the ratio of the original KV cache size to the compressed size. A higher CR indicates a greater reduction in storage and remote I/O traffic.
- *F1-score:* We use the F1-score in 2WikiMQA, MuSiQue and NeedleBench to measure the token overlap between the generated response and the ground truth.
- *Code Similarity Score:* For code generation tasks in RepoBench, we use code similarity score to evaluate the similarity between the generated code snippets and the reference implementation, capturing both structural and semantic alignment.
- *Exact Match (EM):* EM is a stringent metric requiring the output to exactly match the ground truth, used as a rigorous check to ensure no semantic degradation.

Competitors. We evaluate 3 popular PIC methods: CacheBlend [65], EPIC [21], and KVShare [63]. CacheBlend recomputes KV caches based on KV deviation, EPIC recomputes a fixed number of tokens at the prefix, and KVShare uses the accumulated deviation of the attention scores to identify

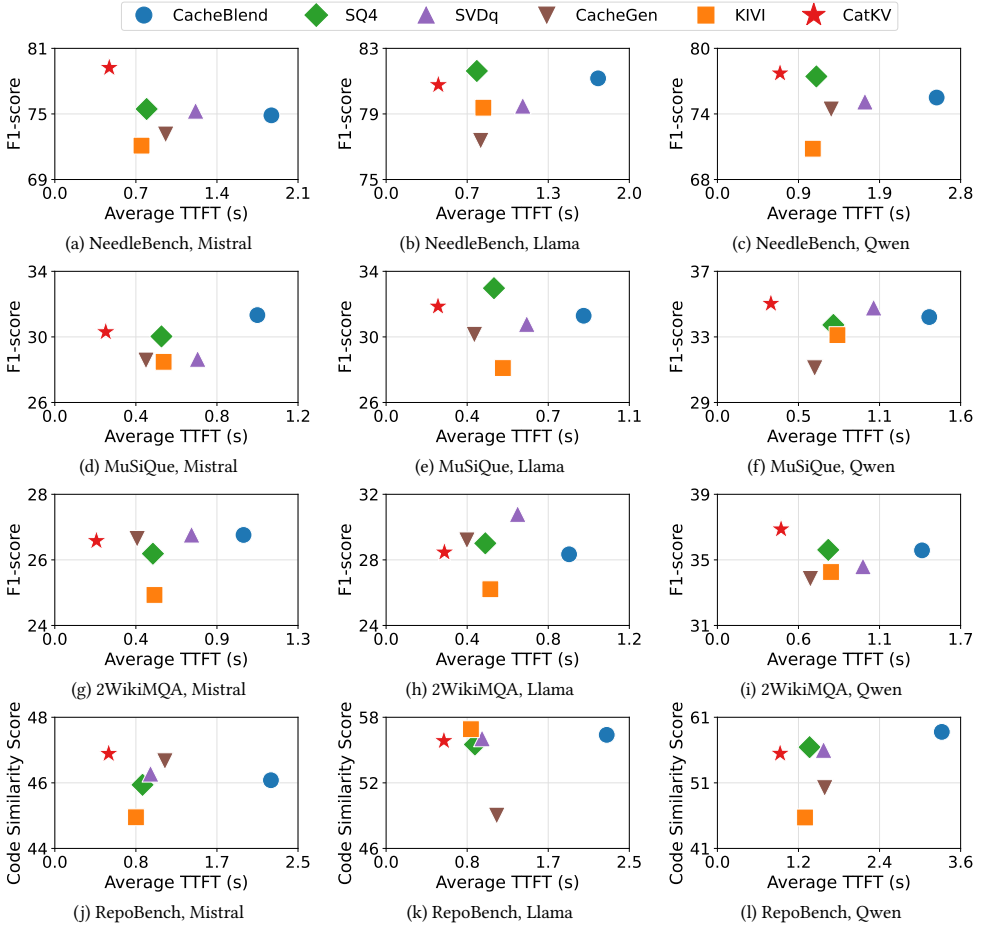


Fig. 16. TTFT and score of CatKV and baselines. CatKV delivers 2–5× lower TTFT than the default PIC solution while preserving the original model accuracy.

incorrect tokens. We compare CatKV with their default implementation and variants by adding representative KV cache compression algorithms:

- **KIVI** [37]: KIVI is a representative SQ method, we use KIVI’s tuning-free asymmetric quantization for the 2-bit and 4-bit quantization with group size and residual dimension set to 64.
- **CacheGen** [36]: CacheGen is the state-of-the-art cloud KV cache transmission method based on bitstream compression.
- **SVDq** [64]: We use SVDq to represent SVD-based methods. While SVDq primarily targets Key caches, we extend its low-rank approximation also to Value caches for a fair comparison.

6.2 Overall Performance Evaluation

TTFT reduction. Figure 16 illustrates the model accuracy and TTFT of CatKV and 5 compared solutions across all 4 benchmarks and 3 LLMs. CacheBlend is the default PIC algorithm. The result confirms that CatKV achieves a 2–5× TTFT reduction compared to the representative compression methods while preserving similar LLM generation quality with CacheBlend. The superior efficiency of CatKV mainly stems from two parts: (i) a higher compression ratio that significantly reduces

Table 5. EM scores across benchmarks. The best and second best results are marked as **bold** and underlined.

Method	CR	NeedleBench	2WikiMQA	MuSiQue	RepoBench
CacheBlend	1	69.5	18.5	17.3	<u>8.7</u>
SVDq	3	69.0	19.5	16.0	8.3
SQ4	4	<u>70.5</u>	18.5	<u>16.7</u>	9.3
KIVI	5.3	66.5	16.5	14.0	8.0
CacheGen	6	67.0	19.5	11.3	7.7
CatKV	8	71.0	<u>19.0</u>	16.0	<u>8.7</u>

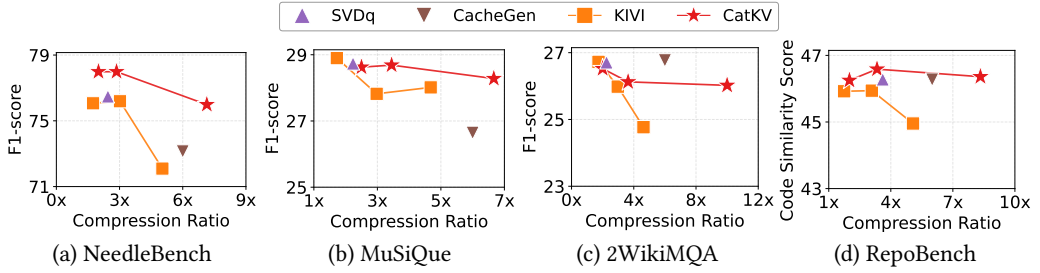


Fig. 17. CatKV reduces the KV cache footprint by 6–10× compared with the default PIC serving system, while incurring only a negligible accuracy drop.

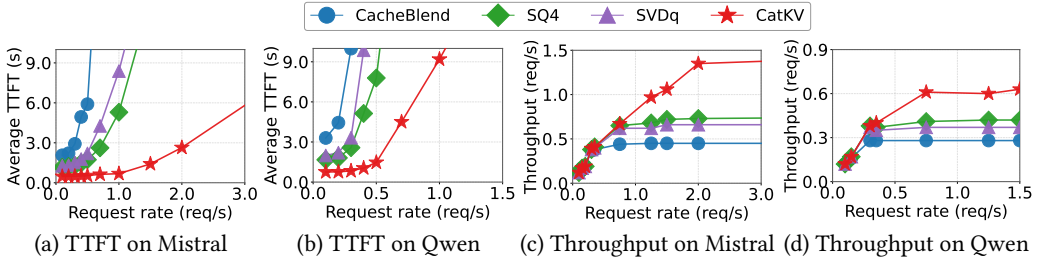


Fig. 18. Performance evaluation by varying request rates.

remote KV cache transfer overhead; and (ii) a six-stage fine-grained pipeline to overlap GPU model computation with other data transfers. SVDq and SQ4 generally achieve similar model accuracy with CatKV and CacheBlend but shows much longer TTFT due to insufficient compression ratio, causing significant I/O bottleneck. KIVI and CacheGen yield much better TTFT but harm model accuracy by over 5% in 6 out of 12 cases due to the aggressive use of low-bit approximations, i.e., 2-bit SQ, which leads to substantial estimation errors. Notably, despite its high compression ratio, CacheGen in some cases has longer TTFT than SQ4. This is because the complex decompression logic of CacheGen introduces significant computational overhead adding to GPU model computation on the critical path. Note that CatKV even yields better model accuracy than CacheBlend on 3 out of 12 cases. This is mainly due to the KV correction mechanism of PIC algorithms which may recompute compressed KV caches with high estimation errors.

EM evaluation. To rigorously evaluate generation quality, we compare the EM scores of different methods using the Mistral-7B. The results are reported in Table 5. It is clear that KIVI and CacheGen suffer from substantial quality degradation, especially on benchmarks NeedleBench and MuSiQue. In contrast, CatKV yields EM scores on par with CacheBlend, and achieves a much higher compression ratio than SVDq and SQ4. Thus, it suggests that CatKV effectively preserves the semantic integrity of the context via the proposed context-adaptive compression.

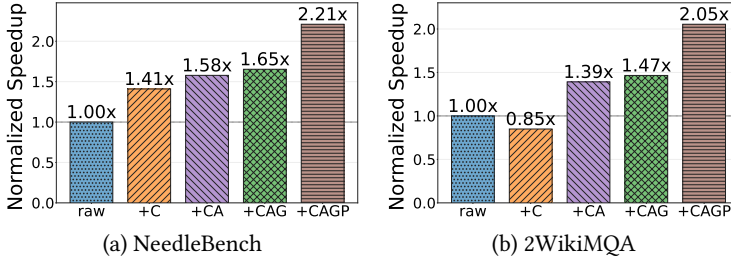


Fig. 19. Performance breakdown of different components in CatKV using Mistral-7B.

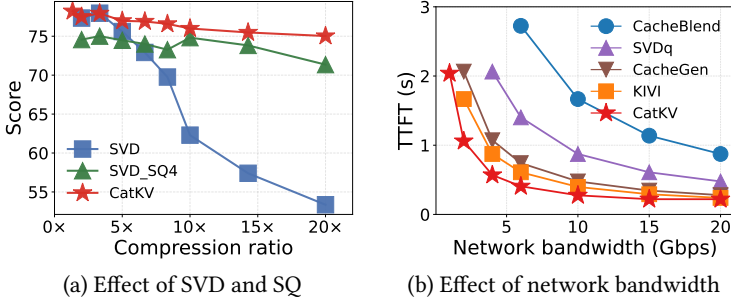


Fig. 20. Performance analysis using Mistral-7B.

KV cache compression ratio. Figure 17 shows the changing of accuracy on Mistral-7B with different compression ratios of CatKV and other competitors across 4 benchmarks. The 4-bit version of KIVI is the same as SQ4, we refer both as KIVI in Figure 17. SVDq only has a point in Figure 17 as it used fixed parameter, the same as CacheGen. The results show that CatKV consistently achieves the best accuracy among all competitors. Specifically, CatKV can achieve similar or better model accuracy than baseline solutions using just 30% to 50% of their KV cache sizes. The gap is more significant in the high compression ratios (e.g., 6-8 $\times$), where KIVI and CacheGen suffer severe accuracy degradation due to quantization errors, while CatKV maintains robust performance with a negligible accuracy drop (<3%). In particular, CatKV achieves near-lossless model accuracy even under aggressive compression configurations on RepoBench. This further verifies the effectiveness of our proposed mixed-precision approach combining SVD and SQ. Notably, increasing the KV cache size occasionally degrades accuracy, e.g., on MuSiQue and RepoBench. This counterintuitive result could be attributed to the attention dilution phenomenon [13, 32], where longer contexts tend to introduce distracting tokens. Consequently, with the growth of KV cache size, more noisy tokens are correctly represented and thus more likely to be attended. Similar experimental results are also observed in existing KV cache compression studies [37, 64].

Request rate evaluation. Figure 18 reports the average TTFT and throughput of CatKV under various request rates. We compare CatKV against CacheBlend, SVDq and SQ4 as they do not affect model generation quality. As shown in Figure 18, CatKV achieves the best TTFT and throughput among all cases, and the improvement is more pronounced with higher request rates. In particular, the throughput of baselines saturates at approximately 0.5 req/s, and this leads to exponential TTFT increases under higher loads. In comparison, CatKV maintains a consistently low TTFT even at 1.5 req/s and achieves a peak throughput up to 3 $\times$ higher than the baselines. This improvement stems from the reduced I/O latency and overlapped execution. With the mixed precision and asynchronous CPU compression, CatKV achieves the highest compression ratio and successfully

Table 6. Effect of grouped compression (w/ vs. w/o).

Datasets		KV Size (GB)	F1-score
2WikiMQA	CacheBlend	84.0	26.8
	w/o GC	25.9	26.2
	w/ GC	17.0	26.3
NeedleBench	CacheBlend	91.0	76.2
	w/o GC	15.5	76.1
	w/ GC	11.1	75.9

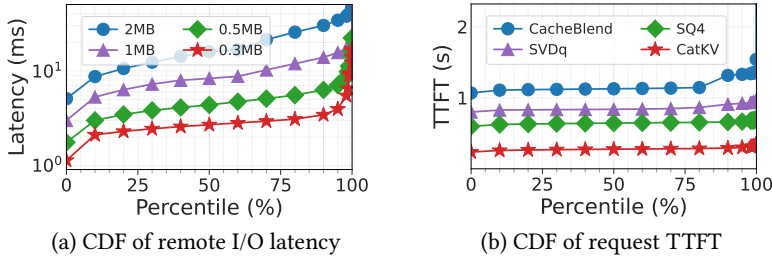


Fig. 21. Remote I/O latency and request TTFT analysis.

hides the I/O and CPU execution within the GPU model computation. Additionally, short TTFT for PIC workloads directly translates to high throughput as the prefill phase dominates the generation cost of long input contexts.

6.3 Effectiveness Study

Ablation study. Figure 19 depicts the contribution of each devised technique to the end-to-end speedup across benchmarks. Compared with CacheBlend (denoted as raw in Figure 19), introducing adaptive compression (+C) effectively reduces I/O latency, yielding a 1.41 $\times$ speedup on NeedleBench. Interestingly, adaptive compression degrades the performance to 0.85 $\times$ on 2WikiMQA. This disparity stems from the much lower chunk cache hit ratio on 2WikiMQA, and in this case the system performs heavy online compression sequentially with model computation on GPU. Consequently, this compression overhead is directly exposed on the critical path, outweighing the I/O benefits. To resolve this issue, we offload the compression asynchronously to the CPU (+CA). The results show that CPU-asynchronous compression effectively hides this overhead from the critical path and boosts the speedup to around 1.4 $\times$. Furthermore, grouped compression (+CAG) reduces both the remote storage size and I/O traffic by sharing one B matrix over multiple A matrices, yielding further speedup. Last but not least, layer-wise pipelining (+CAGP) maximizes hardware utilization by fully overlapping execution stages, propelling the end-to-end normalized speedup to over 2 $\times$ across two workloads.

Effect of SVD and SQ in CatKV. To validate the necessity of applying SQ over SVD, we conduct a decomposition of SVD and SQ on the NeedleBench using Mistral-7B. Specifically, we compare CatKV against vanilla truncated SVD and SVD combined with scalar quantization (SVD+SQ4). As shown in Figure 20(a), CatKV exhibits the least model accuracy degradation, offering the best accuracy-compression tradeoff. Even at an aggressive 20 $\times$ compression ratio, CatKV retains 95% relative accuracy. Notably, SVD+SQ4 under-performs pure SVD at low compression ratios because SVD already incurs little truncation error, while SQ adds dominant noise to the sensitive basis matrix B , outweighing the benefit of slightly higher ranks. In contrast, pure SVD collapses at

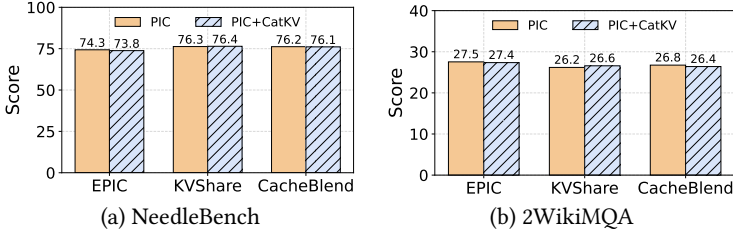


Fig. 22. Generality of CatKV with various PIC algorithms.

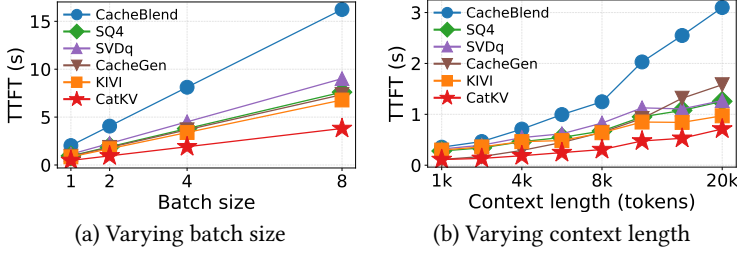


Fig. 23. TTFT of CatKV with different settings.

high compression ratios because it must aggressively reduce rank. CatKV avoids both failure modes by keeping error-sensitive low-rank components in high precision and quantizing residuals aggressively, achieving the best tradeoff across compression budgets.

Grouped compression. Table 6 validates the effectiveness of grouped compression mechanism in CatKV using the Mistral-7B model across 2 benchmarks. To ensure a fair comparison, we maintain the identical truncation rank for SVD in both *with grouped compression* and *without grouped compression*. Specifically, grouped compression applies a shared basis matrix B to cold chunks instead of storing them individually. The results show that, in the high-compression scenarios, grouped compression significantly reduces memory consumption by 20–30% while incurring negligible accuracy degradation.

Network bandwidth and I/O stability analysis. Figure 20(b) illustrates the sensitivity of TTFT to network bandwidth under a 16K context workload. CatKV’s performance saturates early at approximately 10 Gbps, indicating that the bottleneck shifts from I/O transmission to GPU computation. In contrast, other methods require substantially higher bandwidth (e.g., >20 Gbps) to reduce TTFT, whereas CatKV reaches near-optimal performance with a standard 10 Gbps connection. Figure 21 illustrates the request latency distributions of remote I/O and end-to-end execution. The results show that while the tail latency in remote I/O is long due to network fluctuations, the overall TTFT remains stable. This is reasonable as the prefill stage contains 32 layers, and each layer overlaps the remote I/O load with GPU compute operations. It is mostly unlikely that 32 layers all hit the tail latency. As such, isolated I/O delay in some layers has a minimal impact on the overall TTFT.

Other PIC algorithms. Figure 22 shows that CatKV is compatible with 3 representative PIC algorithms. To ensure a fair evaluation, we precomputed the reusable KV cache across all requests, thereby eliminating potential bias from the retrieval phase itself. The results on Mistral-7B indicate CatKV maintains accuracy comparable to the uncompressed baseline. These results suggest that CatKV is orthogonal to PIC algorithms, serving as a transparent compression layer that does not impair downstream correction efficacy.

Batch size and context length. Figure 23 presents that CatKV consistently minimizes TTFT across varying batch sizes and context lengths. Figure 23(a) shows that with a fixed 16K context,

CatKV exhibits the slowest latency growth as batch size increases, significantly outperforming baseline. Figure 23(b) suggests that CatKV maintains a substantial speedup as context length scales from 1K to 20K. The TTFT improvement of CatKV stems from its high compression ratio, which effectively mitigates the transmission bottlenecks inherent in existing solutions.

7 Related Work

Context compression. Prior work has reduced prefill cost by compressing prompts or caching knowledge-related intermediates. LLMingua [22, 41] compresses long prompts via token pruning under a budget, enabling faster inference. Other works like [25, 26, 67] targets RAG workloads by caching intermediate states of retrieved knowledge or graph structures to reduce TTFT.

KV cache management. Serving systems extensively optimize KV cache allocation and reuse. vLLM [27] introduces PagedAttention for efficient KV cache management and supports prefix caching to avoid redundant prefill computation. SGLang [70] further enables fast prefix reuse with RadixAttention. Other works [3, 14, 43] build hierarchical KV stores and cache-adaptive scheduling to reuse KV caches for multi-turn or agentic workloads. Beyond prefix reuse, PromptCache [17] exposes a modular schema to precompute and reuse attention states of recurring prompt segments. More recently, PIC reuse methods [2, 21, 63, 65] reuse KV caches of chunks beyond strict prefix order. CatKV addresses a specific PIC deployment challenge by compressing small, reusable chunk-level KV caches to reduce remote KV transfer overhead while preserving accuracy.

Sparse attention. Sparse attention aims to accelerate long-context inference by selectively attending to a subset of tokens, thereby reducing attention computation and KV cache size. Early works adopt static sparsity [29, 59, 60], where predefined patterns derived from empirical observations are fixed during decoding. Dynamic sparse attention methods [7, 10, 44, 52, 58] select important KV pairs at runtime based on attention approximation or lightweight retrieval mechanisms. Trainable sparse attention [16, 38, 68] further learns sparsity patterns through auxiliary modules, allowing the model to retain sparsity while closely approximating full attention behavior. While sparse attention targets attention computation, CatKV optimizes KV cache reuse at the system level. The two are orthogonal and can be used together.

8 Conclusion

In this work, we architect CatKV, a high-performance serving system for Position-Independent Caching (PIC) workloads. The core of CatKV is a context-adaptive compression scheme. By leveraging the elbow method to automatically determine precision configurations based on the singular-value spectrum, CatKV achieves aggressive KV cache compression while preserving original model accuracy. To further improve system efficiency, it introduces CPU asynchronous compression, grouped compression, and a six-stage layer-wise pipeline. The TTFT of CatKV is consistently faster than compared competitors by 2-5 $\times$, and CatKV reduces KV cache storage by up to 10 $\times$. The promising direction for future work is exploring fine-grained, multi-group quantization for emerging and complex workloads, which further improves the compression ratio.

9 Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (Grant Nos. 62422206, 62532007, 62532001), the High-level Special Funds (Grant No. G03034K009) in Southern University of Science and Technology and a research gift from AlayaDB.AI.

References

- [1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altmenschmidt, Sam Altman, et al. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).
- [2] Shubham Agarwal, Sai Sundaresan, Subrata Mitra, Debabrata Mahapatra, Archit Gupta, Rounak Sharma, Nirmal Joshua Kapu, Tong Yu, and Shiv Saini. 2025. Cache-Craft: Managing Chunk-Caches for Efficient Retrieval-Augmented Generation. In *SIGMOD*.
- [3] Zhuohang Bian, Feiyang Wu, Teng Ma, and Youwei Zhuo. 2025. TokenCake: A KV-Cache-centric Serving Framework for LLM-based Multi-Agent Applications. *arXiv preprint arXiv:2510.18586* (2025).
- [4] Brian J. Chan, Chao-Ting Chen, Jui-Hung Cheng, and Hen-Hsen Huang. 2025. Don't Do RAG: When Cache-Augmented Generation is All You Need for Knowledge Tasks. In *WWW*. 893–897.
- [5] Chi-Chih Chang, Chien-Yu Lin, Yash Akhauri, Wei-Cheng Lin, Kai-Chiang Wu, Luis Ceze, and Mohamed S Abdelfattah. 2025. xKV: Cross-layer svd for kv-cache compression. *arXiv preprint arXiv:2503.18893* (2025).
- [6] Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Chong-Yan Chen, Yu-Fang Hu, Pei-Shuo Wang, Ning-Chi Huang, Luis Ceze, Mohamed S. Abdelfattah, and Kai-Chiang Wu. 2025. Palu: KV-Cache Compression with Low-Rank Projection. In *ICLR*.
- [7] Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. 2024. ArkVale: Efficient generative llm inference with recallable key-value eviction. In *NeurIPS*.
- [8] Shiyang Chen, Rain Jiang, Dezhi Yu, Jinlai Xu, Mengyuan Chao, Fanlong Meng, Chenyu Jiang, Wei Xu, and Hang Liu. 2024. KVDirect: Distributed Disaggregated LLM Inference. *arXiv preprint arXiv:2501.14743* (2024).
- [9] Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. 2025. IMPRESS: An Importance-Informed Multi-Tier Prefix KV Storage System for Large Language Model Inference. In *FAST*. 187–201.
- [10] Yaoqi Chen, Jinkai Zhang, Baotong Lu, Qianxi Zhang, Chengruidong Zhang, Jing Liu, Jingjia Luo, Di Liu, Huiqiang Jiang, Qi Chen, Bailu Ding, Xiao Yan, Jiawei Jiang, Chen Chen, Mingxing Zhang, Cheng Li, Yuqing Yang, Fan Yang, and Mao Yang. 2025. RetroInfer: A Vector-Storage Approach for Scalable Long-Context LLM Inference. *Proc. VLDB Endow.* 19, 5 (2025).
- [11] Yangshen Deng, Zhengxin You, Long Xiang, Qilong Li, Peiqi Yuan, Zhaoyang Hong, Yitao Zheng, Wanting Li, Runzhong Li, Haotian Liu, Kyriakos Mouratidis, Man Lung Yiu, Huan Li, Qiaomu Shen, Rui Mao, and Bo Tang. 2025. AlayaDB: The Data Foundation for Efficient and Effective Long-context LLM Inference. In *SIGMOD*.
- [12] Dayou Du, Shijie Cao, Jianyi Cheng, Luo Mai, Ting Cao, and Mao Yang. 2026. BitDecoding: Unlocking Tensor Cores for Long-Context LLMs with Low-Bit KV Cache. In *HPCA*.
- [13] Yufeng Du, Mingyang Tian, Srikanth Ronanki, Subendhu Rongali, Sravan Bodapati, Aram Galstyan, Azton Wells, Roy Schwartz, Eliu A Huerta, and Hao Peng. 2025. Context Length Alone Hurts LLM Performance Despite Perfect Retrieval. In *EMNLP*.
- [14] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient large language model serving for multi-turn conversations with CachedAttention. In *ATC*. 111–126.
- [15] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2, 1 (2023).
- [16] Yizhao Gao, Zhichen Zeng, Dayou Du, Shijie Cao, Peiyuan Zhou, Jiaying Qi, Junjie Lai, Hayden Kwok-Hay So, Ting Cao, Fan Yang, and Mao Yang. 2024. SeerAttention: Learning intrinsic sparse attention in your llms. *arXiv preprint arXiv:2410.13276* (2024).
- [17] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular attention reuse for low-latency inference. In *MLSYS*.
- [18] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).
- [19] Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. 2020. Constructing A Multi-hop QA Dataset for Comprehensive Evaluation of Reasoning Steps. In *COLING*. 6609–6625.
- [20] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. 2024. KVQuant: Towards 10 million context length llm inference with kv cache quantization. In *NeurIPS*.
- [21] Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, Tiancheng Hu, Zhang Qin, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. 2025. EPIC: Efficient Position-Independent Caching for Serving Large Language Models. In *ICML*.
- [22] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LLMingua: Compressing Prompts for Accelerated Inference of Large Language Models. In *EMNLP*.
- [23] Wenqi Jiang, Suvinay Subramanian, Cat Graves, Gustavo Alonso, Amir Yazdanbakhsh, and Vidushi Dadu. 2025. RAGO: Systematic Performance Optimization for Retrieval-Augmented Generation Serving. In *ISCA*.

- [24] Wenqi Jiang, Marco Zeller, Roger Waleffe, Torsten Hoefer, and Gustavo Alonso. 2024. Chameleon: a Heterogeneous and Disaggregated Accelerator System for Retrieval-Augmented Language Models. *Proc. VLDB Endow.* 18, 1 (2024).
- [25] Yinsicheng Jiang, Yeqi Huang, Liang Cheng, Cheng Deng, Xuan Sun, and Luo Mai. 2026. ContextPilot: Fast Long-Context Inference via Context Reuse. In *MLSYS*.
- [26] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. 2025. RAGCache: Efficient knowledge caching for retrieval-augmented generation. *TOCS* 44, 1 (2025), 1–27.
- [27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*. 611–626.
- [28] Mo Li, Songyang Zhang, Taolin Zhang, Haodong Duan, Yunxin Liu, and Kai Chen. 2024. NeedleBench: Can LLMs Do Retrieval and Reasoning in Information-Dense Context? *arXiv preprint arXiv:2407.11963* (2024).
- [29] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. SnapKV: Llm knows what you are looking for before generation. In *NeurIPS*.
- [30] Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, Rui Kong, Yile Wang, Hanfei Geng, Jian Luan, Xuefeng Jin, Zilong Ye, Guanqing Xiong, Fan Zhang, Xiang Li, Mengwei Xu, Zhijun Li, Peng Li, Yang Liu, Ya-Qin Zhang, and Yunxin Liu. 2024. Personal llm agents: Insights and survey about the capability, efficiency and security. *arXiv preprint arXiv:2401.05459* (2024).
- [31] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).
- [32] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranajape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. In *TACL*.
- [33] Renjie Liu, Haitian Jiang, Xiao Yan, Bo Tang, and Jinyang Li. 2025. PolyG: Effective and Efficient GraphRAG with Adaptive Graph Traversal. *arXiv preprint arXiv:2504.02112* (2025).
- [34] Tianyang Liu, Canwen Xu, and Julian J. McAuley. 2024. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. In *ICLR*.
- [35] Yuhan Liu, Yuyang Huang, Jiayi Yao, Shaoting Feng, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, Madan Musuvathi, and Esha Choukse. 2026. DroidSpeak: KV Cache Sharing Across Fine-tuned Model Variants. In *NSDI*. 319–338.
- [36] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *SIGCOMM*. 38–56.
- [37] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen (Henry) Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache. In *ICML*.
- [38] Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, Zhiqi Huang, Huan Yuan, Suting Xu, Xinran Xu, Guokun Lai, Yanru Chen, Huabin Zheng, Junjie Yan, Jianlin Su, Yuxin Wu, Yutao Zhang, Zhilin Yang, Xinyu Zhou, Mingxing Zhang, and Jiezhong Qiu. 2025. MoBA: Mixture of Block Attention for Long-Context LLMs. In *NeurIPS*.
- [39] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. 2016. MS MARCO: A Human Generated MACHine Reading COMprehension Dataset. In *Proceedings of the NeurIPS Workshop on Cognitive Computation*.
- [40] OpenAI. 2024. ChatGPT. <https://chatgpt.com>. Accessed: 2026-01-18.
- [41] Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. 2024. LLMingua-2: Data Distillation for Efficient and Faithful Task-Agnostic Prompt Compression. In *Findings of ACL*. 963–981.
- [42] Boci Peng, Yun Zhu, Yongchao Liu, Xiaohe Bo, Haizhou Shi, Chuntao Hong, Yan Zhang, and Siliang Tang. 2025. Graph Retrieval-Augmented Generation: A Survey. In *TOIS*.
- [43] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *FAST*. 155–170.
- [44] Luka Ribar, Ivan Chelombiev, Luke Hudliss-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. 2024. SparQ attention: bandwidth-efficient LLM inference. In *ICML*.
- [45] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. 2011. Finding a "needle" in a haystack: Detecting knee points in system behavior. In *Proc. ICDCS Workshops*. 166–171.
- [46] Amazon Web Services. 2024. Amazon EBS - High-Performance Block Storage. <https://aws.amazon.com/ebs/>. Accessed: 2026-01-18.
- [47] Amazon Web Services. 2024. Amazon S3 - Cloud Object Storage. <https://aws.amazon.com/s3/>. Accessed: 2026-01-18.

- [48] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *ICML*.
- [49] Kun-Woo Shin, Jay H Park, Moonwook Oh, Yohan Jo, Jaeyoung Do, and Sang-Won Lee. 2025. MatKV: Trading Compute for Flash Storage in LLM Inference. *arXiv preprint arXiv:2512.22195* (2025).
- [50] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. RoFormer: Enhanced transformer with Rotary Position Embedding. *Neurocomput.* 568 (2024).
- [51] Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. 2025. ShadowKV: KV Cache in Shadows for High-Throughput Long-Context LLM Inference. In *ICML*.
- [52] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. QUEST: query-aware sparsity for efficient long-context LLM inference. In *ICML*.
- [53] The Mistral AI Team. 2023. Mistral 7B: Efficient and Competitive Open Language Model. *arXiv preprint arXiv:2310.06825* (2023).
- [54] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. MuSiQue: Multihop Questions via Single-hop Question Composition. In *TACL*. 539–554.
- [55] Dandan Wang and Shiqing Zhang. 2024. Large language models in medical and healthcare fields: applications, advances, and challenges. *Artif. Intell. Rev.* 57, 11 (2024), 299.
- [56] Shen Wang, Tianlong Xu, Hang Li, Chaoli Zhang, Joleen Liang, Jiliang Tang, Philip S Yu, and Qingsong Wen. 2024. Large language models for education: A survey and outlook. *arXiv preprint arXiv:2403.18105* (2024).
- [57] Xin Wang, Samiul Alam, Zhongwei Wan, Hui Shen, and Mi Zhang. 2025. SVD-LLM V2: Optimizing Singular Value Truncation for Large Language Model Compression. In *NAACL*. 4287–4296.
- [58] Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. 2024. Inflm: Training-free long-context extrapolation for llms with an efficient context memory. In *NeurIPS*.
- [59] Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. 2025. DuoAttention: Efficient Long-Context LLM Inference with Retrieval and Streaming Heads. In *ICLR*.
- [60] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient Streaming Language Models with Attention Sinks. In *ICLR*.
- [61] Ceyu Xu, Yongji Wu, Xinyu Yang, Beidi Chen, Matthew Lentz, Danyang Zhuo, and Lisa Wu Wills. 2025. LLM.265: Video Codecs are Secretly Tensor Codecs. In *MICRO*. 445–460.
- [62] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, et al. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671* 2, 3 (2024).
- [63] Huan Yang, Renji Zhang, Mingzhe Huang, Weijun Wang, Yin Tang, Yuanchun Li, Yunxin Liu, and Deyu Zhang. 2025. KVShare: An LLM Service System with Efficient and Effective Multi-Tenant KV Cache Reuse. *arXiv preprint arXiv:2503.16525* (2025).
- [64] Hong Yankun, Li Xing, Zhen Hui-Ling, Yu Xianzhi, Liu Wulong, and Yuan Mingxuan. 2025. SVDq: 1.25-bit and 410x Key Cache Compression for LLM Attention. *arXiv preprint arXiv:2502.15304* (2025).
- [65] Jiayi Yao, Hanchen Li, Yuhao Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *EuroSys*. 94–109.
- [66] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Gunhee Soo, and Byung-Gon Chun. 2022. Orca: A distributed serving system for Transformer-based generative models. In *OSDI*. 521–538.
- [67] Hao Yuan, Xin Ai, Qiang Wang, Peizheng Li, Jiayang Yu, Chaoyi Chen, Xinbo Yang, Yanfeng Zhang, Zhenbo Fu, Yingyou Wen, and Ge Yu. 2025. DepCache: A KV Cache Management Framework for GraphRAG with Dependency Attention. In *SIGMOD*.
- [68] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. 2025. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *ACL*. 23078–23097.
- [69] Rongzhi Zhang, Kuang Wang, Liyuan Liu, Shuohang Wang, Hao Cheng, and Yelong Shen. 2024. LORC: Low-Rank Compression for LLMs KV Cache with a Progressive Compression Strategy. In *NeurIPS*.
- [70] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: efficient execution of structured language model programs. In *NeurIPS*.
- [71] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *OSDI*. 193–210.

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026