ChunkKV: Semantic-Preserving KV Cache Compression for Efficient Long-Context LLM Inference

Xiang LIU[♡*] **Zhenheng TANG**[♣*] **Peijie DONG**[♡] **Zeyu LI**[♡]
Yue LIU[♢†] **Bo LI**[♠♣] **Xuming HU**[♡†] **Xiaowen CHU**[♡†]
[♡] The Hong Kong University of Science and Technology (Guangzhou)
[♣] CSE, The Hong Kong University of Science and Technology
[♠]Guangzhou HKUST Fok Ying Tung Research Institute
[♢] Terminus Technologies
{xliu886,pdong212,zli755}@connect.hkust-gz.edu.cn
{zhtang.ml, bli}@cse.ust.hk {xuminghu, xwchu}@hkust-gz.edu.cn

Abstract

Large Language Models (LLMs) require significant GPU memory when processing long texts, with the key value (KV) cache consuming up to 70% of total memory during inference. Although existing compression methods reduce memory by evaluating the importance of individual tokens, they overlook critical semantic relationships between tokens, resulting in fragmented context and degraded performance. We introduce ChunkKV, which fundamentally reimagines KV cache compression by treating semantic chunks - rather than isolated tokens - as basic compression units. This approach preserves complete linguistic structures and contextual integrity, ensuring that essential meaning is retained even under aggressive compression. Our innovation includes a novel layer-wise index reuse technique that exploits the higher cross-layer similarity of preserved indices in ChunkKV, reducing computational overhead and improving throughput by 26.5%. Comprehensive evaluations on challenging benchmarks: LongBench, Needle-In-A-HayStack, GSM8K, and JailbreakV demonstrate that ChunkKV outperforms state-of-the-art methods by up to 8.7% in precision while maintaining the same compression ratio. These results confirm that semantic-aware compression significantly enhances both efficiency and performance for long-context LLM inference, providing a simple yet effective solution to the memory bottleneck problem. *The code is available at link.*

1 Introduction

Large Language Models (LLMs) have become essential for addressing various downstream tasks of natural language processing (NLP), including summarization and question answering, which require the interpretation of a long context from sources such as books, reports, and documents, often encompassing tens of thousands of tokens [1–5]. Recent advances in long-context technology within the field of machine learning (ML) systems [6–8] have significantly improved computational throughputs and reduced latency of LLMs to process increasingly large input context lengths [9, 10] with historical KV cache (key value attentions). However, the memory requirement of the KV cache in serving super-long contexts becomes a new bottleneck [11–14]. For instance, the KV cache for a

[*]Equal Contribution.
[†]Corresponding Author.

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

single token in a 7B-parameter model requires approximately 0.5 MB of GPU memory, resulting in a 10,000-token prompt consuming around 5 GB of GPU memory.

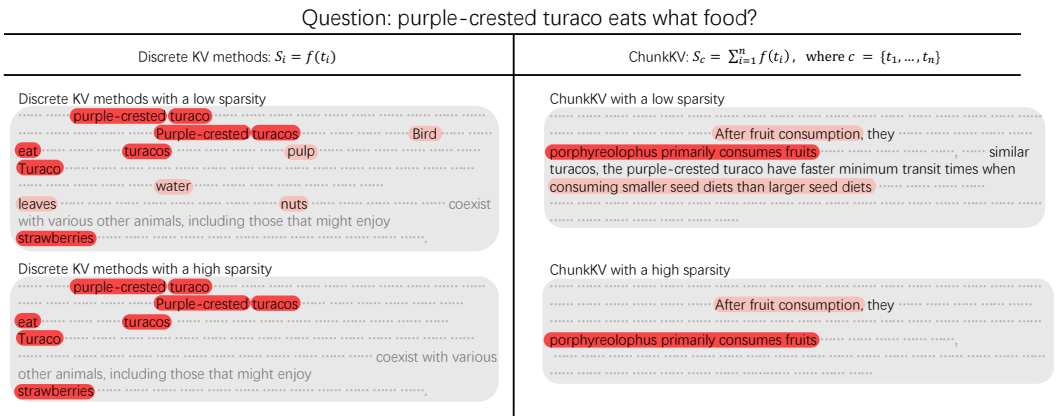


Figure 1: Illustration of the impact of the token discrete method and the chunk method on semantic preservation. The discrete method preserves words related to the question but often omits the subject. In contrast, the chunk method retains the subject of the words, maintaining more accurate semantic information. For the equation: S is the score function, and c is a chunk of tokens.

To address the substantial GPU memory consumption caused by KV caching, recent studies consider compressing the KV cache by pruning non-important discrete parts from the prompt tokens [11, 12, 15–21]. H2O [11] and SnapKV [15] have shown that retaining less than 50% of the discrete KV cache can significantly reduce GPU memory usage with minimal impact on performance. However, we identify that the previous KV cache compression methods [11, 17] measure token importance isolatedly, neglecting the dependency between different tokens on the characteristics of the real-world language. For example, as shown in Figure 1, focusing on the importance of the token level can excessively focus on words about subjects "turaco" in the question while omitting crucial information about objects (foods) in the documents, resulting in the loss of essential semantic information. This motivates us to rethink the following question:

How to avoid isolated token importance measurement and preserve the semantic information in KV cache?

Table 1: Comparison of Methods on KV Cache Compression.

Method	KV Cache Compression	Dynamic Policy	Layer-Wise Policy	Semantic Information	Efficient Index Reuse
StreamingLLM [8]	✓				
H2O [11]	✓	✓			
SnapKV [15]	✓	✓			
PyramidInfer [19]	✓	✓	✓		
PyramidKV [17]	✓	✓	✓		
ChunkKV(Ours)	✓	✓	✓	✓	✓

In light of this, we observe that complete semantic information usually appears in a continuous sequence [4, 22–24]. Thus, we introduce a straightforward yet effective ChunkKV, grouping the tokens in a chunk as a basic compressing unit, which should be preserved or discarded as a whole. Thus, it retains the most informative **semantic chunks** from the original KV cache. As shown in Figure 1, preserving a chunk helps catch the subject, predicate, and object. In Table 13, we quantify the minimal loss and higher recovery of the attention score achieved by the chunk-based approach at the inference stage. Furthermore, we investigate that *the preserved KV cache indices by ChunkKV exhibit a higher similarity* compared to previous methods. Consequently, we develop a technique called layer-wise index reuse, which reduces the additional computational time introduced by the KV cache compression method. As outlined in Table 1, recent highly relevant KV cache compression methods *lack the ability to retain semantic information and efficiently reuse indices*.

To evaluate ChunkKV's performance, we conduct comprehensive experiments across multiple cutting-edge long-context benchmarks: long-context tasks including LongBench [25] and Needle-

In-A-HayStack (NIAH) [26], in-context learning tasks such as GSM8K [27] and JailbreakV [28]. And also different models including DeepSeek-R1-Distill-Llama-8B [29],LLaMA-3-8B-Instruct [30], Mistral-7B-Instruct [31], and Qwen2-7B-Instruct [32]. Our experimental results demonstrate that ChunkKV surpasses existing KV cache compression methods in both efficiency and accuracy, primarily due to its ability to preserve essential information through selective chunk retention. These findings establish ChunkKV as a simple yet effective approach to KV cache compression.

We summarize our key contributions as follows:

- We identify the phenomenon in which discrete KV cache compression methods inadvertently prune the necessary semantic information.
- We propose ChunkKV, a simple KV cache compression method that uses the fragmentation method that retains semantic information, and design the layer-wise index reuse technique to reduce the additional computational time.
- We evaluate ChunkKV on cutting-edge long-context benchmarks including LongBench and Needle-In-A-HayStack, as well as the GSM8K, many-shot GSM8K and JailbreakV in-context learning benchmark, and multi-step reasoning (O1 and R1) LLMs, achieving state-of-the-art performance.

2 Related Work

KV Cache Compression. KV cache compression technology has developed rapidly in the era of LLM, with methods mainly focused on evicting unimportant tokens. The compression process occurs before the attention blocks, optimizing both the prefilling time and GPU memory. Xiao et al. [8] and Han et al. [33] propose that initial and recent tokens consistently have high attention scores between different layers and attention heads. As a result, retaining these tokens in the KV cache is more likely to preserve important information. Furthermore, FastGen [16] evicts tokens based on observed patterns. H2O [11] and SnapKV [15] employ dynamic KV cache compression methods, evaluating the importance of tokens based on attention scores and then evicting the less important ones. As inference scenarios become increasingly complex, dynamic KV cache compression methods demonstrate powerful performance. Recently, Yang et al. [19] and Cai et al. [17] have closely examined the distributions of attention scores during the pre-filling stage of the Retrieval-Augmented Generation (RAG) task, discovering a pyramidal KV cache compression pattern in different transformer layers. In contrast, FlowKV [34] employs a novel multi-turn isolation mechanism that preserves the accumulated compressed KV cache from past turns and only compresses the KV pairs from the most recent turn, effectively mitigating the re-compression of older context and the problem of catastrophic forgetting.

Although these KV cache compression methods have explored efficient GPU memory management while maintaining original performance, our study focuses more on the semantic information of the prompt. We find that chunks of the original KV cache are more important than discrete tokens.

3 ChunkKV

3.1 Preliminary Study of KV Cache Compression

With the increasing long-context capabilities of LLMs, the KV cache has become crucial for improving the inference efficiency. However, it can consume significant GPU memory when handling long input contexts. The GPU memory cost of the KV cache for the decoding stage can be calculated as follows:

$$M_{KV} = 2 \times B \times S \times L \times N \times D \times 2 \tag{1}$$

where B is the batch size, S is the sequence length of prompt and decoded length, L is the number of layers, N is the number of attention heads, D is the dimension of each attention head, and the first 2 accounts for the KV matrices, while the last 2 accounts for the precision when using float16. Table E shows the configuration parameters for LLaMA-3-8B-Instruct [30]. With a batch size $B = 1$ and a sequence length of prompt $S = 2048$, the GPU memory cost of the KV cache is nearly 1 GB. If the batch size exceeds 24, the GPU memory cost of the KV cache will exceed the capacity of an RTX 4090 GPU. To address this issue, KV cache compression methods have been proposed, with the aim of retaining only a minimal amount of KV cache while preserving as much information as possible. For more details on the LLM configuration parameters, refer to Appendix E.

3.2 Chunk Based KV Compression

To address the limitations of existing KV cache compression methods, we propose ChunkKV, a novel KV cache compression method that retains the most informative semantic chunks. The key idea behind ChunkKV is to group tokens in the KV cache into chunks that preserve more semantic information, such as a chunk containing a subject, verb, and object. As illustrated in Figure 1, ChunkKV preserves the chunks of the KV cache that contain more semantic information. First, we define a chunk as a group of tokens that contain related semantic information. By retaining the most informative chunks from the original KV cache, ChunkKV can effectively reduce the memory usage of the KV cache while preserving essential information. For more information on ChunkKV, please refer to Section A.

Algorithm 1 ChunkKV

Input: $\mathbf{Q} \in \mathbb{R}^{T_q \times d}$, $\mathbf{K} \in \mathbb{R}^{T_k \times d}$, $\mathbf{V} \in \mathbb{R}^{T_v \times d}$, observe window size w, chunk size c, compressed KV cache max length L_{max}
Output: Compressed KV cache $\mathbf{K}'$, $\mathbf{V}'$
Observe Window Calculation:
$\mathbf{A} \leftarrow \mathbf{Q}_{T_q-w:T_q}\mathbf{K}^T$ ▷ Attention scores for the observe window
$C \leftarrow \left\lceil \frac{T_k}{c} \right\rceil$ ▷ Calculate the number of chunks
Chunk Attention Score Calculation:
for $i = 1$ to C **do**
  $\mathbf{A}_i \leftarrow \sum_{j=(i-1)c+1}^{ic} \mathbf{A}_{:,j}$ ▷ Sum of attention scores for each chunk
end for
Top-K Chunk Selection:
$k \leftarrow \left\lfloor \frac{L_{\text{max}}}{c} \right\rfloor$
Top_K_Indices $\leftarrow$ indices of Top-k chunks based on $\mathbf{A}_i$
Compression:
$\mathbf{K}', \mathbf{V}' \leftarrow$ index_select$(\mathbf{K}, \mathbf{V}, \textit{Top_K_Indices})$
Concatenation:
$\mathbf{K}' \leftarrow \text{concat}(\mathbf{K}'_{0:L_{\text{max}}-w}, \mathbf{K}_{T_k-w:T_k})$
$\mathbf{V}' \leftarrow \text{concat}(\mathbf{V}'_{0:L_{\text{max}}-w}, \mathbf{V}_{T_v-w:T_v})$
$\mathbf{K}', \mathbf{V}'$

The algorithm 1 shows the pseudocode for ChunkKV. First, following the approach of H2O [11] and SnapKV [15], we set the observe window by computing the attention scores $\mathbf{A} \leftarrow \mathbf{Q}_{T_q-w:T_q}\mathbf{K}^T$, where $\mathbf{Q}_{T_q-w:T_q}$ is the observe window, $\mathbf{K}$ is the Key matrix and the window size w is usually set to $\{4, 8, 16, 32\}$. Next, the number of chunks C is calculated as $C = \left\lceil \frac{T_k}{c} \right\rceil$, where T_k is the length of the Key matrix and c is the chunk size. The attention scores for each chunk are then computed as $\mathbf{A}_i = \sum_{j=(i-1)c+1}^{ic} \mathbf{A}_{:,j}$ for $i = 1, 2, \ldots, C$. We use the top-k algorithm as the sampling policy for ChunkKV. The top-k chunks are selected based on their attention scores, where $k = \left\lfloor \frac{L_{\text{max}}}{c} \right\rfloor$, and L_{max} is the maximum length of the compressed KV cache. The size of the last chunk is set to $\min(c, L_{\text{max}} - (k-1) \times c)$. The indices of the top-k chunks preserve the original sequence order. In the compression step, only the key and value matrices corresponding to the selected indices are retained, resulting in the compressed KV cache. Finally, the observe window of the original KV cache will be concatenated to the compressed KV cache by replacing the last w tokens to keep important information. The compressed KV cache is then used for subsequent attention computations. For implementation, we add vectorized operations, memory optimizations, etc., to optimize the code. Refer to Appendix A.5 for more details.

3.3 Layer-Wise Index Reuse

Furthermore, we investigated the KV cache indices preserved by ChunkKV and found that they exhibit higher similarity compared to previous methods. Figure 2 shows the layer-wise similarity heatmaps of SnapKV and ChunkKV. Each cell represents the similarity between the

Table 2: Retained KV Cache Indices Similarity of Adjacent Layers for Different Models.

Method	H2O	SnapKV	ChunkKV
LLaMA-3-8B	25.31%	27.95%	**57.74%**
Qwen2-7B	14.91%	16.50%	**44.26%**
Mistral-7B	15.15%	15.78%	**52.16%**

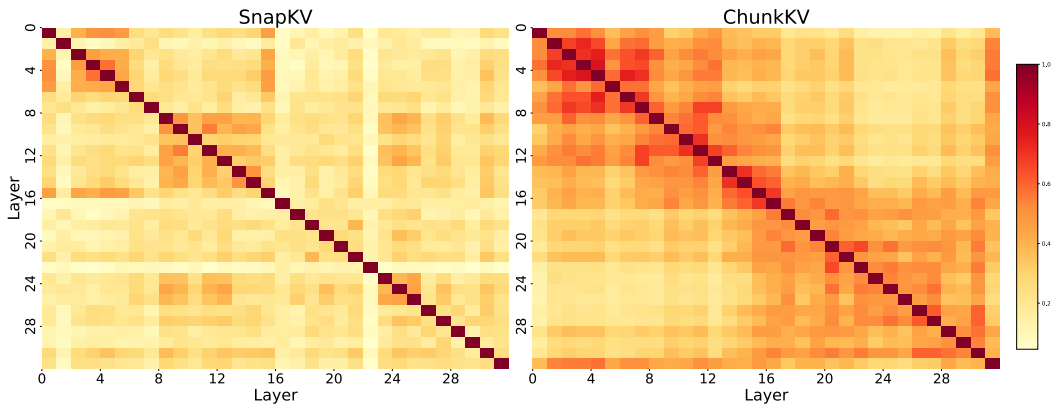


Figure 2: Layer-wise similarity heatmaps of the preserved KV **cache indices** by SnapKV (left) and ChunkKV (right) on LLaMA-3-8B-Instruct. Deep colors indicate higher similarity. More visualization can be found in Appendix B.1.3.

preserved KV cache indices of two layers, with deeper colors indicating higher similarity. The results demonstrate that the KV cache indices preserved by ChunkKV are more similar to those in neighboring layers.

As shown in Table 2, ChunkKV consistently achieves a higher average Jaccard similarity between adjacent layers compared to SnapKV in different model architectures, indicating that the retained token index in ChunkKV is more similar to each other. For a more detailed visualization, please refer to Appendix B.1.3.

Based on the above findings of the KV cache index, we propose a training-free *layer-wise index reuse* method to further reduce the additional cost of KV cache compression, which reuses compressed token indices across multiple layers. We evaluate the efficiency and effectiveness of the layer-wise index reuse method in Section 4.3. Reuse of the layer-wise index reduces the KV cache compression time by 20% compared to the FullKV baseline, with a performance drop of only 0.5%.

Algorithm 2 Layer-wise Index Reuse for ChunkKV

Input: Number of layers in LLMs N_{layers}, number of reuse layers N_{reuse}
Initialize: Dictionary to store indices $\mathcal{I}_{\text{reuse}} = \{\}$
for $l = 0$ to $(N_{\text{layers}} - 1)$ **do**
  if $l \mod N_{\text{reuse}} == 0$ **then**
    $\mathbf{K}'_l, \mathbf{V}'_l, \mathcal{I}_l \leftarrow \text{ChunkKV}(\mathbf{K}_l, \mathbf{V}_l)$
    $\mathcal{I}_{\text{reuse}}[l] \leftarrow \mathcal{I}_l$
  else
    $\mathcal{I}_l \leftarrow \mathcal{I}_{\text{reuse}}[\lfloor \frac{l}{N_{\text{reuse}}} \rfloor \times N_{\text{reuse}}]$
  end if
  $\mathbf{K}'_l \leftarrow \text{index\_select}(\mathbf{K}_l, \mathcal{I}_l)$
  $\mathbf{V}'_l \leftarrow \text{index\_select}(\mathbf{V}_l, \mathcal{I}_l)$
end for

This *layer-wise index reuse* method is formally described in Algorithm 2. The ChunkKV compression process returns the compressed KV cache and their respective token indices, denoted as $\mathcal{I}_l$. For layer-wise index reuse, we define a grouping of layers such that all N_{reuse} layers share the same token indices for ChunkKV. Specifically, for a group of layers $\{l, l+1, \ldots, l+N_{\text{reuse}}-1\}$, we perform ChunkKV on the first layer l to obtain the token indices $\mathcal{I}_l$ and reuse $\mathcal{I}_l$ for the subsequent layers $l+1, l+2, \ldots, l+N_{\text{reuse}}-1$. The notation $\mathbf{K}_l[\mathcal{I}_l]$ and $\mathbf{V}_l[\mathcal{I}_l]$ indicates the selection of key and value caches based on the indices in $\mathcal{I}_l$. The efficiency analysis for layer-wise index reuse is provided in Appendix B.1.1.

Theoretical Understanding. We provide a theoretical understanding from the in-context

Table 3: GSM8K Performance Comparison. # SLM=StreamingLLM, SKV=SnapKV, PKV=PyramidKV

Ratio	SLM	H2O	SKV	PKV	**ChunkKV (Ours)**
DeepSeek-R1-Distill-Llama-8B FullKV: 69.4% ↑					
10%	51.6%	55.6%	57.6%	62.6%	**65.7%**
LlaMa-3.1-8B-Instruct FullKV: 79.5% ↑					
30%	70.5%	72.2%	76.1%	77.1%	**77.3%**
20%	63.8%	64.0%	68.8%	71.4%	**77.6%**
10%	47.8%	45.0%	50.3%	48.2%	**65.7%**
LlaMa-3-8B-Instruct FullKV: 76.8% ↑					
30%	70.6%	73.6%	70.2%	68.2%	**74.6%**
Qwen2-7B-Instruct FullKV: 71.1% ↑					
30%	70.8%	61.2%	70.8%	64.7%	**73.5%**

learning (ICL) [24] to interpret why maintaining the KV cache according to a continuous sequence in ChunkKV is better than according to sparse tokens. Informally speaking, the continuously chunk-level KV cache preserves the whole examples (semantic information) in ICL, thus reducing the requirement on distinguishability, i.e., lower bound of KL divergence between the example and the question (Equation 4 in condition 2). The complete analysis is provided in Appendix C.

4 Experiment Results

In this section, we conduct experiments to evaluate the effectiveness of ChunkKV on KV cache compression in two benchmark fields, with a chunk size set to 10 even for various model architectures. The first is the In-Context Learning benchmark, for which we select GSM8K [27] and Jailbreakv [28, 35, 36] to evaluate the performance of ChunkKV, furthermore, we also include multi-step reasoning LLM DeepSeek-R1-Distill-Llama-8B [29] to evaluate the performance of ChunkKV. The In-Context Learning scenario is a crucial capability for LLMs and has been adapted in many powerful technologies such as Chain-of-Thought [37–40]. The second is the Long-Context benchmark, which includes LongBench [25] and Needle-In-A-HayStack (NIAH) [26], both widely used for assessing KV cache compression methods. All experiments were carried out three times, using the mean score to ensure robustness.

4.1 In-Context Learning

The In-Context Learning (ICL) ability significantly enhances the impact of prompts on LLMs. For example, the Chain-of-Thought approach [37] increases the accuracy of the GSM8K of the PaLM model [41] from 18% to 57% without additional training. In this section, we evaluate the performance of ChunkKV on the GSM8K, Many-Shot GSM8K [42], and JailbreakV [28] benchmarks.

GSM8K. In the in-context learning scenario, we evaluated multiple KV cache compression methods for GSM8K [27], which contains more than 1,000 arithmetic questions on LLaMA-3-8B-Instruct, LLaMA-3.1-8B-Instruct [30], Qwen2-7B-Instruct [32] and DeepSeek-R1-Distill-Llama-8B [29]. Following Agarwal et al. [42], we consider many-shot GSM8K as a long-context reasoning scenario, which is a more challenging task than long-context retrieval benchmark LongBench [25]. The CoT prompt settings for this experiment are the same as those used by Wei et al. [37], for many-shot GSM8K we set the number of shots to 50, where the prompt length is more than 4k tokens. For more details on the prompt settings, please refer to the APPENDIX G.

Table 4: Many-Shot (50-shot) GSM8K Performance Comparison.

Ratio	SLM	H2O	SKV	PKV	**ChunkKV (Ours)**
DeepSeek-R1-Distill-Llama-8B FullKV: 71.2% ↑					
10%	63.2%	54.2%	54.1%	59.2%	**68.2%**
LlaMa-3.1-8B-Instruct FullKV: 82.4% ↑					
10%	74.3%	51.2%	68.2%	70.3%	**79.3%**

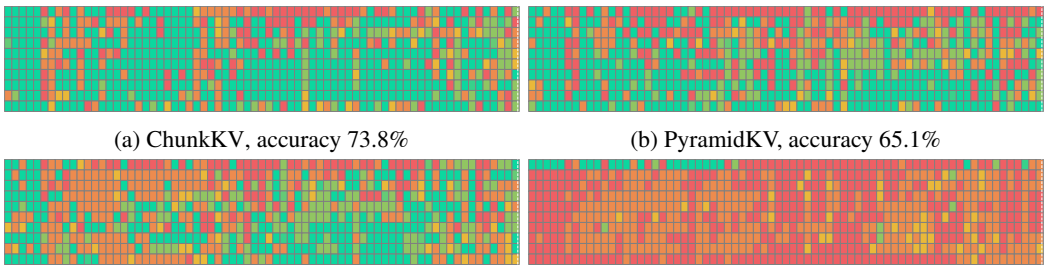
Table 3 presents the performance comparison. The results show that ChunkKV outperforms other KV cache compression methods on different models and compression ratios. Table 4 presents the performance comparison of many-shot GSM8K, and also ChunkKV outperforms other KV cache compression methods. The consistent superior performance of ChunkKV in both models underscores its effectiveness in maintaining crucial contextual information for complex arithmetic reasoning tasks by chunk level KV cache rather than the discrete token level.

Jailbreak. In this section, we evaluate the performance of ChunkKV on the JailbreakV benchmark [28], which is a safety Jailbreak benchmark for language models. The prompt settings are the same as those used by Luo et al. [28].

Table 5: JailbreakV Performance Comparison.

Ratio	SLM	H2O	SKV	PKV	**ChunkKV (Ours)**
LlaMa-3.1-8B-Instruct FullKV: 88.9% ↑					
20%	65.0%	71.7%	88.0%	87.5%	**89.0%**
10%	53.1%	65.4%	84.3%	85.5%	**87.9%**

Table 5 presents the performance comparison. The results demonstrate that ChunkKV outperforms other KV cache compression methods on different compression ratios. This shows that for safety benchmark, the chunk level KV cache is more effective than other discrete token level compression methods.



(a) ChunkKV, accuracy 73.8%

(b) PyramidKV, accuracy 65.1%

(c) SnapKV, accuracy 58.9%

(d) StreamingLLM, accuracy 23.7%

Figure 3: NIAH benchmark for LLaMA3-8B-Instruct with KV cache size=128 under 8k context length.

4.2 Long-Context Benchmark

LongBench and NIAH are two widely used benchmarks for KV cache compression methods. Both benchmarks have a context length that exceeds $10K$. NIAH requires retrieval capability, while LongBench is a meticulously designed benchmark suite that tests the capabilities of language models in handling extended documents and complex information sequences. For more details on LongBench, please refer to the APPENDIX F.

LongBench. We use LongBench [25] to assess the performance of ChunkKV on tasks involving long-context inputs. We evaluated multiple KV cache eviction methods using the LongBench benchmark with LLaMA-3-8B-Instruct [30], Mistral-7B-Instruct-v0.3 [31], and Qwen2-7B-Instruct [32], with a KV cache compression ratio of 10%. LongBench-ZH provides the Chinese subtask, and Qwen2-7B-Instruct also supports Chinese, so we tested Qwen2-7B-Instruct with different KV cache compression methods on the Chinese subtasks.

Tables 6 present the performance gap (in percentage) between each method and the FullKV baseline, where negative values indicate performance degradation compared to FullKV. The table is evaluated in both the LongBench English and Chinese subtasks, where ChunkKV outperforms other compression methods overall. This suggests that ChunkKV's approach of retaining semantic chunks is more effective in preserving important information compared to other discrete token-based compression methods. For the 70B model and Chinese subtask results, please refer to Appendices B.2 and B.5.

Table 6: KV cache compression methods on the LongBench benchmark. Results show performance gap compared to FullKV baseline (negative values indicate worse performance).

Ratio	SLM	H2O	SKV	PKV	**ChunkKV (Ours)**
LlaMa-3-8B-Instruct FullKV: 41.46 ↑					
10%	-13.80%	-10.61%	-3.16%	-3.33%	**-2.29%**
20%	-6.42%	-8.85%	-2.24%	-2.00%	**-1.74%**
30%	-2.36%	-5.38%	-0.07%	-0.22%	**+0.31%**
Mistral-7B-Instruct-v0.3 FullKV: 48.08 ↑					
10%	-16.58%	-9.30%	-3.54%	-3.52%	**-2.85%**
Qwen2-7B-Instruct FullKV: 40.71 ↑					
10%	-5.28%	-0.64%	-0.39%	-0.98%	**+0.42%**
Qwen2-7B-Instruct on LongBench-ZH FullKV: 38.60 ↑					
10%	-15.95%	-5.31%	+0.18%	-5.31%	**+2.20%**

Needle-In-A-HayStack. We use NIAH [26] to evaluate the long-context retrieval capability of LLMs. NIAH assesses how well LLM extracts hidden tricked information from extensive documents, and following LLM-as-a-Judge [43] we apply GPT-4o-mini [44] to assess the accuracy of the retrieved information. We evaluated multiple KV cache eviction methods using NIAH with LLaMA-3-8B-Instruct and Mistral-7B-Instruct-v0.2, setting benchmark context lengths to 8k and 32k tokens.

Table 7 provides statistical results for different compression methods. These findings clearly indicate the effectiveness of ChunkKV in managing varying token lengths and depth percentages, making it a robust choice for KV cache management in LLMs. Figure 3 presents the NIAH benchmark results for LLaMA-3-8B-Instruct. The vertical axis represents the depth percentage, while the horizontal axis represents the token length, with shorter lengths on

Table 7: NIAH Performance Comparison.

KV cache Size	SLM	H2O	SKV	PKV	**ChunkKV (Ours)**
LlaMa-3.1-8B-Instruct FullKV: 74.6% ↑					
512	32.0%	68.6%	71.2 %	72.6%	**74.5%**
256	28.0%	61.7%	68.8%	69.5%	**74.1%**
128	23.7%	47.9%	58.9%	65.1%	**73.8%**
96	21.5%	41.0%	56.2%	63.2%	**70.3%**
Mistral-7B-Instruct FullKV: 99.8% ↑					
128	44.3%	88.2%	91.6%	99.3%	**99.8%**

the left and longer lengths on the right. A cell highlighted in green indicates that the method can retrieve the needle at that length and depth percentage. The detailed visualization of the NIAH benchmark can be found in the Appendix B.3. The visualization results demonstrate that ChunkKV outperforms other KV cache compression methods.

4.3 Index Reuse

This section will evaluate the performance of the layer-wise index reuse approach with ChunkKV from the two aspects of efficiency and performance.

Measuring Efficiency. We evaluated the latency and throughput of ChunkKV compared to FullKV using LLaMA3-8B-Instruct on an A40 GPU. All experiments were conducted with reuse layer is 2, batch size set to 1 and inference was performed using Flash Attention 2, each experiment was repeated 10 times and the average latency and throughput were reported.

The results in Table 8 show that the layer-wise index reuse strategy (ChunkKV_reuse) further boosts performance, achieving up to a 20.7% reduction in latency, and throughput improvements are particularly notable for longer input sequences, with ChunkKV_reuse delivering up to a 26.5% improvement over FullKV. For more detailed results, please refer to Appendix B.8.

Table 8: Latency and throughput comparison between ChunkKV and FullKV under different input-output configurations. Percentages in parentheses indicate improvements over FullKV baseline.

Method	Sequence Length		Performance Metrics	
	Input	Output	Latency(s) ↓	Throughput(T/S) ↑
FullKV	4096	1024	43.60	105.92
ChunkKV	4096	1024	37.52 (13.9%)	118.85 (12.2%)
ChunkKV_reuse	4096	1024	**37.35** (14.3%)	**124.09** (17.2%)
FullKV	4096	4096	175.50	37.73
ChunkKV	4096	4096	164.55 (6.2%)	40.58 (7.6%)
ChunkKV_reuse	4096	4096	**162.85** (7.2%)	**41.12** (9.0%)
FullKV	8192	1024	46.48	184.08
ChunkKV	8192	1024	37.83 (18.6%)	228.96 (24.4%)
ChunkKV_reuse	8192	1024	**36.85** (20.7%)	**232.99** (26.5%)
FullKV	8192	4096	183.42	55.93
ChunkKV	8192	4096	164.78 (10.2%)	65.14 (16.5%)
ChunkKV_reuse	8192	4096	**162.15** (11.6%)	**66.05** (18.1%)

Measuring Task Performance. We evaluate the effectiveness of our proposed layer-wise index reuse technique on LongBench [25] and GSM8K [27] benchmarks. For these experiments, we use the same configuration as our main LongBench experiments in Section 4.2, with index reuse applied to consecutive layers (reuse layers = 2).

As shown in Table 9, layer-wise index reuse maintains the performance of the models while reducing computational requirements. In LongBench, performance degradation is minimal (less than 0.6%) for both models, while GSM8K shows neutral or slightly positive effects. This validates that semantic chunks selected by ChunkKV remain consistently important across adjacent transformer layers, enabling efficient computation without sacrificing accuracy. An additional analysis of different reuse depths and their impact on throughput is provided in Appendix B.1.2.

Overall, these findings on efficiency and performance suggest that layer-wise index reuse can be an effective technique for optimizing the efficiency-performance trade-off in KV cache compression, with the potential for model-specific tuning to maximize benefits.

Table 9: Reusing Indexing Performance Comparison on LongBench and GSM8K. △ indicates the performance degradation of index reuse compared to the baseline.

Model	ChunkKV	
	Baseline	Index Reuse$_{\triangle}$
LongBench		
LLaMA-3-8B-Inst	40.51	$40.27_{-0.59\%}$
Mistral-7B-Inst	46.71	$46.43_{-0.59\%}$
Qwen2-7B-Inst	40.88	$40.76_{-0.29\%}$
GSM8K		
LLaMA-3-8B-Inst	74.5	$74.6_{+0.13\%}$
Qwen2-7B-Inst	71.2	$71.2_{+0.00\%}$

4.4 Chunk Size

This section aims to investigate the impact of chunk size on the performance of ChunkKV. Different chunk sizes will lead to varying degrees of compression on the semantic information of the data. We used the same experiment setting as in LongBench and NIAH (Section 4.2). The chunk size is set from the range $\{3, 5, 10, 20, 30\}$ under the compression rate 10% for LongBench and the 128 KV cache size for NIAH. For more experiments with different compression ratios, see Appendix B.4.

Table 10: LongBench and NIAH Results with Chunk Size Ablation

Model	Full KV	Chunk KV					SnapKV	H2O
		size=3	size=5	size=10	size=20	size=30		
LongBench ↑								
LLaMA-3-8B-Instruct	41.46	40.49	40.47	**40.51**	40.05	39.57	40.15	37.06
Mistral-7B-Instruct	48.08	46.45	46.51	**46.71**	46.42	45.98	46.38	43.61
NIAH ↑								
LLaMA-3-8B-Instruct	74.6	65.6	69.1	**73.8**	72.0	71.2	58.9	47.9
Mistral-7B-Instruct	99.8	98.1	99.2	**99.8**	99.8	99.1	91.6	88.2

As shown in Table 10, the performance remains relatively stable when the chunk size is between 5 and 20, with the best results consistently achieved at chunk size 10. When the chunk size is too small (e.g., 3), the context is fragmented, leading to a slight drop in performance. Conversely, when the chunk size is too large (for example, 30), the semantic granularity becomes too coarse, and important fine-grained information may be lost, also resulting in performance degradation. This trend is consistent across both LongBench and NIAH benchmarks, as well as across different model architectures, indicating that the optimal chunk size is not highly sensitive to the specific task or model. We also provide a line graph in Appendix B.4 to visually illustrate this trend. Based on these findings, we recommend using a chunk size of 10 as a robust default for most applications. For users with specific requirements, chunk size can be further tuned, but our results suggest that moderate values (5-20) generally offer a good trade-off between semantic preservation and compression efficiency.

4.5 Comparing with KV Quantization

For comprehensively evaluate the effectiveness of ChunkKV, we performed experiments comparing ChunkKV with the KIVI quantization methods [45]. Although both approaches aim to optimize LLM inference, they operate on fundamentally different principles: Quantization reduces KV matrix precision, whereas our eviction method reduces KV matrix size. For more detail, see Appendix B.6.

From an implementation perspective, quantization methods require the full KV cache during prefilling to produce quantized representations, which are then used during decoding. In contrast, ChunkKV employs token removal prior to prefilling, enabling operation with a compressed cache throughout the entire inference process. This distinction creates different efficiency profiles, and each method offers unique advantages.

Due to the fact that KIVI requires an old Python version, the ChunkKV results are not aligned with Table 8. The efficiency results in Table 11 reveal ChunkKV's significant advantages in latency-critical metrics. Although both methods substantially reduce cache size (ChunkKV to 10% and KIVI-2bits to 15.63%), ChunkKV delivers superior metrics for Time to First Token (TTFT) and Token Processing Time (TPOT). Most remarkably, ChunkKV achieves a 164.66s total generation time compared to KIVI's 226.52s at 2-bit quantization, representing a 27.3% improvement in overall inference speed.

Table 11: Efficiency Results for ChunkKV and KIVI on LlaMa-3-8B-Instruct

Configuration	Prompt Length	Output Length	Compression Ratio / nbits	Prefilling Time(s) ↓	Cache Size(GB) ↓	TTFT (s) ↓	TPOT (ms) ↓	Total Gen. Time(s) ↓
FullKV	8192	4096	-	1.5621	1.0000	1.6013	45.9421	184.2934
KIVI	8192	4096	2bits	1.4024	0.1563	1.4325	54.9561	226.5234
KIVI	8192	4096	4bits	1.3916	0.2813	1.4146	52.0510	214.5634
ChunkKV	8192	4096	10%	1.3653	0.1000	1.3914	39.8702	164.6600

4.6 Analysis of Hybrid Compression: Chunk-level vs. Token-level at Different Layer Depths

A critical question raised during the review process was whether the benefits of chunk-level compression diminish in deeper Transformer layers, where semantic information becomes more abstract and diffused. To investigate this, we designed a hybrid compression model that applies different strategies at varying network depths.

Experimental Setup We created a hybrid version of the LLaMA-3-8B-Instruct model. For the first 16 layers (bottom half), we applied our chunk-based ChunkKV. For the final 16 layers (top half), we applied SnapKV, a state-of-the-art token-level compression method. We then compared this hybrid model's performance on the diverse LongBench benchmark against pure ChunkKV and pure SnapKV at identical compression ratios.

Table 12: Performance of the hybrid compression model on LongBench. While the hybrid model shows strengths in global understanding tasks, pure ChunkKV achieves the best overall performance, validating the robustness of preserving semantic chunks even in deep layers.

Method	Single-Doc QA	Multi-Doc QA	Summarization	Few-shot	Synthetic	Code	Avg. Score ↑
FullKV	32.19	34.59	24.96	68.48	36.96	54.41	41.46
SnapKV (10% Ratio)	28.11	32.55	24.12	67.81	36.01	55.67	40.15
Hybrid Model (10% Ratio)	28.38	30.37	**24.54**	**67.87**	36.37	55.32	39.80
ChunkKV (10% Ratio)	**28.50**	**33.46**	22.20	67.62	**37.47**	**58.98**	**40.51**

Analysis and Insights The results in Table 12 provide two key insights. First, the pure ChunkKV model achieves the highest overall average score. This empirically validates our core hypothesis: preserving local semantic integrity via chunking is a robust and effective strategy across all layers of the model, even where information is highly processed.

Second, the hybrid model reveals a fascinating, task-dependent performance trade-off.

- **For local information retrieval tasks** (e.g., Single- and Multi-Document QA), pure ChunkKV is significantly superior. These tasks often require retrieving precise, intact text fragments. ChunkKV's ability to preserve complete linguistic units prevents critical information from being fragmented, which is essential in these scenarios.
- **For global understanding tasks** (e.g., Summarization and Few-shot Learning), the hybrid model performs best. In these tasks, synthesizing information from across the entire context is key. In the deeper layers, where abstract representations are formed, the token-level SnapKV method may be more adept at retaining a broader, more diffuse set of globally important signals.

This analysis does not undermine our approach but rather enriches it, suggesting that the future of KV cache compression may lie in adaptive, task-aware strategies. However, for a general-purpose and robust solution, pure ChunkKV proves to be the most effective.

5 Conclusion

In this paper, we first indicate that current KV cache methods lack the semantic information of the data, and then we proposed a novel KV cache compression method that preserves semantic information by retaining more informative chunks. Through extensive experiments across multiple state-of-the-art LLMs (including DeepSeek-R1, LLaMA-3, Qwen2, and Mistral) and diverse benchmarks (GSM8K, LongBench, NIAH, and JailbreakV), we demonstrate that ChunkKV consistently outperforms existing methods while using only a fraction of the memory. Our comprehensive analysis shows that ChunkKV's chunk-based approach maintains crucial contextual information, leading to superior performance in complex reasoning tasks, long-context understanding, and safety evaluations. The method's effectiveness is particularly evident in challenging scenarios like many-shot GSM8K and multi-document QA tasks, where semantic coherence is crucial. Furthermore, our proposed layer-wise index reuse technique provides significant computational efficiency gains with minimal performance impact, achieving up to 20.7% latency reduction and 26.5% throughput improvement. These findings, supported by detailed quantitative analysis and ablation studies, establish ChunkKV as a significant advancement in KV cache compression technology, offering an effective solution for deploying LLMs in resource-constrained environments while maintaining high-quality outputs.

Acknowledge

This work was supported by the NSFC grant 62432008; RGC RIF grant R6021-20; an RGC TRS grant T43-513/23N-2; RGC CRF grants C7004-22G, C1029-22G and C6015-23G; NSFC/RGC grant CRS_HKUST601/24 and RGC GRF grants 16207922, 16207423 and 16203824; National

Natural Science Foundation of China (Grant No.62506318); Guangdong Provincial Department of Education Project (Grant No.2024KQNCX028); Scientific Research Projects for the Higher-educational Institutions (Grant No.2024312096), Education Bureau of Guangzhou Municipality; Guangzhou-HKUST(GZ) Joint Funding Program (Grant No.2025A03J3957), Education Bureau of Guangzhou Municipality;the Guangzhou Municipal Joint Funding Project with Universities and Enterprises under Grant No. 2024A03J0616 and Guangzhou Municipality Big Data Intelligence Key Lab (2023A03J0012).

References

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[2] Yi Tay, Mostafa Dehghani, Vinh Q Tran, Xavier Garcia, Dara Bahri, Tal Schuster, Huaixiu Steven Zheng, Neil Houlsby, and Donald Metzler. Unifying language learning paradigms. *ArXiv preprint*, abs/2205.05131, 2022. URL https://arxiv.org/abs/2205.05131.

[3] Zhenheng Tang, Xiang Liu, Qian Wang, Peijie Dong, Bingsheng He, Xiaowen Chu, and Bo Li. The lottery LLM hypothesis, rethinking what abilities should LLM compression preserve? In *The Fourth Blogpost Track at ICLR 2025*, 2025.

[4] Qian Wang, Zhenheng Tang, Zichen Jiang, Nuo Chen, Tianyu Wang, and Bingsheng He. Agenttaxo: Dissecting and benchmarking token distribution of llm multi-agent systems. In *ICLR 2025 Workshop on Foundation Models in the Wild*, 2025.

[5] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *ArXiv preprint*, abs/2307.09288, 2023. URL https://arxiv.org/abs/2307.09288.

[6] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

[7] Sam Ade Jacobs et al. DeepSpeed Ulysses: System optimizations for enabling training of extreme long sequence Transformer models. *ArXiv preprint*, abs/2309.14509, 2023. URL https://arxiv.org/abs/2309.14509.

[8] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=NG7sS51zVF.

[9] Hao Liu, Wilson Yan, Matei Zaharia, and Pieter Abbeel. World model on million-length video and language with ringattention. *ArXiv preprint*, abs/2402.08268, 2024. URL https://arxiv.org/abs/2402.08268.

[10] Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, et al. Yi: Open foundation models by 01. ai. *ArXiv preprint*, abs/2403.04652, 2024. URL https://arxiv.org/abs/2403.04652.

[11] 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. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023.

[12] Yuanbing Zhu, Zhenheng Tang, Xiang Liu, Ang Li, Bo Li, Xiaowen Chu, and Bo Han. OracleKV: Oracle guidance for question-independent KV cache compression. In *ICML 2025 Workshop on Long-Context Foundation Models*, 2025. URL https://openreview.net/forum?id=KHM2YOGgX9.

[13] Qian Wang, Tianyu Wang, Zhenheng Tang, Qinbin Li, Nuo Chen, Jingsheng Liang, and Bingsheng He. Megaagent: A large-scale autonomous llm-based multi-agent system without predefined sops. In *The 63rd Annual Meeting of the Association for Computational Linguistics*, 2025.