SCOPE: Optimizing Key-Value Cache Compression in Long-context Generation

Anonymous ACL submission

Abstract

Key-Value (KV) cache has become a bottleneck of LLMs for long-context generation. Despite the numerous efforts in this area, the optimization for the decoding phase is generally ignored. However, we believe such optimization is crucial, especially for long-output generation tasks based on the following two observations: *(i) Excessive compression* during the prefill phase which requires specific full context, impairs the comprehension of the reasoning task; *(ii) Deviation of heavy hitters*[1] occurs in the reasoning tasks with long outputs. Therefore, **SCOPE**, a simple yet efficient framework that separately performs KV cache optimization during the prefill and decoding phases, is introduced. Specifically, the KV cache during the prefill phase is preserved to maintain the essential information, while a novel strategy based on sliding is proposed to select essential heavy hitters for the decoding phase. Memory usage and memory transfer are further optimized using adaptive and discontinuous strategies. Extensive experiments on LONGGENBENCH show the effectiveness and generalization of SCOPE and its compatibility as a plug-in to other prefill-only KV compression methods. [2]

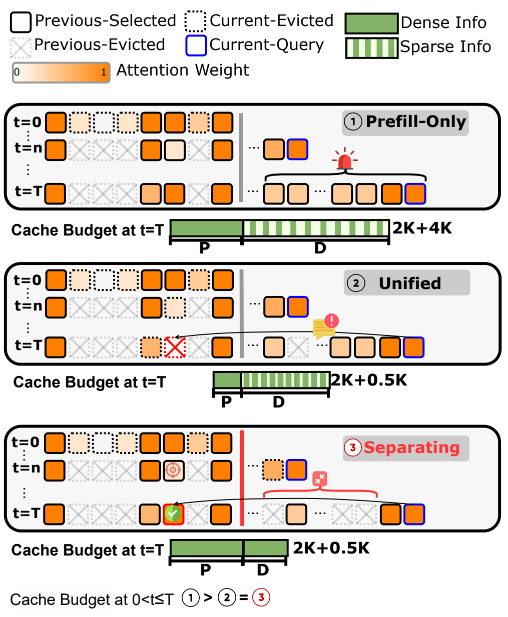


Figure 1: Illustration of three paradigms for compression during the decoding phase on a task with 4K input and 4K output. Separating the prefill and decoding phases facilitates the preservation of the essential information KV cache from the prefill phase while allowing for efficient allocation of the KV cache generated during the decoding phase.

1 Introduction

Large Language Models (LLMs) (Dubey et al., 2024; Jiang et al., 2023; Yang et al., 2024a; Team et al., 2024; Achiam et al., 2023; Anthropic, 2024) have demonstrated powerful abilities for processing long-context tasks. When LLMs infer on these long-context tasks, the Key-Value (KV) cache occupies a larger amount of GPU memory and becomes a substantial bottleneck (Waddington et al., 2013; Luohe et al., 2024; Yuan et al., 2024; Fu, 2024). For example, an RTX 3090 server struggles to handle the KV cache for a 64K context in LLaMA3.1-8B, which has a 128K context window. Therefore, compressing the KV cache while maintaining the performance is crucial.

LLM inference process involves the *prefill* phase and the *decoding* phase. For tasks with long inputs and short output (Kamradt, 2023; Bai et al., 2024) (*e.g.*, long-form QA or sentence retrieve), effective compression of the KV cache during the prefill phase is crucial. However, for tasks with both long inputs and long outputs (Liu et al., 2024b,c) (*e.g.*, lengthy text summarization and multi-question answering), KV cache compression holds equal importance in both the prefill and decoding phases.

Previous methods fall into two categories: (1)

[1]According to Zhang et al. (2023), "*heavy hitters*" refer to the KV cache of pivotal tokens, a small subset of the entire KV cache, that effectively captures the critical information.

[2]The code is available in `https://anonymous.4open.science/r/SCOPE-7235`

The *Prefill-Only Compression* method compresses the KV cache only during the prefill phase while retaining all KV cache generated during the decoding phase. (2) The *Unified Compression* method treats both phases as a unified process. For *Prefill-Only Compression*, methods like SnapKV (Li et al., 2024) and PyramidKV (Cai et al., 2024), retaining all KV cache generated during the decoding phase, leading to linear cache growth with the output length and memory pressure 🚨, especially for long outputs, as shown in Figure 1. For *Unified Compression*, such as H_2O (Zhang et al., 2023) and PyramidInfer (Yang et al., 2024b), prioritizes retaining the KV cache generated during decoding while discarding the earlier KV cache influenced by recent tokens typically receiving higher attention weights (Zhao et al., 2021; Song et al., 2024). This poses substantial challenges for reasoning tasks that rely on understanding the whole input content 📩. There has been no dedicated exploration of KV cache compression strategies for handling lengthy outputs.

In this paper, we first unravel two essential observations that serve as the foundation for our exploration: ***(i)*** excessive compression during the prefill phase significantly affects the ability of LLM to reason through the query; ***(ii)*** heavy hitters deviate during the decoding phase in long-text generation, leading to skewed KV cache allocation. Building upon the insight, we introduce 🔭**SCOPE**, a simple yet efficient framework that **S**eparately performs KV **C**ache **O**ptimization during the **P**refill and d**E**coding phases. To our knowledge, we are the **first** to decouple the prefill and decoding phases to compress the KV cache independently. Specifically, we first maintain the KV cache generated during the prefill phase to ensure an understanding of long content. Then, we allocate heavy hitters using the sliding way in the decoding phase to optimize the memory of the KV cache. Building on the intuitive slide strategy, we further optimize *memory-usage* and *memory-transfer*, introducing adaptive strategy and discontinuous strategy.

To thoroughly validate our framework, we select LongGenBench (Liu et al., 2024c) as the benchmark for our experiments over two mainstream LLMs. **SCOPE** can achieve comparable performance to the full KV cache when the overall compression rate is 35%. Additionally, our framework is seamlessly compatible with other compression methods in the prefill phase.

The contributions of this work are as follows: 1). A simple yet efficient framework SCOPE is proposed to address the deviation of heavy hitters inspired by the observations and insights from an inference perspective. 2). Three strategies are developed to mitigate the deviation during the decoding phase. 3). Empirically, extensive experiments and analytical evaluations validate the effectiveness and generalizability of SCOPE.

2 Pilot Observation

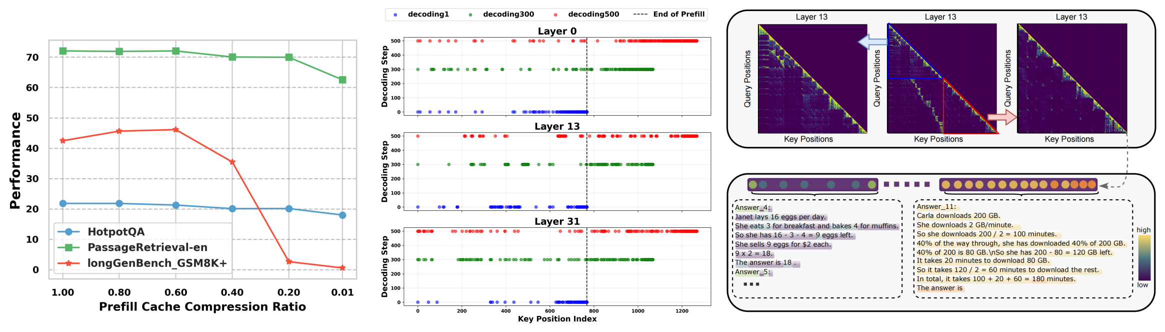
2.1 KV Cache in Inference Perspective

Each request for an LLM involves two distinct phases (Zhou et al., 2024). The first phase, known as ***prefill***, processes the complete input prompt to generate the initial output token. The second phase, termed ***decoding***, iteratively produces the remaining output tokens, one at a time. We conduct pilot experiments through the lens of each phase in the inference process.

Prefill Phase: Existing work focusing on the prefill phase is grounded in the notion that attention is naturally sparse in typical tasks (Singhania et al., 2024; Tang et al., 2024; Wu et al., 2024). For *PassageRetrieval-en* and HotpotQA within LongBench, a 20% compression ratio during the prefill phase still maintained performance nearly identical to that of the full cache, demonstrating the model's ability to effectively retrieve and understand context even with significant compression, as shown in Figure 2a. However, when tasks require specific full context, such as reasoning tasks, attention is **not** always highly sparse (Chen et al., 2024), even if the output is short. As illustrated in Figure 2a, the same 20% compression rate during the prefill phase resulted in nearly 95% degradation in accuracy on the GSM8k+ task within LongGenBench. Although sufficient performance is achieved on conventional tasks using KV cache compress during the prefill phase, the performance is notably poor on reasoning tasks when the compression ratio reaches a modest threshold, leaving room for targeted optimization through compression during the decoding phase.

Observations (i): For tasks that require specific full context, such as reasoning tasks, excessive compression during the prefill phase significantly compromises performance.

Decoding Phase: We analyze the distribution of heavy hitters during the prefill and decoding phases



Answer_4:
Janet lays 16 eggs per day.
She eats 3 for breakfast and bakes 4 for muffins.
So she has 16 - 3 - 4 = 9 eggs left.
She sells 9 eggs for $2 each.
9 x 2 = 18.
The answer is 18 .
Answer_5:
...

Answer_11:
Carla downloads 200 GB.
She downloads 2 GB/minute.
So she downloads 200 / 2 = 100 minutes.
40% of the way through, she has downloaded 40% of 200 GB.
40% of 200 is 80 GB.\nSo she has 200 - 80 = 120 GB left.
It takes 20 minutes to download 80 GB.
So it takes 120 / 2 = 60 minutes to download the rest.
In total, it takes 100 + 20 + 60 = 180 minutes.
The answer is

(a) (b) (c)

Figure 2: (a) Performances across various compression ratios during the prefill phase on three tasks under the full decoding cache condition. (b) Position distribution of the heavy hitters, selected by top 15% attention scores, at decoding steps 1, 300, and 500 across layers 0, 13, and 31. (c) Attention heatmaps for layer 13 of a GSM8k+ sample in LONGGENBENCH and details of the correspondence between attention scores and generated token positions. The complete case employed in the probing experiment is presented in Appendix 6.

as the decoding length increased in Figure 2b. Across all three layers, the retained heavy hitters predominantly originate from the KV cache generated during the decoding phase. This phenomenon has also been mentioned by several recent studies and can be attributed to the inherent properties of the attention mechanism, wherein tokens near the end often receive higher attention weights (Zhao et al., 2021; Song et al., 2024). This is particularly harmful for multi-question answering tasks, like LONGGENBENCH, as addressing such queries needs careful consideration of the question context. Previous prefill-only or unified compression strategies may overlook this distinction. In long-output tasks, as the output length increases, the deviation becomes more pronounced, making it imperative to preserve the heavy hitters identified during the prefill phase while providing appropriate management for those emerging in the decoding stage.

Observations (ii): During the decoding phase of long text generation, the use of the greedy algorithm may lead to a deviation in heavy hitters.

2.2 KV Cache Budget Reallocation

Building on the empirical observations from our pilot experiments, we derive the following insight:

Insight: It is crucial to allocate the budget of the KV cache during the prefill and decoding phases **separately**.

This insight inspires the design of SCOPE, which decouples compression into the prefill and decoding phases to effectively allocate the KV cache budget, preserving all KV cache generated during the prefill phase and enabling more effective reallocation of the KV cache budget. While numerous studies have explored the heavy hitters during the prefill phase, to our knowledge, no prior work has specifically investigated this aspect of the decoding phase. We dive deeper into the sparsity in the KV cache during the decoding to design strategy, selecting essential heavy hitters dynamically. To gain deeper insights, following prior works (Xiao et al., 2024b; Cai et al., 2024), we analyze the attention heatmaps, comparing the attention weights between the prefill and decoding phases, as shown in Figure 2c. The leftmost and rightmost plots represent the prefill and decoding phases, respectively. For tasks that require simultaneous reasoning for multiple questions, it is essential to recognize the position of the current prediction. This information can be captured by heavy hitters identified using a greedy algorithm, as illustrated in Figure 2c. Thus, it remains necessary to allocate a portion of the KV cache budget specifically for heavy hitters. Furthermore, owing to the autoregressive nature of LLMs, it remains essential to retain the recent tokens, which exhibit stronger correlations with current tokens.

3 Method

3.1 Revisiting KV Cache Compression

Initialization KV cache compression essentially involves adjusting the cache based on the given KV cache budget, where we allocate a cache pool, denoted as Φ, consisting of Φ^p and Φ^d, which stores the KV cache generated during the prefill and decoding phases, respectively. The cache pool is updated at each step t, denoted as Φ_t. The widely recognized function for selecting heavy hitters based on the greedy algorithm is denoted as $\Psi_K(\mathbf{Att})$, which represents the selection of the Top-K KV caches from the given attention weights $\mathbf{Att}$.

Prefill Phase Given the input prompt tensor $\mathbf{P} \in \mathbb{R}^{M\times D}$, represented as $\mathbf{P} = \{\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_M\}$, where $\mathbf{P}_i$ denote i-th token embeddings, and M represent the number of input tokens and D is the model's hidden dimension. The key and value tensors are computed as follows:

$$\mathbf{K}_\mathcal{P}\mathbf{V}_\mathcal{P} = \mathbf{P}\mathbf{W}_K, \mathbf{P}\mathbf{W}_V, \tag{1}$$

where $\mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{D\times D}$ are the weights matrices for the key and value projections, respectively. The KV pairs are denoted as $\mathbf{K}_\mathcal{P}\mathbf{V}_\mathcal{P}$. The attention weights $\mathbf{Att}_\mathcal{P}$ is caculated by $\mathbf{P}$ and $\mathbf{K}_\mathcal{P}\mathbf{V}_\mathcal{P}$. The most effective and widely adopted approach, as established through early explorations (Zhang et al., 2023; Yang et al., 2024b; Li et al., 2024), two import hyperparameters α_1 and α_2 are introduced, where α_1 represents the length of *prefill essential history window* and α_2 represents the length of *prefill local window* during the prefill phase. The length of the total reserved KV cache is $\alpha_1 + \alpha_2$, which also corresponds to the size of the cache pool Φ^p during the prefill. For compression during the prefill phase is:

$$\mathbf{K}_0\mathbf{V}_0 = \Psi_{\alpha_1}(\mathbf{Att}_\mathcal{P}[:-\alpha_2]) \cdot \mathbf{K}_\mathcal{P}\mathbf{V}_\mathcal{P}[-\alpha_2:], \tag{2}$$

where $\cdot$ denotes concatenation and the function $\Psi_{\alpha_1}(\mathbf{Att}_\mathcal{P})$ selects the KV cache with the Top-α_1 attention weights from $\mathbf{Att}_\mathcal{P}[:-\alpha_2]$.

$\mathbf{K}_0\mathbf{V}_0$ is stored in Φ^p_0. Maintain an *essential history window* α_1 to retain KV with higher attention weights for the current query and a *local window* α_2 to reserve the KV of recently generated tokens, ensuring both contextual continuity and retention of attention. Notably, the compression is only executed once, at $t = 0$, marking the end of the prefill phase before transitioning into the decoding phase.

Decoding Phase During the decoding phase, the KV cache from the prefill phase is employed and updated to sequentially generate tokens. At each time step t, keys and values are computed only for the new token tensor $\mathbf{X}_{t,t\in\{1,T\}}$ as follows:

$$\mathbf{K}_t\mathbf{V}_t = \mathbf{X}_t\mathbf{W}_K, \mathbf{X}_t\mathbf{W}_V, \tag{3}$$

$\mathbf{K}_t\mathbf{V}_t$ is concatenated with previously retained KV cache, which is stored in Φ, to obtain the current retained KV pairs. This is then computed with the current query $\mathbf{X}_t$ to compute the attention $\mathbf{Att}_t$.

The main difference from previous KV compression methods lies in the distribution of Φ^p and Φ^d within the cache pool Φ. The *Prefill-Only Compression* method does not compress the KV cache generated during the decoding phase. Instead, it involves a linear growth of the KV cache with each newly generated token. Φ^t_p remains constant, and at each step t, it stores the originally preserved $\mathbf{KV}_0$. Φ^t_d stores the KV cache at each time step t during the decoding phase, from $\mathbf{K}_1\mathbf{V}_1$ to $\mathbf{K}_T\mathbf{V}_T$, which leads to a significant increase of memory consumption as the length grows. The *Unified Compression* method in the decoding phase will apply the $\Psi_{\alpha_1}(\mathbf{Att}_t[:-\alpha_2])$ at each t to update cache pool Φ. As the number of generated tokens increases, the attention mechanism tends to assign higher weights to tokens at the end, meaning that the Top-α_1 KV caches returned by Ψ are all generated during the decoding phase, while those from the prefill phase are discarded. As t increases, Φ^p_t grows larger, while Φ^d_t becomes smaller. This results in more information being retained in Φ^d within Φ, while the information in Φ^p decreases, leading to potential essential information loss that may be needed in future decoding steps.

3.2 SCOPE

The primary goal of **SCOPE** is to mitigate the deviation of heavy hitters, thereby ensuring a more balanced allocation of Φ^p and Φ^d. Motivated by the findings in §2.1, where excessive compression during the prefill phase hinders performance on reasoning tasks, the cache pool Φ_p is constant at each t, *i.e.*, we reserved all compressed KV Cache generated during the prefill phase. The operation on Φ^p in **SCOPE** is the same as that in the previously unified compression method.

It is necessary to leverage the sparsity of the KV cache generated during decoding to enable efficient allocation. Three strategies for the decoding phase are developed: **Slide**, **Adaptive**, and **Discontinuous**, all of which update only Φ_d. The adaptive strategy optimizes memory based on the slide strategy, and the discontinuous strategy optimizes computation on top of the adaptive one. We will introduce them one by one below in detail. For each strategy, Python-style pseudocode is provided in Figure 5 to facilitate comprehension of details.

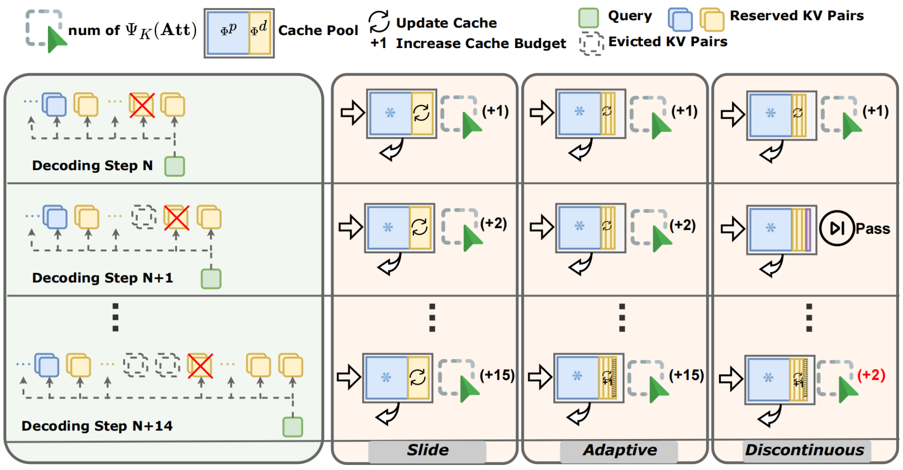


Figure 3: Illustration of three strategies of **SCOPE**. The prefilled cache pool Φ_t^p is constant at each t. The slide strategy updates the decoding cache pool at each decoding step while the size of the decoding cache pool is constant. The adaptive strategy incrementally increases the size of the decoding cache pool at regular intervals of $\frac{T-\beta_2}{\beta_1}$. The discontinuous strategy, built upon the adaptive strategy, executes $\Psi_K(\mathbf{Att})$ at intervals of the same time period.

Slide We compress the KV cache in the decoding phase by sliding the *decoding essential history window* β_1 and the *decoding local window* β_2, where β_1 helps identity the position of the current prediction and β_2 stores global information strongly correlated with previous tokens other as discussed in §2.2.

The slide strategy starts from $t > M + \beta_1 + \beta_2$, applying the function $\Psi_{\beta_1}(\mathbf{Att}_t[\alpha_1 + \alpha_2 : -\beta_2])$ to restrictively update only Φ^d while keeping Φ^p constant. This is achieved by limiting the selecting function Ψ to operate on $\mathbf{Att}_t$ starting from $\alpha_1 + \alpha_2$, thus excluding the attention weights from the prefill phase. It can be completely independent of the KV cache pool during the prefill phase Φ^p.

Adaptive We can optimize the β_1 of slide strategy to adaptively increase its size from a ***memory-usage*** perspective. When the length of the tokens generated during decoding is relatively short, a long *decoding essential history window* β_1 is unnecessary, it is unnecessary to place all these KV caches in the Φ_t^d. β_1 can be adaptively increased as needed. A function of time steps t and the maximum length T is proposed to adaptively adjust the length of the *decoding essential history window* β_1, where $T \gg \beta_1 + \beta_2$. It starts with a base size β_2 and grows linearly with time step t:

$$\hat{\beta}_1 = \frac{(t-\beta_2)\cdot\beta_1}{T-\beta_2} \quad \text{if } t > \beta_2, \tag{4}$$

The budget size of Φ_t^d also increases adaptively and is given by $\beta_2 + \frac{(t-\beta_2)\cdot\beta_1}{T-\beta_2}$ when $t < T$, which helps optimize memory usage, as the ratio $\frac{(t-\beta_2)}{(T-\beta_2)}$ is less than 1. As t reaches T, the size of Φ_t^d becomes $\beta_1 + \beta_2$. This adjustment aligns with the autoregressive token-by-token encoding characteristic of LLMs, ensuring more efficient use of resources. In addition, $\Psi_K(\mathbf{Att})$ begins execution earlier than the sliding strategy. The adaptive strategy optimizes the budget of Φ_t^d and reduces unnecessary overhead while still retaining enough historical context for an effective generation without introducing additional hyperparameters.

Discontinuous We further optimize the adaptive strategy from a ***memory-transfer*** perspective to ensure by reducing the frequency of execution of $\Psi_K(\mathbf{Att})$. The top-K selection operation $\Psi_K(\mathbf{Att})$ would be executed a total of $T - \beta_2$ times using previous strategies, which potentially leads to frequent GPU I/O due to the update operation of Φ_d at each step. Motivated by the characteristic that consecutive queries tend to select similar keys (Zhao et al., 2024; Tang et al., 2024), we make the update operation, *i.e.*, Top-K selection $\Psi_K(\mathbf{Att})$ discontinuous. This strategy optimizes the times of execution frequency of selection operation $\Psi_K(\mathbf{Att})$, with ζ occurring once every interval of $\frac{T-\beta_2}{\beta_1}$, whereas previous strategies execute at each step t. This interval is consistent with the growth of $\hat{\beta}_1$ in the adaptive strategy. The frequency can be reduced to $\frac{T-\beta_2}{\frac{T-\beta_2}{\beta_1}} = \beta_1$ using this strategy, thereby alleviating the memory I/O pressure caused by frequent updates to Φ_d.

Table 1: Performance of our proposed SCOPE using three strategies and baselines on the LongGenBench benchmark with LLaMA-3.1-8B-Instruct. The best results among all methods are in **bolded**. The prefill compression ratio averages around 60%. Table 6 shows the results on Mistral-7B-Instruct-v0.3.

Method	LongGenBench-4K				LongGenBench-8K			
	GSM8K+	MMLU+	CSQA+	Avg.	GSM8K++	MMLU++	CSQA++	Avg.
Full Cache	42.50	54.85	71.67	56.34	26.50	51.01	64.92	47.48
	Decoding Compression Ratio=25.0%				Decoding Compression Ratio=12.5%			
StreamingLLM	10.83	30.00	43.67	28.17	10.67	27.87	44.92	27.82
H_2O	37.33	46.02	69.50	50.95	18.17	48.21	59.58	41.99
PyramidInfer	33.02	45.25	70.75	49.67	23.00	47.21	58.75	42.99
SCOPE *(Slide)*	**38.83**	46.96	70.75	**52.18**	20.17	**49.87**	**66.17**	45.40
SCOPE *(Adaptive)*	35.00	**49.12**	70.75	51.62	22.00	49.08	63.58	44.89
SCOPE *(Discontinuous)*	35.17	47.13	**72.50**	51.60	**24.33**	48.33	64.83	**45.83**
	Decoding Compression Ratio=12.5%				Decoding Compression Ratio=6.25%			
StreamingLLM	10.83	30.00	43.67	28.17	10.67	28.03	44.92	27.82
H_2O	28.17	44.04	65.25	45.82	16.17	45.94	55.08	39.06
PyramidInfer	30.83	45.23	70.75	48.94	18.33	45.07	60.63	41.34
SCOPE *(Slide)*	39.00	**47.60**	**73.50**	**53.37**	16.83	**48.14**	62.25	42.41
SCOPE *(Adaptive)*	35.33	47.19	72.08	51.53	19.00	47.08	61.83	42.64
SCOPE *(Discontinuous)*	**39.67**	46.73	72.75	53.05	**19.83**	46.54	**66.00**	**44.12**

4 Experiments

4.1 Datasets

We develop two **open-sourced** datasets, LongGenBench-4K ({subtask}+) and LongGenBench-8K ({subtask}++), where multiple reasoning tasks must be handled simultaneously[3], each containing three subtasks synthesized from GSM8K (Cobbe et al., 2021), MMLU (Hendrycks et al., 2021), and CSQA (Hendrycks et al., 2021). These subtasks are designed to address long-input challenges with output lengths of 4K and 8K, respectively.[4] To validate the effectiveness of SCOPE on general long-output tasks, we select the En.Sum task from ∞Bench (Zhang et al., 2024), with an average output length of 1.1K. For the detailed statistics of datasets and additional details corresponding to each subtask, refer to Appendix B.

4.2 Baselines

To validate the effectiveness of SCOPE, we compare it with **Full Cache** and representative unified compression methods, including **StreamingLLM** (Xiao et al., 2024b), which keeps the KV of early and recent tokens; **H_2O** (Zhang et al., 2023), which balances recent and Heavy Hitter (H_2) tokens based on cumulative attention scores; and **PyramidInfer** (Yang et al., 2024b), which reduces the cache in deeper layers using sparse attention patterns. To validate modularity, we apply SCOPE in combination with **SnapKV** (Li et al., 2024) and **PyramidKV** (Cai et al., 2024) during the decoding phase, as detailed in §4.4.

[3]Prompt template is provided in Appendix B.

[4]The selected examples have output lengths of 4K and 8K, ensuring no premature cessation of the response.

4.3 Implementation Details

We build SCOPE using two open-sourced LLMs, specifically LLaMA-3.1-8B-Instruct and Mistral-7B-Instruct-v0.3. Based on the preliminary experiments (Figure 2a), we set the the size of Φ_p, *i.e.*, $\alpha_1 + \alpha_2$ to 2048 for LongGenBench-4K and 4096 for LongGenBench-8K, corresponding to approximately 60% of the input length. α_2 is set to 8 following previous works (Cai et al., 2024; Li et al., 2024). $\beta_1 + \beta_2$ are set to 512 and 1024 in two configurations, corresponding to different compression ratios for outputs of 4K and 8K. β_2 is set to 256 to accommodate the CoT length in answers, avoiding performance loss from overly short sequences. For a fair comparison, the total budget of KV cache during both the prefill and decoding phases is consistent across all methods. More details can be found in the Appendix C.

4.4 Results

Comparison with Baselines Table 1 presents a comprehensive analysis of our proposed SCOPE and baselines. SCOPE (with three strategies) achieves the best results under both decoding compression methods, and the discontinuous strategy, optimized for memory-usage and memory-transfer, delivers outstanding performance. On the challenge GSM8K+/GSM8K++ tasks, SCOPE highlights the importance of preserving the KV cache generated during the prefill process, while other compression methods lead to marked performance degradation. This ensures that the understanding of the problem statement remains intact, achieving comparable performance to the full cache without compromising comprehension. StreamingLLM poses challenges on LongGenBench, where vital information may lie within the middle of the input, consistent with the findings in prior study (Zhang et al., 2023). This inevitably results in the loss of crucial information if only the first few tokens and local tokens are preserved. Performance between PyramidInfer and H_2O shows no notable difference, indicating that the layer-wise sparsity feature is not prominent for tasks with long outputs.

Plug-in to Prefill-Only Methods Table 2 shows the results of seamlessly integrating our decoding phase compression strategy with prefill-only compression methods. Some strategies even outperform the full cache results, despite compressing 35%[5] of the KV cache. This validates the sparsity of the KV cache generated during the decoding phase in multi-QA tasks and demonstrates the effectiveness of our proposed strategies. PyramidKV (Cai et al., 2024), a variant of SnapKV, adjusts the budget allocation across layers without observing improvements in the preliminary experiments, consistent with the empirical finding (§4.4).

Actually, the retained KV cache during the prefill phase can be regarded as "*attention sinks*", which bears a resemblance to the principle of StreamingLLM. We extend this concept to broader, more realistic long-output scenarios.

5 Analysis and Discussion

5.1 Mitigating the Loss of Essential H_2

The unified compression method, such as H_2O, suffers from the loss of crucial KV cache generated during prefill, which is essential to understanding the context due to the deviation of heavy hitters. In Figure 4a, we show the relationship between prediction position and performance. The performance of H_2O drops markedly in later predictions, while all three of our strategies mitigate this decline, validating the effectiveness of preserving the prefill KV cache.

[5]The average input-output length is 7.4K in the GSM8K+ task. With budgets Φ_p of 2K and Φ_d of 0.5K, the total reserved KV cache size is 2.5K, leading to a full compression ratio of about 35%.

Table 2: The plug-in experiment results of LLaMA3.1-8B on the GSM8K+ task from LongGenBench-4K. The results exceeding the full cache are in **bold**.

Decoding Phase Strategy	Prefill Phase		
	Full Cache	SnapKV	PyramidKV
Full Cache	42.50	31.0	29.50
	Decoding Compress Ratio=25.0%		
Slide	**43.00**	25.17	26.50
Adaptive	39.33	**31.33**	**30.83**
Discontinuous	42.17	**31.33**	**29.83**
	Decoding Compress Ratio=12.5%		
Slide	**42.33**	22.33	23.50
Adaptive	36.67	**30.83**	**30.83**
Discontinuous	36.00	29.83	**29.83**

5.2 Influence on $\beta_1 + \beta_2$ and $\Psi_K(\mathbf{Att})$

KV cache budget during the decoding phase $\beta_1+\beta_2$ and selection algorithm $\Psi_K(\mathbf{Att})$ are the key hyperparameters within the SCOPE framework. The budget $\beta_1+\beta_2$, *i.e.*, the compression ratio is scaled using two mainstream top-K selection algorithms as illustrated in Figure 4b. Unlike the prefill phase, where performance on the GSM8k+ task significantly drops as the compression ratio increases, compressing to 25% during the decoding phase only results in a 15% performance decline. It validated that compression in both phases is better than solely focusing on extreme compression during prefill and the necessity of optimizing the KV cache **separately** for the prefill and decoding phases. Using the Top-K selection strategy based on cumulative attention yields better results than the Top-K selection strategy based on the observation window. For tasks like LongGenBench, predictions still require reviewing and capturing the corresponding question, making a short observation window insufficient.

5.3 Efficiency on Memory Usage and Transfer

Our adaptive and discontinuous strategies building on slide strategy improve memory efficiency, as explored in Table 3. Compared to the full cache and prefill-only compression methods, both our method and the unified compression approach effectively reduce memory usage pressure by storing less KV cache overall. Our adaptive strategy further optimizes performance by dynamically adjusting the budget. However, this introduces frequent updates to the stored KV cache pool, leading to increased I/O transfer and latency. The optimized strategy effectively mitigates this issue by executing computations discontinuously.

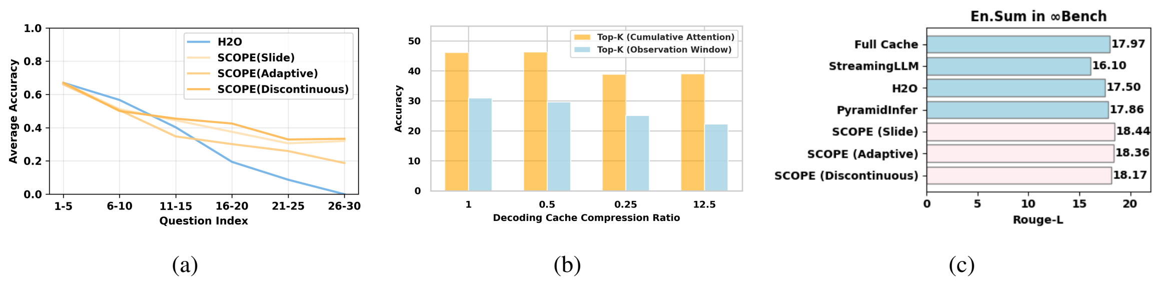


Figure 4: (a) Accuracy distribution of different question positions. (b) Accuracy across different cache compression ratios during the decoding phase using two Tok-K selection algorithms while the KV cache during the prefill phase is compressed to a 60% ratio. Top-K (Observation Window) (Li et al., 2024; Cai et al., 2024), is computed within a fixed-size local window of recent key-value pairs while Top-K (Cumulative Attention) (Zhang et al., 2023; Yang et al., 2024b), attention is computed globally across all key-value pairs. (c) Results on En.Sum task from ∞BENCH, with the condition $\beta_1 + \beta_2 = 512$.

Table 3: Efficiency analysis on Peak KV memory and latency (Lat.) for LLaMA3.1-8B with a prefill compression ratio of 60% and a decoding compression ratio of 12.5%.

Method	Peak KV Mem.	Tokens/s
Full Cache	15.6(100%)	36.57
SnapKV	12.5(80.1%)	38.28
PyramidKV	12.5(80.1%)	36.90
StreamingLLM	5.8(37.1%)	22.02
H_2O		21.78
PyramidInfer		22.38
SCOPE *(Slide)*	5.8(37.1%)	18.28
SCOPE *(Adaptive)*		18.28
SCOPE *(Discontinuous)*		**25.92**

5.4 Generalization of SCOPE

Results of our proposed SCOPE and baselines on ∞BENCH are shown in Figure 4c. All three of our strategies **outperform** the full cache setting, thoroughly validating the generalization of SCOPE. It is effective not only for multi-QA tasks but also for summarization tasks, demonstrating that traditional tasks may also be **well-suited** to the separation of prefill-decoding KV cache budget allocation.

6 Related Work

KV Cache Compression KV cache compression methods focus on leveraging the sparsity in attention to address memory bottlenecks, complementing other efficient techniques (Kwon et al., 2023; Dao, 2024; Wang et al., 2024; Liu et al., 2024d). While recent work has optimized the prefill phase by adjusting the compression budget (Yang et al., 2024b; Feng et al., 2024; Cai et al., 2024), phase-specific optimization remains unexplored. Our approach tailors KV cache compression to the distinct characteristics of each phase, offering a novel perspective.

Long-context Tasks Recent advancements in LLMs have focused on enhancing the capabilities for long-context tasks. Previous evaluations of long-context tasks have mainly concentrated on tasks with long inputs, and numerous benchmarks have been proposed, such as Needle-in-a-Haystack (NIAH) (Kamradt, 2023), LongBench (Bai et al., 2024) and ∞BENCH (Zhang et al., 2024) for comprehensive understanding tasks, where the output is generally short for most sub-tasks. Most research on KV cache compression has been conducted within the context of these benchmarks, where the focus has been primarily on optimizing the prefill phase. In this work, we leverage LONGGENBENCH, which focuses on long-input and long-output tasks (Liu et al., 2024c), to optimize KV cache compression in scenarios where the output can be as long as 8K tokens.

7 Conclusion

In this paper, we propose SCOPE, a framework that optimizes KV cache usage for long-context generation in LLMs. We observe that excessive compression during the prefill phase harms reasoning capabilities while the deviation of heavy hitters during decoding. To resolve these issues, SCOPE preserves essential KV cache during the prefill phase and employs a sliding strategy to efficiently manage the KV cache generated during decoding. Additionally, we introduce adaptive and discontinuous strategies to further optimize memory usage and transfer. Our extensive experiments demonstrate that SCOPE achieves near-full KV cache performance with only 35% of the original memory while remaining compatible with existing prefill compression methods.

Limitations

SCOPE separates the prefill and decoding phases for long-text generation tasks, while a Top-K algorithm is used to select the *heavy hitters* in both the prefill and decoding phases. We discuss the following limitations:

Prefill Phase We employ the widely recognized top-K algorithm during the prefill phase, and future work could explore chunking or other techniques (Song et al., 2024; Xu et al., 2024) to further enhance the estimation of previous tokens. As discussed in §4.4, the retained KV cache during the prefill phase can be regarded as an "*attention sinks*". Enhancing the quality of this overall "*attention sinks*" is a potential direction for future research. Moreover, our phase-level approach is orthogonal to other KV reuse methods (Xiao et al., 2024a; Lee et al., 2024; Liu et al., 2024a) and could be integrated with these techniques to further optimize memory management and computation efficiency.

Decoding Phase The execution of Top-K at each decoding step is time-costly due to the frequent GPU I/O. Though we optimize the operation frequency in the discontinuous strategy, we can also reduce the I/O size to lower latency. Specifically, by leveraging the PD-separated framework, optimizing I/O for just Φ_d would be more efficient, as the size of Φ_p is constant, while we currently update the entire Φ.

Modality Although SCOPE has shown advantages for long-output tasks in the text modality, there is potential for our method to be applied to long-output tasks in **vision**, such as multi-image generation, where the KV cache required for storing each image is substantial.

Dataset Our experiments demonstrate the effectiveness of SCOPE on **two** well-established benchmarks: LONGGENBENCH and ∞BENCH. In both benchmarks, our strategies consistently outperform the baseline, highlighting the generalization of SCOPE. While these results are robust, we also expect to evaluate SCOPE on more diverse and challenging benchmarks in the future, further validating its scalability and broader applicability.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. arXiv preprint arXiv:2303.08774.

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. Accessed: 2024-07-09.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2024. LongBench: A bilingual, multitask benchmark for long context understanding. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 3119–3137, Bangkok, Thailand. Association for Computational Linguistics.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, et al. 2024. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. arXiv preprint arXiv:2406.02069.

Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, et al. 2024. Magicpig: Lsh sampling for efficient llm generation. arXiv preprint arXiv:2410.16179.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. arXiv preprint arXiv:2110.14168.

Tri Dao. 2024. Flashattention-2: Faster attention with better parallelism and work partitioning. In The Twelfth International Conference on Learning Representations.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. arXiv preprint arXiv:2407.21783.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S Kevin Zhou. 2024. Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference. arXiv preprint arXiv:2407.11550.

Yao Fu. 2024. Challenges in deploying long-context transformers: A theoretical peak performance analysis. arXiv preprint arXiv:2405.08944.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. In International Conference on Learning Representations.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. arXiv preprint arXiv:2310.06825.

Greg Kamradt. 2023. Llms need needle in a haystack: Test-pressure testing llms. Accessed: 2024-11-20.

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 Proceedings of the 29th Symposium on Operating Systems Principles, pages 611–626.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 155–172.

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 The Thirty-eighth Annual Conference on Neural Information Processing Systems.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In Text Summarization Branches Out, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Guangda Liu, Chengwei Li, Jieru Zhao, Chenqi Zhang, and Minyi Guo. 2024a. Clusterkv: Manipulating llm kv cache in semantic space for recallable compression. arXiv preprint arXiv:2412.03213.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024b. Lost in the middle: How language models use long contexts. Transactions of the Association for Computational Linguistics, 12:157–173.

Xiang Liu, Peijie Dong, Xuming Hu, and Xiaowen Chu. 2024c. LongGenBench: Long-context generation benchmark. In Findings of the Association for Computational Linguistics: EMNLP 2024, pages 865–883, Miami, Florida, USA. Association for Computational Linguistics.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024d. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In Forty-first International Conference on Machine Learning.

Shi Luohe, Hongyi Zhang, Yao Yao, Zuchao Li, and hai zhao. 2024. Keep the cost down: A review on methods to optimize LLM's KV-cache consumption. In First Conference on Language Modeling.

Prajwal Singhania, Siddharth Singh, Shwai He, Soheil Feizi, and Abhinav Bhatele. 2024. Loki: Low-rank keys for efficient sparse attention. In The Thirty-eighth Annual Conference on Neural Information Processing Systems.

Woomin Song, Seunghyuk Oh, Sangwoo Mo, Jaehyung Kim, Sukmin Yun, Jung-Woo Ha, and Jinwoo Shin. 2024. Hierarchical context merging: Better long context understanding for pre-trained LLMs. In The Twelfth International Conference on Learning Representations.

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 Forty-first International Conference on Machine Learning.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv preprint arXiv:2403.05530.

Daniel Waddington, Juan Colmenares, Jilong Kuang, and Fengguang Song. 2013. Kv-cache: A scalable high-performance web-object cache for manycore. In 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing, pages 123–130. IEEE.

Zhenglin Wang, Jialong Wu, Yilong Lai, Congzhi Zhang, and Deyu Zhou. 2024. Seed: Accelerating reasoning tree construction via scheduled speculative decoding. arXiv preprint arXiv:2406.18200.

Wenhao Wu, Yizhong Wang, Guangxuan Xiao, Hao Peng, and Yao Fu. 2024. Retrieval head mechanistically explains long-context factuality. arXiv preprint arXiv:2404.15574.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. 2024a. InfLLM: Training-free long-context extrapolation for LLMs with an efficient context memory. In The Thirty-eighth Annual Conference on Neural Information Processing Systems.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024b. Efficient streaming language models with attention sinks. In The Twelfth International Conference on Learning Representations.

Yuhui Xu, Zhanming Jie, Hanze Dong, Lei Wang, Xudong Lu, Aojun Zhou, Amrita Saha, Caiming Xiong, and Doyen Sahoo. 2024. Think: Thinner key cache by query-driven pruning. arXiv preprint arXiv:2407.21018.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024a. Qwen2 technical report. arXiv preprint arXiv:2407.10671.

Dongjie Yang, Xiaodong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. 2024b. PyramidInfer: Pyramid KV cache compression for high-throughput LLM inference. In Findings of the Association for Computational Linguistics: ACL 2024, pages 3258–3270, Bangkok, Thailand. Association for Computational Linguistics.

Jiayi Yuan, Hongyi Liu, Shaochen Zhong, Yu-Neng Chuang, Songchen Li, Guanchu Wang, Duy Le, Hongye Jin, Vipin Chaudhary, Zhaozhuo Xu, Zirui Liu, and Xia Hu. 2024. KV cache compression, but what must we give in return? a comprehensive benchmark of long context capable approaches. In Findings of the Association for Computational Linguistics: EMNLP 2024, pages 4623–4648, Miami, Florida, USA. Association for Computational Linguistics.

Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Hao, Xu Han, Zhen Thai, Shuo Wang, Zhiyuan Liu, and Maosong Sun. 2024. ∞Bench: Extending long context evaluation beyond 100K tokens. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 15262–15277, Bangkok, Thailand. Association for Computational Linguistics.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. Advances in Neural Information Processing Systems, 36:34661–34710.

Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2024. Atom: Low-bit quantization for efficient and accurate llm serving. Proceedings of Machine Learning and Systems, 6:196–209.

Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In International conference on machine learning, pages 12697–12706. PMLR.

Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, et al. 2024. A survey on efficient inference for large language models. arXiv preprint arXiv:2404.14294.

A Discussion

SCOPE is the first framework to compress the KV cache from a phase-level perspective. Unlike token-level eviction methods such as H_2O, SCOPE introduces a phase-aware paradigm, differentiating between the prefill and decoding phases. During prefill, we adopt a token-eviction strategy similar to H_2O, while in the decoding phase, we propose three tailored eviction strategies. This phase-level granularity enables finer control over the KV cache, addressing the limitations of token-level methods and improving efficiency in long-context inference tasks. As discussed in §4.4, our approach is **orthogonal** to existing token-level techniques, offering a new direction for memory optimization in large language models.

SCOPE can be integrated with the KV cache reuse methods seamlessly. While methods like InfLLM (Xiao et al., 2024a) retain all KV data across CPU and GPU, our approach selectively keeps only the most critical KV data on the GPU, improving memory efficiency. This selective retention can be integrated with block-level unit selection of InfLLM, enabling phase-level operations for finer-grained token and unit lookups. Moreover, our strategy is orthogonal to other KV reuse techniques, such as those proposed by Lee et al. (2024) and Liu et al. (2024a). By combining phase-level eviction with these methods, our approach provides a flexible framework for optimizing memory and computation, demonstrating the broader applicability of phase-level strategies in KV compression and reuse.

B Dataset Details

For LongGenBench, We utilize the script[6] from the official repository from the LongGenBench benchmark to construct the version used in our evaluation. The specific setting is provided in Table 4. For ∞Bench, we use the 103 examples of En.Sum from the official repository[7].

[6] https://github.com/Dominic789654/LongGenBench

[7] https://github.com/OpenBMB/InfiniteBench

Type	GSM8K		MMLU		CSQA	
	K	T	K	T	K	T
LongGenBench-4K	30	20	30	53	40	30
LongGenBench-8K	60	10	60	53	80	15

Table 4: Configuration details for the experiment. The table shows the number of questions in one query (K) and the number of iteration times (T).

Prompt Template in LongGenBench

{System Prompt}

Examples:
{CoT Question_1}...{CoT Question_8}
{CoT Answer_1}...{CoT Answer_8}

Following Question:
{CoT Question9}...{CoT Question36}

""

C Experimental Details

C.1 Environment and Evaluation Metrics

Experiments are conducted on NVIDIA A100 (80GB) and RTX 3090 (24GB) GPUs, with integration of Flash Attention 2 (Dao, 2024). The efficiency results, obtained on an RTX-3090 (24GB) with a batch size of 8 using eager attention. For each subtask in LongGenBench, the evaluation metric used is *Accuracy*. The evaluation metric used is *ROUGE-L-Sum* (Lin, 2004).

C.2 Budget Setup

All predictions are generated through greedy decoding for a fair comparison. In the LongGenBench-4K benchmark, we evaluate on the GSM8K+, MMLU+, and CSQA+ datasets. The total cache budget during the prefill phase is set to 2048, which corresponds to approximately 60% of the average input length. During the decoding phase, the total cache budget is set to 2048+512 (decoding compression ratio = 12.5%) and 2048+1024 (decoding compression ratio = 25%). In the LongGenBench-8K benchmark, we evaluate the GSM8K++, MMLU++, and CSQA++ datasets. The total cache budget in the prefill phase is set to 4096 since the number of questions in the multi-QA task doubles. Consequently, the reserved budget is also doubled for simplicity. During the decoding phase, the total cache budget is set to 4096+512 (decoding compression ratio = 6.25%) and 4096+1024

Table 5: Performance comparison for LongGenBench-4K(GSM8k+) and ∞Bench(En.Sum) datasets with total budgets in prefill and decoding.

Method	Performance	Total Budget	
		in Prefill	in Decoding
LongGenBench-4K(GSM8k+)			
StreamingLLM	11.83	2560	2560
H_2O	32.83	2560	2560
SCOPE *(Slide)*	39.00	2048	2048+512=2560
SCOPE *(Adaptive)*	35.33	2048	2048+512=2560
SCOPE *(Discontinuous)*	**39.67**	2048	2048+512=2560
∞Bench(En.Sum)			
StreamingLLM	16.93	2560	2560
H_2O	17.64	2560	2560
SCOPE *(Slide)*	**18.44**	2048	2048+512=2560
SCOPE *(Adaptive)*	18.36	2048	2048+512=2560
SCOPE *(Discontinuous)*	18.17	2048	2048+512=2560

(decoding compression ratio = 12.5%). These settings apply to all experiments presented in Tables 1, 2, and 6. For the En.sum dataset in ∞Bench, due to the large input size (average length > 170K), truncation occurs when using the backbone. This truncation is fair for the input information. In all experiments on this dataset, the settings are as follows: prefill total cache = 2048, and decoding total cache = 2048+512 (decoding compression ratio 50%). This is because the average output length for En.sum slightly exceeds 1K.

To ensure a fair comparison, we conducted preliminary experiments under the setting where methods like H_2O and StreamingLLM utilize the entire cache during the prefill phase. While this setup might initially appear disadvantageous to our proposed SCOPE—since our method intentionally uses less cache by excluding the decoding cache during prefill—our approach still demonstrates superior performance compared to the baselines. As presented in Table 5, the result highlights the effectiveness of our decoding strategies and validates the benefits of separating the prefill and decoding phases, as opposed to employing a unified cache.

C.3 Baselines

We compare the following representative compression methods and full cache to validate the effectiveness of our proposed **SCOPE**.

Unified Compression **StreamingLLM** (Xiao et al., 2024b) keeping the KV of the first few tokens and recent tokens based on the attention sink phenomenon; **H_2O** (Zhang et al., 2023), retains a balance of recent and Heavy Hitter (H_2) tokens based on cumulative attention scores; **PyramidInfer** (Yang et al., 2024b), by leveraging the sparse attention across layers, reduces the cache in the deeper layers, thereby using less budget. §4.4 shows the results of the SCOPE along with these unified compression baselines.

Prefill-Only Compression **SnapKV** (Li et al., 2024), using an observation window to capture attention signals and a pooling strategy to compress KV cache in prefill phase. **PyramidKV** (Cai et al., 2024), is a variant of SnapKV that adjusts the budget allocation across layers. Both methods retain all KV cache generated during the decoding phase. To demonstrate the modularity of SCOPE, we apply it in combination with SnapKV during the decoding phase, as presented in §4.4.

The open-source version of PyramidInfer[8] is not integrated with Flash Attention 2. To ensure a fair comparison with our framework and other baselines, we reproduce its core ideas based on the implementations of H_2O and PyramidKV. During the prefill phase, the retained budget follows the configuration of PyramidKV. During the decoding phase, an additional budget is allocated to maintain the window and recent context, again distributed linearly across layers. Although the budget is allocated linearly across layers, the total budget remains consistent with that of other baselines. In our reproduction of StreamingLLM, we allocated half of the total token budget to the start and the other half to the end, ensuring the preservation of the task instruction and the question.

[8]https://github.com/mutonix/pyramidinfer

Table 6: Performance of our proposed SCOPE using three strategies and baselines on the LONGGENBENCH benchmark with **Mistral-7B-Instruct-v0.3**. The best results among all methods are in **bolded**. The prefill compression ratio averages around 60%.

Method	LONGGENBENCH-4K			LONGGENBENCH-8K			**Avg.**
	GSM8K+	MMLU+	CSQA+	GSM8K++	MMLU++	CSQA++	
Full Cache	16.50	27.49	59.92	12.00	28.67	50.33	32.49
	Decoding Compression Ratio=25.0%			Decoding Compression Ratio=12.5%			
StreamingLLM	12.67	18.30	57.92	11.33	13.5	47.67	26.89
H_2O	7.00	25.44	47.25	6.83	16.20	37.41	23.36
PyramidInfer	6.37	25.35	48.50	6.67	17.52	38.75	23.86
SCOPE *(Slide)*	8.33	17.02	56.67	4.00	17.38	45.25	24.78
SCOPE *(Adaptive)*	14.83	25.38	58.50	7.50	19.29	50.33	29.31
SCOPE *(Discontinuous)*	14.50	27.13	58.50	9.26	19.73	50.33	**29.91**
	Decoding Compression Ratio=12.5%			Decoding Compression Ratio=6.25%			
StreamingLLM	12.67	18.30	57.92	11.33	13.5	47.67	26.89
H_2O	8.00	21.11	41.67	5.17	16.07	34.83	21.14
PyramidInfer	7.75	24.46	41.50	5.67	17.38	44.75	23.59
SCOPE *(Slide)*	8.17	13.51	47.75	4.67	13.98	39.25	21.22
SCOPE *(Adaptive)*	14.50	19.18	58.88	4.17	15.37	50.33	**27.07**
SCOPE *(Discontinuous)*	13.83	18.95	58.88	4.50	15.27	50.33	26.96

Python-style Pseudocode for SCOPE Implement

```python
# Pseudocode for Prefill and Decoding Phases with Three Strategies: Slide, Adaptive, Discontinuous

#
class CachePool:
    def __init__(self):
        self.prefill_cache = (key, value)
        self.decoding_cache = (key, value)

    def total_cache(self):
        return self.prefill_cache + self.decoding_cache

# Prefill phase
def prefill_phase(input_tokens, model, alpha1, alpha2):
    kv_cache = CachePool()
    input_query, input_key, input_value = compute_qkv(input_tokens, model)
    attention_scores = compute_attention(input_query, input_key)
    selected_key, selected_value = select_top_k_cache(attention_scores[:-alpha2], k=alpha1)
    compressed_key = [selected_key, key[-alpha2:]]
    compressed_value = [selected_value, value[-alpha2:]]
    kv_cache.prefill_cache = compressed_key, compressed_value # Update prefill_cache
    return kv_cache

# Decoding phase with SCOPE
def decoding_phase(output_tokens, model, kv_cache, beta1, beta2, strategy):
    for step in range(1, len(output_tokens)):
        token = output_tokens[step]
        current_query, current_key, current_value = compute_qkv(token, model)
        kv_cache.decoding_cache.append(current_key, current_value)
        attention_scores = compute_attention(query, kvcache.total_cache) # Attention in total_cache

        if strategy == "Slide":
            # Retain a sliding window of size decoding_window_len
            if step > max_prompt_len + beta1 + beta2:
                selected_key, selected_value = select_top_k_cache(attention_scores[alpha1+alpha2:-beta2], k=beta1)
                compressed_key = [selected_key, key[-beta2:]]
                compressed_value = [selected_value, value[-beta2:]]
                kv_cache.decoding_cache = compressed_key, compressed_value # Update decoding_cache


        elif strategy == "Adaptive":
            # Dynamically adjust beta1 based on decoding progress
            if step > max_prompt_len + beta2:
                adaptive_beta1 = beta1 * (step - beta2) // (len(output_tokens)-beta2)
                selected_key, selected_value = select_top_k_cache(attention_scores[alpha1+alpha2:-beta2], k=adaptive_beta1) # Use adaptive_beta1
                ... # Update decoding_cache

        elif strategy == "Discontinuous":
            # Jump to noncontinuous
            if step > max_prompt_len + beta2:
                adaptive_beta1 = beta1 * (step-beta2) // (model.max_new_token-beta2)
                jump_interval = (len(output_tokens) - beta2) // beta1  # Interval between jumps
                if step % jump_interval == 0: # Noncontinuous
                    selected_key, selected_value = select_top_k_cache(attention_scores[alpha1+alpha2:-beta2], k=adaptive_beta1) # Use adaptive_beta1
                    ... # Update decoding_cache

    return kv_cache
```

Figure 5: Pseudocode for SCOPE Implement.

Probe Case

```
<<SYS>>
Answer each question step by step, adhering to the format shown in the examples provided. Start each response with 'Answer_' and introduce the final
response with 'The answer is'. Do not repeat the question. Ensure that you respond to all the questions presented, regardless of their number.
<</SYS>>

Examples:
Question_1:
There are 15 trees in the grove. Grove workers will plant trees in the grove today. After they are done, there will be 21 trees. How many trees did the
grove workers plant today?

Question_2:
If there are 3 cars in the parking lot and 2 more cars arrive, how many cars are in the parking lot?

Question_3:
Leah had 32 chocolates and her sister had 42. If they ate 35, how many pieces do they have left in total?

Answer_1:
There are 15 trees originally.
Then there were 21 trees after some more were planted.
So there must have been 21 − 15 = 6.
The answer is 6.

Answer_2:
There are originally 3 cars.
2 more cars arrive.
3 + 2 = 5.
The answer is 5.

Answer_3:
Originally, Leah had 32 chocolates.
Her sister had 42.
So in total they had 32 + 42 = 74.
After eating 35, they had 74 − 35 = 39.
The answer is 39.

Following Question:
Question_4:
Janet's ducks lay 16 eggs per day. She eats three for breakfast every morning and bakes muffins for her friends every day with four. She sells the
remainder at the farmers' market daily for $2 per fresh duck egg. How much in dollars does she make every day at the farmers' market?
Question_5:
A robe takes 2 bolts of blue fiber and half that much white fiber. How many bolts in total does it take?
Question_6:
Josh decides to try flipping a house. He buys a house for $80,000 and then puts in $50,000 in repairs. This increased the value of the house by 150%.
How much profit did he make?
Question_7:
James decides to run 3 sprints 3 times a week. He runs 60 meters each sprint. How many total meters does he run a week?
Question_8:
Every day, Wendi feeds each of her chickens three cups of mixed chicken feed, containing seeds, mealworms and vegetables to help keep them healthy.

 She gives the chickens their feed in three separate meals. In the morning, she gives her flock of chickens 15 cups of feed. In the afternoon, she gives
her chickens another 25 cups of feed. How many cups of feed does she need to give her chickens in the final meal of the day if the size of Wendi's flock
 is 20 chickens?
Question_9:
Kylar went to the store to buy glasses for his new apartment. One glass costs $5, but every second glass costs only 60% of the price. Kylar wants to buy
 16 glasses. How much does he need to pay for them?
Question_10:
Toulouse has twice as many sheep as Charleston. Charleston has 4 times as many sheep as Seattle. How many sheep do Toulouse, Charleston, and
Seattle have together if Seattle has 20 sheep?
Question_11:
Carla is downloading a 200 GB file. Normally she can download 2 GB/minute, but 40% of the way through the download, Windows forces a restart to
install updates, which takes 20 minutes. Then Carla has to restart the download from the beginning. How load does it take to download the file?

"""
```

Figure 6: The probe case used in the pilot observation.