arXiv:2502.01068v4 [cs.LG] 28 Oct 2025

FastKV: Decoupling of Context Reduction and KV Cache Compression for Prefill-Decoding Acceleration

Dongwon Jo[1]*, Jiwon Song[1]*, Yulhwa Kim[2], Jae-Joon Kim[1]
[1]Seoul National University, [2]Sungkyunkwan University
{dongwonjo,jiwon.song,kimjaejoon}@snu.ac.kr
{yulhwakim}@skku.edu

Abstract

While large language models (LLMs) excel at handling long-context sequences, they require substantial prefill computation and key-value (KV) cache, which can heavily burden computational efficiency and memory usage in both prefill and decoding stages. Recent works that compress KV caches with prefill acceleration reduce this cost but inadvertently tie the prefill compute reduction to the decoding KV budget. This coupling arises from overlooking the layer-dependent variation of critical context, often leading to accuracy degradation. To address this issue, we introduce FastKV, a KV cache compression framework designed to reduce latency in both prefill and decoding by leveraging the stabilization of token importance in later layers. FastKV performs full-context computation until a Token-Selective Propagation (TSP) layer, which forwards only the most informative tokens to subsequent layers. From these propagated tokens, FastKV independently selects salient KV entries for caching, thereby decoupling KV budget from the prefill compute reduction based on the TSP decision. This independent control of the TSP rate and KV retention rate enables flexible optimization of efficiency and accuracy. Experimental results show that FastKV achieves speedups of up to 1.82× in prefill and 2.87× in decoding compared to the full-context baseline, while matching the accuracy of the baselines that only accelerate the decoding stage. Our code is available at `https://github.com/dongwonjo/FastKV`.

1 Introduction

Large Language Models (LLMs) have rapidly advanced and now support extended context windows of 128K and even beyond one million tokens (Achiam et al., 2023; Comanici et al., 2025; Anthropic, 2024). This capability significantly enables a broad range of applications for LLMs such as retrieval-augmented generation, multi-document reasoning, and code generation (Yi et al., 2024; Laban et al., 2023; Rando et al., 2025). However, the computational and memory overhead of long-context inference remains a critical bottleneck.

*Equal Contribution

Method	Prefill	Decoding	Acc.
Full-context	Slow	Slow	High
StreamingLLM	Slow	Fast	Low
SnapKV	Slow	Fast	High
GemFilter	Fast	Fast	Low
FastKV	**Fast**	**Fast**	**High**

Table 1: Comparison of KV cache compression methods. FastKV uniquely achieves fast prefill and decoding, and high accuracy simultaneously.

Long-context inference introduces substantial burdens in both the prefill and decoding stages. In the prefill stage, attention computation scales quadratically with input length, making very long prompts expensive to process. In the decoding stage, the large amount of KV cache becomes the dominant factor, consuming GPU memory and reducing throughput since every generated token must repeatedly access this cache. Together, these trends make inference prohibitively costly: prefill slows down with longer inputs, while decoding efficiency deteriorates as the cache size grows with context length.

Recent studies have explored two complementary directions to overcome these burdens. Most existing KV cache compression methods, such as StreamingLLM (Xiao et al., 2023) and SnapKV (Li et al., 2024) target the decoding stage, by pruning already-generated KV cache, but do not accelerate the prefill stage at all.

In contrast, GemFilter (Shi et al., 2024) and PyramidInfer (Yang et al., 2024a) focus on the prefill stage, aiming to reduce the quadratic cost of processing long prompts by generating KV cache of only critical tokens. Despite these advances, a fundamental trade-off remains: decodingfocused approaches fail to alleviate the prefill burden, whereas prefill-focused methods compromise accuracy when the KV budget is reduced to levels that would significantly accelerate decoding.

To bridge this gap, we propose FastKV, a KV cache compression framework that accelerates prefill and decoding stages without compromising accuracy. FastKV is motivated by two key observations: (i) during prefill, early layers must propagate the full-context so that later layers retain the opportunity to attend to any part of the context; (ii) however, during decoding, each layer ultimately attends to only a small fraction of the prefilled tokens, meaning that it is unnecessary to retain the entire KV cache built during prefill.

Building on these insights, FastKV introduces two techniques. First, we adopt a two-stage prefill strategy that retains the full-context in early layers while context usage remains unstable, and, once stabilization becomes evident, switches to propagating only salient tokens in later layers. Second, we decouple the KV budget to separate the context processed during prefill from the amount of KV cache retained for decoding.

As summarized in Table 1, these techniques address the prefill–decoding trade-off by performing context reduction at the right time and preserving only critical KV caches for decoding, achieving up to 1.82× faster prefill and up to 2.87× faster decoding compared to the full-context baseline, while maintaining accuracy drop within 1% on LongBench benchmark.

2 Background

2.1 Bottlenecks of Long-context Inference

Auto-regressive LLM inference consists of two stages: prefill and decoding stage.

- In the prefill stage, the model processes the entire input prompt and builds the KV cache across all layers. The computational cost of attention in this stage scales quadratically with the input length.
- In the decoding stage, the model generates tokens auto-regressively while reusing the KV cache. Here, the per-step attention cost is only linear in the number of cached tokens, but the cache itself grows linearly with input length and must be repeatedly accessed at every step.

When the context is short, these costs are manageable. However, as context length increases, both stages become severe bottlenecks: Prefill latency explodes as the context length increases due to quadratic attention, and the decoding latency deteriorates as the linearly growing KV cache must be repeatedly accessed at every step, creating significant memory bandwidth overhead.

2.2 Prior Approaches and Limitations

A large body of work has explored KV cache compression to alleviate the burden of long-context inference. Early efforts such as StreamingLLM (Xiao et al., 2023) exploit the observation of attention sink tokens, retaining only those sink tokens together with the most recent context in the KV cache. Building on this idea, SnapKV and H2O (Li et al., 2024; Zhang et al., 2023) introduced attention-based importance metrics to retain only salient tokens, thereby reducing the KV cache more selectively. Subsequent studies (Feng et al., 2024; Fu et al., 2024; Cai et al., 2024) further refined this direction by assigning fine-grained KV budgets at the head or layer level, aiming to preserve accuracy. However, these methods provide at best marginal accuracy improvements over SnapKV, while leaving the fundamental bottleneck unsolved: they still require producing the KV cache for the full-context before selecting which tokens to retain, so prefill latency remains unreduced.

In contrast, prefill-aware KV cache compression methods have emerged. Instead of processing all tokens during prefill, they attempt to accelerate inference by reducing the effective context length up front. For example, GemFilter (Shi et al., 2024) leverages pre-defined filter layer's attention to select a compact subset of input tokens, while PyramidInfer (Yang et al., 2024a) exploits cross-layer redundancy to gradually reduce the hidden states propagated to subsequent layers. These approaches naturally produce a smaller KV cache, yielding both prefill and decoding speedups. Nevertheless, they suffer from inherent trade-offs: GemFilter enforces the same reduced set of tokens across all subsequent layers, discarding potentially useful information for deeper processing, while PyramidInfer, though layer-aware, still prunes aggressively from early layers, both of which compromise accuracy. Moreover, in these designs, the KV cache compression is tightly coupled with the amount of prefill compute reduction: achieving sufficient decoding acceleration requires aggressive context pruning, which simultaneously amplifies accuracy degradation. Importantly, this fragility is not due

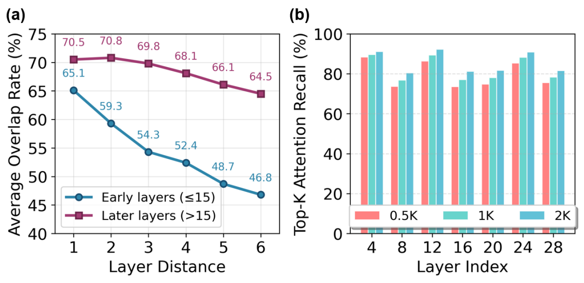

Figure 1: (a) Early layers exhibit unstable context focus, reflected by low critical token overlap. (b) Attention distributions are sparse, with Top-K tokens dominating the scores.

to a tighter memory budget; prefill-aware schemes drop tokens on the fly, blocking later layers from attending to them and thus degrading accuracy. The drawbacks of existing prefill-aware KV cache compressions are further discussed in Appendix B.2.

These limitations suggest that existing methods are inherently constrained by conflating KV cache compression with prefill compute reduction into a single stage, and a more balanced approach is needed to jointly optimize prefill and decoding efficiency.

3 Motivations

3.1 Layer-dependent Context Dynamics

A key challenge in reducing prefill latency is to determine when to shrink the amount of context processed at each layer. Existing approaches present two contrasting strategies: GemFilter processes the context up to the filter layer to select salient tokens and restart the prefill stage with the selected tokens, while PyramidInfer gradually reduces the context size layer by layer.

In both methods, even early layers lose the opportunity to access full-context of the tokens. They do not directly examine how the selection of critical tokens evolves across layers, and to what extent early pruning may disrupt later layers' ability to attend to their eventual targets. To answer this question, we analyze how the layer-wise critical tokens changes as depth increases.

We feed a 128K-token input into LLaMA-3.1-8B-Instruct and, at each layer, collect the top-512 critical tokens that receive the highest average attention mass across heads. We then calculate the average overlap ratio of these critical token indices between layers as the layer distance increases. Figure 1(a) presents the results. In the early layers ($\leq$15), the overlap ratio drops sharply with increasing layer distance, indicating that the critical tokens perceived by each layer shift sharply. In contrast, in the later layers ($>$15), the overlap decays much more slowly, suggesting that the same subset of tokens remains consistently important across multiple successive layers.

These observations highlight the difference in context utilization across layers. In the early layers, attention focus is highly unstable, and pruning tokens at this stage irreversibly remove tokens that later layers would otherwise consider critical. Once discarded, these tokens cannot be recovered, causing downstream layers to lose access to potentially indispensable context and leading to severe performance degradation. In contrast, in the later layers, the set of critical tokens shows a high degree of overlap across layers, so aggressive token pruning can be applied with minimal impact on the model accuracy. This layer-dependent context dynamic implies that token pruning during the prefill stage must process the full-context in early layers, and then transition to selective context propagation in later layers.

3.2 Sparse Context Utilization Across Layers

The results of Section 3.1 suggest that early layers must be allowed to process the full-context during prefill so that later layers do not lose access to potentially critical tokens. However, this does not imply that each layer has to cache all of the KV values that it generated. To investigate what fraction of the context is actually used during decoding, we measure the top-K attention recall: the proportion of total attention mass covered by the K most attended tokens at each layer.

As shown in Figure 1(b), across all layers of LLaMA-3.1-8B-Instruct with 128K input tokens, only a sparse subset of tokens dominates the attention distribution. Even with K = 512 (0.38% of entire tokens), the majority of the attention mass is already captured. This indicates that all layers only rely on a small subset of tokens once decoding begins.

This phenomenon of sparse utilization is consistent with results from prior works on KV cache compression during decoding stage, such as SnapKV and H2O. These studies have demonstrated that KV cache compression can be highly effective because only a small fraction of tokens are actively used during decoding.

Overall, our analysis suggests that the early layers require careful management of KV values. In the prefill stage, the full-context must be processed

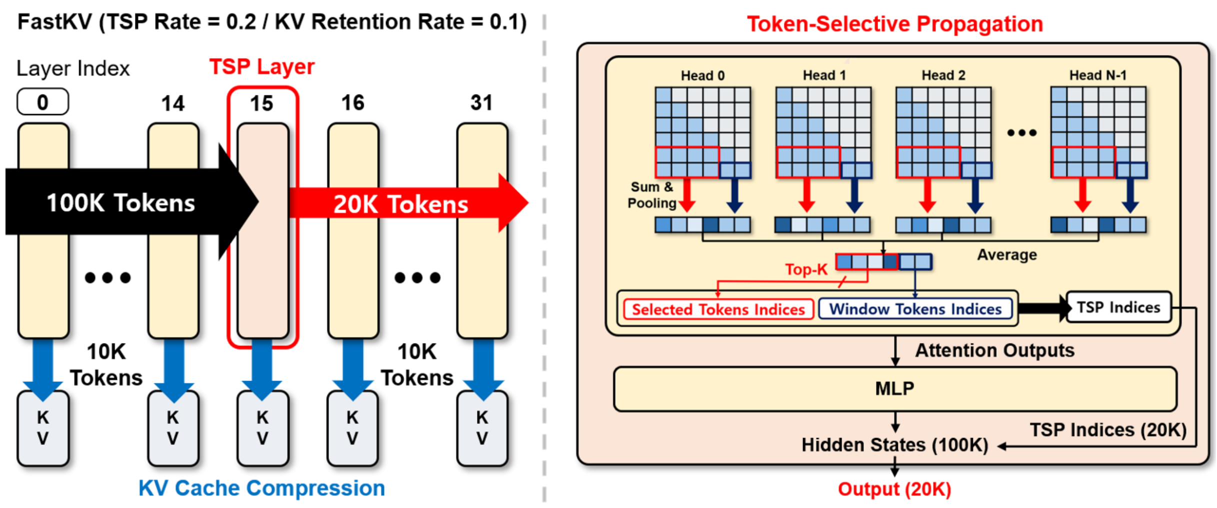

Figure 2: Illustration of the proposed FastKV scheme. The proposed FastKV introduce Token-Selective Propagation approach to selectively propagate only a limited set of tokens while effectively compressing KV cache.

to provide subsequent layers with complete contextual information. However, the model can cache only a small subset of generated tokens, as the decoding stage depends on only a small fraction of the prefill context.

4 Proposed FastKV

4.1 Overview of FastKV

The overall workflow of FastKV is illustrated in Figure 2. FastKV accelerates long-context inference by rethinking how much context is propagated during prefill and how much KV cache is retained during decoding. FastKV introduces two complementary innovations:

- **Token-Selective Propagation (TSP)**. A dedicated TSP layer, placed around the middle of the decoder, forwards only a selected subset of hidden states to later layers rather than propagating the entire prompt. This reduces the context passed downstream while keeping critical information intact.

- **Layer-wise KV retention**. During prefill, each layer independently discards less influential entries and preserves only a specified retention rate of its KV cache. After prefill, every layer thus maintains a compressed KV cache, which significantly accelerates decoding without degrading accuracy.

By allowing early layers to process the full-context before compression, FastKV ensures that they can freely identify the tokens to retain, while later layers, which tend to converge on similar subsets, remain robust even when operating on reduced context. This design allows every layer to carry a compressed but meaningful KV cache into decoding.

Compared to prior work, FastKV avoids the rigid constraints of GemFilter and PyramidInfer. GemFilter enforces a single layer's token selection across all layers, which is particularly harmful to early layers where each layer attends to different subsets of tokens. PyramidInfer reduces context from the very first layers, limiting the flexibility of each layer to select its own important tokens. Furthermore, both methods couple prefill compression with the decoding KV budget, whereas FastKV decouples them, enabling independent control of how much context is propagated during prefill and how much KV is preserved for decoding. This flexibility yields a superior accuracy–efficiency trade-off.

4.2 Two-stage Prefill with Token-Selective Propagation

As shown in Section 3.1, the set of critical tokens fluctuates substantially in the early layers but stabilizes in later layers. This observation motivates a two-stage prefill strategy: using full-context computation in the early layers to capture diverse token dependencies, and then reducing the context in subsequent layers once token importance has stabilized.

To implement this, FastKV introduces a dedicated TSP layer. In this layer, each token is evaluated by how strongly it is attended by the recent window tokens, which serve as queries. Specifically, for each token i, its saliency score S is computed by averaging attention weights over all heads when queried from the window tokens:

$$S_i^{l,h} = \text{Pooling}(\sum_{n=0}^{N_{\text{obs}}} Att_l[h,\, N_I - n,\, i+m]) \quad (1)$$

$$S_i^{TSPlayer} = \frac{1}{H}\sum_{h=0}^{H-1} S_i^{TSPlayer,h}, \quad (2)$$

Here, $S_i^{l,h}$ is the saliency score of i-th token in h-th attention head of the l-th layer. Att_l denotes the attention score matrix of l-th layer, while N_I and N_{obs} indicate the number of tokens in the input prompt and the window token size, respectively. H denotes the number of attention heads in each layer. Since compressing the layer output requires a single index set, we average the attention weights to calculate a score $S_i^{TSPlayer}$ that represents the saliency of tokens at the layer level, as outlined in Equation 2.

Based on these saliency scores, indices of the top-ranked tokens up to the predefined TSP rate are selected. Crucially, all window tokens themselves are always included in the propagated set, and their indices are merged with the saliency-selected ones to form the final TSP token set passed to the next layer.

Prefill therefore proceeds in two distinct stages. In the first stage, from the input up to the TSP layer, all layers process the full-context and construct their compressed KV caches. In the second stage, starting at the TSP layer, only the reduced subset of saliency-selected tokens (together with all window tokens) continues forward, and later layers process this compressed context to form their compressed KV caches.

By applying Token-Selective Propagation only after stabilization, FastKV preserves the heterogeneous attention patterns of early layers while still reducing prefill latency in later layers.

4.3 Impact of TSP and TSP Layer Selection

We first investigate the effect of applying Token-Selective Propagation (TSP) at different layers. For each candidate TSP layer, we compute the final logits and compare them against those obtained from the full-context baseline. Figure 3 reports the normalized L2 distance between the two outputs for LLaMA-3.1-8B-Instruct.

We also include a GemFilter-like strategy as a baseline, where a single filter layer selects a subset of tokens and then the model is re-prefilled using only this subset. This procedure forces even the early layers, which normally attend to distinct tokens, to process the same limited token set, resulting in significant deviations in the final logits.

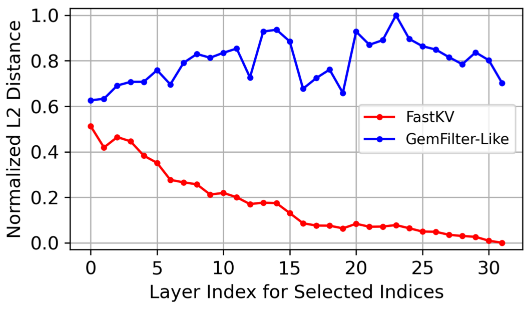

Figure 3: Comparison of normalized L2 distances between hidden states generated by the full-context baseline, TSP, and GemFilter-like methods.

In contrast, TSP preserves full-context processing before the TSP layer, so each early layer can still attend to its own preferred subset of tokens. As a result, when the TSP layer is placed at the middle or later part of the model, the logits remain much closer to the full-context baseline than those produced by GemFilter, as shown in Figure 3.

If the TSP layer is placed too early, token selection becomes overly restricted and the resulting logits deviate substantially from the full-context baseline. If it is placed too late, many layers still process the full-context and prefill latency reduction is limited. Therefore, selecting the TSP layer at the right position is critical, which we formalize as follows:

$$L_{TSP} = \underset{L \le L_{\max}}{\text{argmin}}\ \frac{1}{N}\sum_{i=1}^{N} \left\| \mathbf{H}_i - \mathbf{H}'_{L,i} \right\|_2^2, \quad (3)$$

where $\mathbf{H}_i$ denotes the hidden state at the final layer under full-context for the i-th calibration input, and $\mathbf{H}'_{L,i}$ the corresponding hidden state when TSP is applied at candidate layer L.

The criterion searches for the earliest layer whose output remains close to the full-context baseline, while constraining $L \le L_{\max}$ to avoid excessive late placement. This ensures that the chosen TSP layer simultaneously minimizes model degradation and provides tangible prefill latency savings.

4.4 KV Retention Decoupled with Prefill Compute

In prefill-aware methods such as GemFilter and PyramidInfer, the amount of context reduced during prefill directly determines the KV cache size available for decoding. This tight coupling means that reducing the KV budget inevitably requires more aggressive prefill compute reduction. As a result, early layers are deprived of tokens they uniquely rely on, leading to significant accuracy degradation.

FastKV breaks this coupling by introducing two independent hyperparameters: the TSP rate, which controls how much context is propagated after the TSP layer during prefill, and the KV retention rate, which specifies how many of each layer's KV entries are preserved for reuse during decoding.

By decoupling prefill compute reduction from decoding KV retention, FastKV allows latency savings in prefill and memory savings in decoding to be tuned separately. For example, one can adopt a conservative TSP rate to protect accuracy in prefill, while still using a tighter KV retention rate to reduce KV size during decoding. Conversely, when memory constraints are critical, both rates can be reduced together to maximize efficiency. This independent control enables FastKV to achieve better accuracy–efficiency trade-offs than GemFilter and PyramidInfer, which couple the two stages and thus sacrifice accuracy when aiming for stronger KV cache compression.

5 Experiments

5.1 Setup

Models and Datasets. We evaluate two open-source LLMs of different sizes: LLaMA-3.1-8B-Instruct (Dubey et al., 2024) and Ministral-8B-Instruct (Mistral AI Team, 2025). These models have 32 and 36 decoder layers, respectively, and employ GQA (Ainslie et al., 2023) with a context window size of 128K tokens. We use LongBench (Bai et al., 2023), which consists of 16 subtasks across single-document QA, multi-document QA, summarization, few-shot learning, synthetic tasks, and code completion to assess the models' long-context understanding capabilities. For further examination, LLaMA-3.1-8B-Instruct is also evaluated on Needle-in-a-Haystack (Kamradt, 2023) and RULER (Hsieh et al., 2024). See Appendix C for more details on evaluation.

Implementation Details. We integrate our proposed FastKV method upon self-attention implementation of HuggingFace Transformers library, which utilizes FlashAttention-2 (Dao, 2023) kernel. We select layer 15 as the TSP layer for LLaMA-3.1-8B-Instruct and layer 17 for Ministral-8B-Instruct. We fix the observation window size to 8 and the pooling kernel size to 7. The TSP rate is set to 20% for evaluation. The setting yields 60% prefill compute rate for both models.

Baselines. We compare FastKV against five baseline methods for KV cache compression: StreamingLLM (Xiao et al., 2023), H2O (Zhang et al., 2023) and SnapKV (Li et al., 2024)(decoding-only acceleration); PyramidInfer (Yang et al., 2024a) and GemFilter (Shi et al., 2024) (prefill-aware acceleration). KV retention rate is fixed to 10% and 20% for all methods except PyramidInfer. The indices of GemFilter filter layers for LLaMA-3.1-8B-Instruct and Ministral-8B-Instruct are 13 and 17, respectively. For GemFilter, this yields 51% and 60% of prefill compute rate for LLaMA. This setting yields 60% and 70% of prefill compute rate for Ministral. Setting prefill compute rate 60% for PyramidInfer automatically determines its KV retention rate to be 60%, which is equal to its prefill compute rate.

5.2 Accuracy

LongBench. The accuracy evaluation results on LongBench are summarized in Table 2. The full breakdown of results is provided in Appendix D.1. Previous works for decoding-only acceleration, such as StreamingLLM and H2O, show significant accuracy degradation, whereas SnapKV maintains accuracy after KV cache compression, with average drops below 1.46% and 0.76% at 10% and 20% KV retention, respectively. GemFilter recomputes the prefill stage on a fragmented input consisting only of selected tokens, thereby discarding information carried by the removed tokens and incurring substantially larger accuracy drop of up to 11.58%. PyramidInfer, even under a higher KV retention rate, begins prefill compute reduction from early layers where context usage is still unstable and stores the resulting compressed KV, which propagates information loss and yields inferior accuracy. By contrast, FastKV attains memory and computational efficiency comparable to GemFilter while preserving accuracy on par with the full-context baseline. This demonstrates the effectiveness of FastKV's two-stage prefill strategy retaining the full-context in early layers and propagating only salient tokens in later layers.

RULER. We present the RULER evaluation results for LLaMA-3.1-8B-Instruct with a KV retention rate of 10% in Table 3. The evaluation is conducted with input context lengths up to 128K tokens. Notably, FastKV outperforms other methods even under 128K inputs. Taken together with the Long-

Table 2: LongBench results on LLaMA-3.1-8B-Instruct and Ministral-8B-Instruct.

Method	Prefill	KV	Single-Doc QA	Multi-Doc QA	Summarization	Few-shot	Synthetic	Code	Avg.
LLaMA-3.1-8B-Instruct									
Full-context	100%	100%	43.58	44.65	29.22	69.48	54.21	60.01	50.19
Decoding-Only									
StreamingLLM	100%	10%	28.67	37.01	22.28	63.19	47.96	58.01	42.85
	100%	20%	30.43	38.89	24.19	65.41	46.83	59.44	44.20
H2O	100%	10%	27.12	19.74	25.96	66.65	34.58	54.88	38.15
	100%	20%	30.66	19.69	26.05	68.00	36.28	58.05	39.79
SnapKV	100%	10%	41.90	44.11	25.34	68.17	53.72	59.53	48.73
	100%	20%	43.27	43.92	26.75	68.21	53.73	60.73	49.43
Prefill-Aware									
PyramidInfer	60%	60%	28.41	19.31	23.50	66.40	36.87	61.41	39.32
GemFilter	51%	10%	28.39	40.33	21.78	64.31	46.13	30.75	38.61
	61%	20%	34.63	40.77	24.39	66.39	51.59	36.56	42.39
FastKV	60%	10%	41.95	43.73	24.78	69.45	53.43	57.45	48.47
	60%	20%	42.93	43.74	26.03	69.51	53.56	68.64	49.07
Ministral-8B-Instruct									
Full-context	100%	100%	41.58	49.45	27.72	70.83	54.50	67.13	51.87
Decoding-Only									
StreamingLLM	100%	10%	27.74	35.87	21.00	65.44	34.50	62.70	41.21
	100%	20%	29.14	38.89	23.23	68.01	38.50	–	–
H2O	100%	10%	37.47	43.88	25.96	68.64	38.75	60.95	45.94
	100%	20%	38.93	44.27	26.38	69.30	39.25	63.58	46.95
SnapKV	100%	10%	39.97	49.43	23.92	70.03	55.00	65.75	50.68
	100%	20%	40.63	49.59	25.67	70.64	54.50	67.07	51.35
Prefill-Aware									
PyramidInfer	60%	60%	34.29	37.34	24.16	68.19	52.25	65.83	47.01
GemFilter	60%	10%	37.92	55.22	23.30	65.05	46.00	30.88	42.82
	70%	20%	40.84	54.06	25.31	68.20	51.25	38.56	46.37
FastKV	60%	10%	39.71	49.63	23.38	70.71	55.00	64.00	50.40
	60%	20%	41.08	49.20	25.02	71.41	55.00	64.68	51.07

Table 3: RULER results on LLaMA-3.1-8B-Instruct.

Method	**Prefill**	**KV**	**8K**	**16K**	**32K**	**64K**	**128K**	**Avg.**
Full-context	100%	100%	90.1	95.0	83.4	85.5	76.3	86.0
StreamingLLM	100%	10%	15.0	21.5	24.4	16.9	15.0	18.6
H2O	100%	10%	27.1	-	-	-	-	-
SnapKV	100%	10%	75.6	76.8	72.9	75.0	67.7	73.6
PyramidInfer	60%	60%	66.5	-	-	-	-	-
GemFilter	51%	10%	69.7	68.2	70.4	69.8	69.8	69.6
FastKV	60%	10%	77.8	77.3	77.2	77.4	68.2	75.6

Table 4: Needle-in-a-Haystack results on LLaMA-3.1-8B-Instruct.

Method	**Prefill**	**KV**	**Score**
Full-context	100%	100%	99.0
StreamingLLM	100%	10%	33.5
SnapKV	100%	10%	99.0
GemFilter	51%	10%	95.8
FastKV	60%	10%	99.9

Bench evaluation, these results demonstrate that FastKV effectively offers the best accuracy–latency trade-off across diverse long-context tasks while achieving efficient KV cache compression and acceleration for prefill and decoding stages.

Needle-in-a-Haystack. Table 4 shows the Needle-in-a-Haystack evaluations for LLaMA-3.1-8B-Instruct with a KV retention rate of 10% budget. Score denotes the average accuracy across sequence lengths 16K, 32K, 48K, 64K, 80K, 96K, 112K, and 128K. Results for each sequence length are provided in the Appendix D.2. Consistent with the results observed in LongBench and RULER benchmark, FastKV achieves the best performance.

5.3 Latency

We benchmark end-to-end latency using LLaMA-3.1-8B-Instruct on a single A100 SXM GPU. Results for Ministral-8B-Instruct are reported in the Appendix D. All experiments fix the generation to 256 tokens, while the input context length varies.

As shown in Figure 4, end-to-end latency increases rapidly with longer input context under full-context execution, where prefill latency dominates at 128K. Decoding-only approaches such as StreamingLLM and SnapKV reduce decoding time, but their benefit diminishes when prefill dominates; in particular, SnapKV stores KV states per attention head, which limits decoding speedup in GQA models. H2O, which requires exporting the entire at-

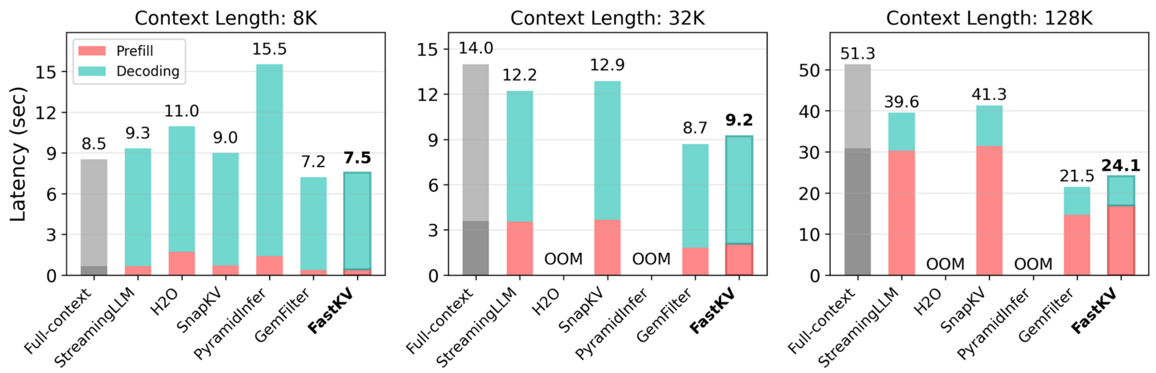

Figure 4: End-to-end inference latency breakdown of LLaMA-3.1-8B-Instruct at varying input context lengths (generating 256 tokens).

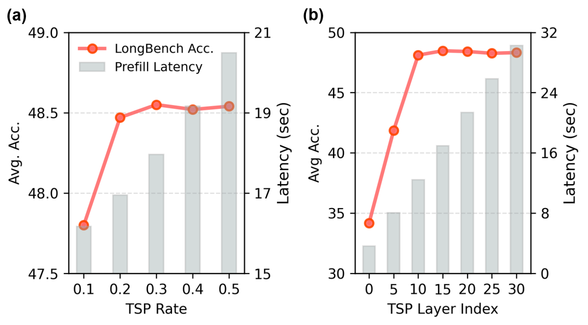

Figure 5: (a) Effect of TSP rate on LongBench average accuracy and prefill latency. (b) Effect of TSP layer index on LongBench average accuracy and prefill latency.

tention map, cannot leverage FlashAttention-2 and is slower than full-context even at 8K, while running out of memory at longer contexts. Similarly, PyramidInfer does not employ FlashAttention-2 in its implementation and therefore causes an out-of-memory error at context length beyond 8K. Even at 8K context length, it shows substantially long decoding latency suffering from large KV cache size due to its enforced high KV retention rate.

In contrast, both GemFilter and FastKV effectively reduce both prefill and decoding latency. Their advantage is most pronounced at 128K, where they achieve more than 2× speedup over the full-context baseline. GemFilter's slight advantage is primarily attributed to its low prefill compute rate, determined by the filter layer selection (13), compared to the TSP layer (15). However, as discussed in Section 5.2, GemFilter is more prone to accuracy degradation.

5.4 Ablation Studies

We conduct ablation studies on LLaMA-3.1-8B-Instruct to examine the effect of the two key hyperparameters of FastKV: TSP rate and TSP layer index.

TSP Rate. Figure 5(a) shows the effect of varying the TSP rate while fixing the TSP layer index and retention rate at 15 and 10% respectively. When the rate is set too low, only a very small subset of tokens is propagated to later layers, which limits the model's ability to convey sufficient context and leads to accuracy degradation. Increasing the rate improves accuracy by allowing more salient tokens to be propagated, and performance stabilizes around 20%, which we adopt as the default setting.

TSP Layer Index. Figure 5(b) reports the impact of the TSP layer index under a fixed TSP rate of 20% and retention rate of 10%. Applying TSP at earlier layers leads to substantial accuracy degradation. This is because early layers exhibit highly dynamic attention patterns, where the set of critical tokens frequently shifts and discarding tokens too early removes information that later layers would still require. By contrast, later layers show more stable token dependencies, making selective propagation more reliable. The proposed selection algorithm identifies layer 15 as an optimal point where the LongBench score saturates, indicating a favorable trade-off between speed and accuracy.

6 Conclusion

In this work, we introduced FastKV, a framework that accelerates long-context inference through a two-stage prefill strategy incorporating Token-Selective Propagation and a KV cache compression that allocates KV retention rate independently of the prefill compute rate. Experiments show that FastKV achieves up to 1.82× faster prefill and 2.87× faster decoding while maintaining the accuracy of full-context baselines. Beyond raw speedup, FastKV demonstrates that accounting for the dynamics of critical context across layers is crucial for efficient long-context processing. This design principle enables scalable inference as context lengths continue to grow, making long-context inference more practical.

7 Limitations

While FastKV yields substantial acceleration in both prefill and decoding stages, its advantage becomes more evident in long-context scenarios where KV cache handling dominates the overall latency. For short-context inputs, however, the KV cache is not a major overhead, and alternative acceleration methods such as quantization (Lin et al., 2024; Frantar et al., 2023; Ashkboos et al., 2024) or optimized kernels (Shah et al., 2024; Ye et al., 2025) may be more beneficial. Combining FastKV with these complementary techniques could enable more consistent efficiency across diverse inference settings. Beyond these research directions, FastKV is readily compatible with modern serving frameworks (Kwon et al., 2023; Zheng et al., 2024). It is orthogonal to batching and paged attention. These directions highlight promising opportunities for extending FastKV toward more adaptive and universally efficient LLM inference.

8 Ethical Considerations

This work focuses on improving inference efficiency of large language models. FastKV reduces computational and memory overhead with minimal impact on output behavior. The method operates at the architectural level and does not introduce new ethical or societal risks beyond those already associated with large language models.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*.

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. 2024. Quarot: Outlier-free 4-bit inference in rotated llms. *Advances in Neural Information Processing Systems (NeurIPS)*, 37:100213–100240.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, and 1 others. 2023. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, and Wen Xiao. 2024. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *CoRR*, abs/2406.02069.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *International Conference on Learning Representations (ICLR)*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 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*.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. Gptq: Accurate post-training quantization for generative pre-trained transformers. *International Conference on Learning Representations (ICLR)*.

Yu Fu, Zefan Cai, Abedelkadir Asi, Wayne Xiong, Yue Dong, and Wen Xiao. 2024. Not all heads matter: A head-level kv cache compression method with integrated retrieval and reasoning. *arXiv preprint arXiv:2410.19258*.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems (NeurIPS)*, 37:1270–1303.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *Preprint*, arXiv:2407.02490.

Greg Kamradt. 2023. Needle in a haystack - pressure testing llms. https://github.com/gkamradt/LLMTest_NeedleInAHaystack.

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.

Philippe Laban, Wojciech Kryscinski, Divyansh Agarwal, A. R. Fabbri, Caiming Xiong, Shafiq R. Joty, and Chien-Sheng Wu. 2023. Summedits: Measuring llm ability at factual reasoning through the lens of summarization. In *Conference on Empirical Methods in Natural Language Processing*.

Xunhao Lai, Jianqiao Lu, Yao Luo, Yiyuan Ma, and Xun Zhou. 2025. Flexprefill: A context-aware sparse attention mechanism for efficient long-sequence inference. In *The Thirteenth International Conference on Learning Representations*.

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. *Advances in Neural Information Processing Systems (NeurIPS)*.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of machine learning and systems*, 6:87–100.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *International Conference on Machine Learning (ICML)*.

Mistral AI Team. 2025. Mistral-nemo. https://mistral.ai/news/ministraux.

Dan Peng, Zhihui Fu, Zewen Ye, Zhuoran Song, and Jun Wang. 2025. Accelerating prefilling for long-context llms via sparse pattern sharing. *arXiv preprint arXiv:2505.19578*.

Stefano Rando, Luca Romani, Alessio Sampieri, Luca Franco, John Yang, Yuta Kyuragi, Fabio Galasso, and Tatsunori Hashimoto. 2025. Longcodebench: Evaluating coding llms at 1m context windows. *arXiv preprint arXiv:2505.07897*.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems (NeurIPS)*, 37:68658–68685.

Zhenmei Shi, Yifei Ming, Xuan-Phi Nguyen, Yingyu Liang, and Shafiq Joty. 2024. Discovering the gems in early layers: Accelerating long-context llms with 1000x input token reduction. *arXiv preprint arXiv:2409.17422*.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*.

Dongjie Yang, Xiaodong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. 2024a. 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.

June Yong Yang, Byeongwook Kim, Jeongin Bae, Beomseok Kwon, Gunho Park, Eunho Yang, Se Jung Kwon, and Dongsoo Lee. 2024b. No token left behind: Reliable kv cache compression via importance-aware mixed precision quantization. *arXiv preprint arXiv:2402.18096*.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and 1 others. 2025. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*.

Zihao Yi, Jiarui Ouyang, Zhe Xu, Yuwen Liu, Tianhao Liao, Haohao Luo, and Ying Shen. 2024. A survey on recent advances in llm-based multi-turn dialogue systems. *arXiv preprint arXiv:2402.18013*.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, and 1 others. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems (NeurIPS)*.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, and 1 others. 2024. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems (NeurIPS)*, 37:62557–62583.

A Related Works

Research efforts to alleviate the burden of long-context inference in LLMs have explored multiple directions. One body of work is sparse attention (Jiang et al., 2024; Lai et al., 2025; Peng et al., 2025) which reduces the amount of computation required for generating the KV cache. They focus on identifying inherent sparse patterns in attention map and leverage them to design optimized kernels.

Methods that compress the generated KV cache are typically categorized into quantization-based and pruning-based approaches. Quantization-based methods, such as KVQuant (Hooper et al., 2024), KIVI (Liu et al., 2024) and MiKV (Yang et al., 2024b), focus on reducing the precision of key and value tensors to lower memory and bandwidth requirements without altering the cache structure itself. While these techniques effectively reduce storage and transfer costs, they still do not alleviate the quadratic computation in the prefill stage, limiting their impact on overall latency. Pruning-based methods (Xiao et al., 2023; Zhang et al., 2023; Li et al., 2024) include the baselines of this work, dropping KV cache in token-wise manner. These approaches directly bring speed up in decoding stage without complex kernel-level optimizations since overhead of loading KV cache and computation itself is reduced. Some works (Yang et al., 2024a; Shi et al., 2024) aim to reduce the number of hidden states processed during the prefill stage, thereby generating smaller KV cache and achieving acceleration in both prefill and decoding stages.

FastKV approaches the problem from the perspective of KV cache pruning, aiming to achieve prefill compute reduction as well. It is orthogonal to kernel-level sparse attention and KV cache quantization techniques, and thus can be combined with them to further extend efficiency.

B Additional Methodological Details

B.1 FastKV Algorithm

We present the pseudocode for the FastKV algorithm, which performs two complementary compression strategies during the prefill stage: (1) reducing hidden states via Token-Selective Propagation (TSP), and (2) compressing the attention key-value (KV) cache at each layer. These two steps are carried out independently: TSP selects a subset of hidden states to propagate to later layers, while KV cache compression prunes KV cache entries at each layer using attention-based importance scores. Together, they improve prefill efficiency and reduce memory usage without modifying the model architecture or introducing any runtime overhead during decoding.

The key terms used in the pseudocode are:

- $\boldsymbol{L_{TSP}}$: Index of TSP layer
- $\boldsymbol{R_{TSP}}$: TSP rate
- $\boldsymbol{R_{KV}}$: KV retention rate
- ***KVCompress***: Selects the KV entries of top-{context length $\times$ $\boldsymbol{R_{KV}}$} critical tokens based on group-wise saliency scores. The group-wise scores are obtained by averaging head-wise saliency values computed using Equation 1 within each key-value group.
- ***HiddenCompress***: Selects hidden states of top-{context length $\times$ $\boldsymbol{R_{TSP}}$} critical tokens, as determined by Equation 2, to propagate to next layer.

Algorithm 1 FastKV algorithm for the KV cache compression during the prefill stage

```
Require: $input\ sequence\ \{I\}, \#layers\ \{L\}, L_{TSP}, R_{TSP}, R_{KV}$
Ensure: $generated\ token\ \{O\}, KV\ Cache\ \{C\}$
 1: $X \leftarrow Embedding(I)$
 2: for $l$ = 0 to $L - 1$ do
 3:     if $l \leq L_{TSP}$ then
 4:         $X, Att_l, K_X, V_X \leftarrow layer_l(X)$                ▷ $X : hidden\ states\ before\ TSP$
 5:         $K, V \leftarrow KVCompress(K_X, V_X, Att_l, R_{KV})$
 6:         if $l == L_{TSP}$ then
 7:             $x \leftarrow HiddenCompress(X, Att_l, R_{TSP})$      ▷ $x : reduced\ hidden\ states$
 8:         end if
 9:     else
10:         $x, Att_l, K_x, V_x = layer_l(x)$
11:         $K, V \leftarrow KVCompress(K_x, V_x, Att_l, R_{KV})$
12:     end if
13:     $C \leftarrow update(K, V)$
14: end for
15: $O \leftarrow LMHead(x)$
16: return $O, C$
```

B.2 Distinction from Prefill-Aware KV Cache Compression Methods

We compare FastKV with baseline prefill-aware KV cache compression methods. Figure 6 illustrates the differences among the three approaches.

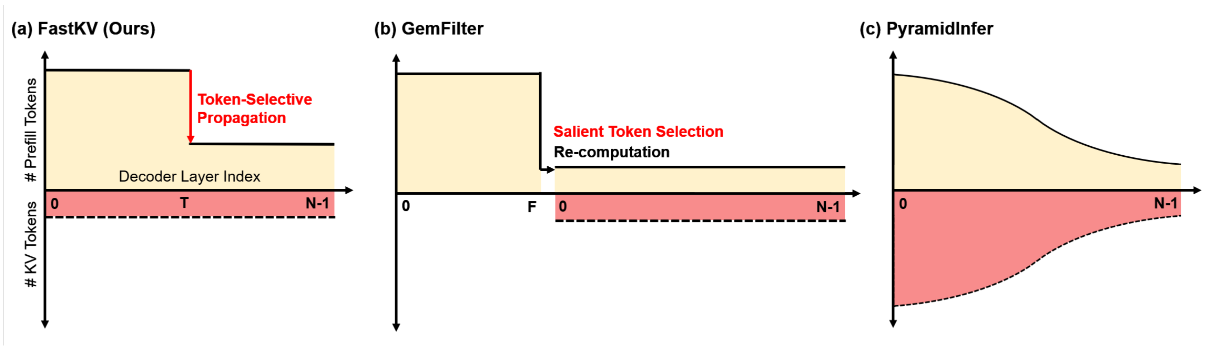

Figure 6: Comparison of processing flows between GemFilter and FastKV. GemFilter prunes the input prompt and recomputes later layers with only selected tokens, while FastKV propagates the full-context to a mid-layer before reducing the token set for further computation.

First, FastKV propagates the hidden states of all input context tokens up to the Token-Selective Propagation (TSP) layer T. At this layer, Token-Selective Propagation occurs: only the selected salient tokens and window tokens are passed to the later layers. Consequently, later layers process fewer tokens than the early layers, since they operate on the reduced token set. Meanwhile, KV cache compression is applied independently at each decoder layer once its prefill computation is complete, according to a separately configured KV retention rate.

In contrast, GemFilter performs prefill up to the filter layer F in order to select salient tokens. Once this set is fixed, the prefill stage is restarted with the reduced token set. This policy enforces the salient token selection made at the filter layer across all decoder layers, which can be problematic for early layers where attention score distributions vary significantly. Moreover, the prefill compute rate is directly tied to the KV retention rate, preventing independent control of prefill compute and KV budget.

PyramidInfer, on the other hand, gradually reduces the number of tokens processed at each decoder layer during prefill. The decay rate follows a cosine schedule, which can lead to premature context reduction before the critical context is stabilized. In addition, the full KV cache of all prefilled tokens in each layer is retained during decoding. As a result, in PyramidInfer the prefill compute rate is always equal to the KV retention rate, further restricting the ability to optimize the trade-off between decoding latency and accuracy.

An additional limitation of GemFilter stems from its strategy of discarding non-salient tokens and restarting prefill. As shown in Figure 7, the information carried by discarded tokens is never represented, often leading to fragmented context understanding. In contrast, FastKV allows tokens that are ultimately discarded by TSP to still contribute through attention computations in the early layers, ensuring that even these tokens can influence the output. As a result, the later layers operating on the reduced hidden states are able to access semantically enriched representations. This design enables FastKV to preserve contextual fidelity while still achieving substantial latency and memory savings.

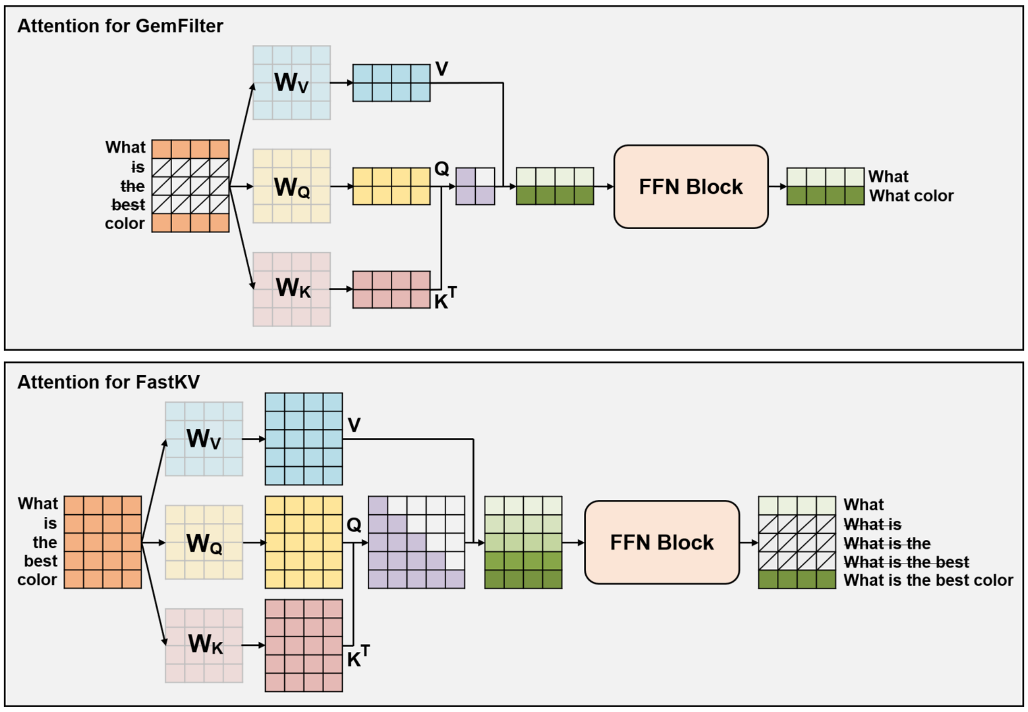

Figure 7: Visualization of attention computation. In GemFilter, discarded tokens are entirely excluded from attention, leading to fragmented context representations. FastKV discards tokens after the TSP layer, but their information has already been integrated through early-layer attention, allowing global semantics to be preserved despite later compression.

C Evaluation Details

C.1 Models

We conduct our main evaluation using two open-source LLMs: LLaMA-3.1-8B-Instruct and Ministral-8B-Instruct. LLaMA-3.1-8B-Instruct has 32 decoder layers and we choose layer 15 for the TSP layer. Ministral-8B-Instruct has 40 decoder layers and we choose layer 17 for the TSP layer. Both of the models support the context window size opf 128K tokens.

The checkpoints for all models are publicly available at:

LLaMA-3.1-8B-Instruct:
https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct
Mistral-8B-Instruct:
https://huggingface.co/mistralai/Mistral-8B-Instruct-2410

C.2 Datasets

LongBench. LongBench is a collection of benchmark datasets designed to assess the long-context understanding capabilities of LLMs. It includes tasks in both English and Chinese, spanning six categories, each consisting of multiple subtasks.

In this study, we evaluate FastKV on a subset of LongBench that includes only the English-language tasks and code-related benchmarks. Detailed information for each selected subtask is provided in Table 5.

Table 5: LongBench dataset details.

Task Type	Task	Abbreviation	Eval metric	Language	#Sample
Single-doc QA	NarrativeQA	NrtvQA	F1	EN	200
	Qasper	Qasper	F1	EN	200
	MultiFieldQA-en	MF-en	F1	EN	150
	HotpotQA	HotpotQA	F1	EN	200
Multi-doc QA	2WikiMultihopQA	2WikiMultihopQA	F1	EN	200
	MuSiQue	MuSiQue	F1	EN	200
Summarization	GovReport	GovReport	Rouge-L	EN	200
	QMSum	QMSum	Rouge-L	EN	200
	MultiNews	MultiNews	Rouge-L	EN	200
Few shot	TREC	TREC	Accuracy	EN	200
	TriviaQA	TriviaQA	F1	EN	200
	SAMSum	SAMSum	Rouge-L	EN	200
Code	LCC	LCC	Edit Sim	Python/C#/Java	500
	RepoBench-P	RB-P	Edit Sim	Python/Java	500
Synthetic	PassageCount	PCount	Accuracy	EN	200
	PassageRetrieval-en	PRe	Accuracy	EN	200

RULER. RULER is a synthetic benchmark for evaluating long-context LLMs with flexible configurations for customized sequence length and task complexity. It extends the Needle-in-a-Haystack paradigm into a broader benchmark suite with 13 tasks across four categories: retrieval (NIAH-style), aggregation (CWE, FWE), multi-hop tracing (VT), and QA. This design enables systematic stress tests that probe retrieval, aggregation, and reasoning beyond simple lookup.

In our evaluation, we used 11 benchmarks spanning retrieval, aggregation, and multi-hop tracing, excluding the two QA benchmarks.

Needle-in-a-Haystack. Needle-in-a-Haystack is a benchmark designed to evaluate the in-context retrieval capabilities of LLMs under long-context settings. A statement, referred to as the needle, is randomly selected and inserted at a specific position within a long input context. The model is tasked with retrieving the needle content given the entire context. Evaluation is conducted across various context lengths and needle depths (insertion positions) to systematically measure retrieval performance under different levels of difficulty.

In our experiments, retrieval performance is evaluated at context lengths ranging from 16K to 128K tokens, with measurements taken at 16K-token intervals.

D More Experimental Results

D.1 LongBench

We provide full breakdown of LongBench results for LLaMA-3.1-8B-Instruct and Ministral-8B-instruct. Table 6 presents the score of each subtask of LongBench.

For H2O and PyramidInfer, we truncate inputs longer than 8K tokens to 8K tokens, as longer sequences cause out-of-memory errors due to their naive attention implementations, which are not optimized for long-context processing. We extended a common practice in LongBench evaluations when handling inputs that exceed a model's supported context window size.

Table 6: Detailed LongBench results for LLaMA-3.1-8B-Instruct and Ministral-8B-Instruct.

	Prefill	KV	Single-Document QA			Multi-Document QA			Summarization			Few-shot Learning			Synthetic		Code	
Method	Compute	Retain	NrtvQA	Qasper	MF-en	HotpotQA	2WikiMQA	Musique	GovReport	QMSum	MultiNews	TREC	TriviaQA	SAMSum	Pcount	PRe	LCC	RB-P
LLaMA-3.1-8B-Instruct																		
Full-context	100%	100%	30.21	45.53	55.01	56.01	46.65	31.28	35.13	25.28	27.25	73.00	91.64	43.80	8.91	99.50	63.38	56.64
Decoding-Only																		
StreamingLLM	100%	10%	26.05	26.29	33.68	48.07	37.93	25.03	25.37	21.49	19.97	58.50	88.46	42.61	7.91	88.00	61.07	54.94
	100%	20%	27.99	28.89	34.42	51.21	41.03	24.43	28.34	22.08	22.16	64.00	89.96	42.27	8.66	85.00	62.02	56.85
H2O	100%	10%	9.32	34.96	37.08	19.65	32.65	6.91	31.29	21.01	25.59	66.50	91.03	42.16	3.41	65.75	58.27	51.48
	100%	20%	9.80	41.09	41.00	19.20	33.92	6.83	32.26	21.50	25.62	66.00	91.27	42.41	4.31	68.25	62.02	54.08
SnapKV	100%	10%	31.51	40.18	41.00	55.83	44.30	30.92	28.88	24.58	22.57	70.50	91.28	42.64	7.93	99.50	62.32	56.71
	100%	20%	31.15	44.11	54.55	55.47	45.14	31.14	31.22	24.94	24.09	70.50	91.90	42.23	7.96	99.50	63.55	57.91
Prefill-Aware																		
PyramidInfer	60%	60%	12.10	35.54	37.58	17.66	33.71	6.56	27.84	21.62	21.04	64.00	91.89	43.31	3.65	70.08	64.92	57.90
GemFilter	51%	10%	24.36	21.07	39.73	25.99	43.92	35.78	28.94	21.42	17.62	61.00	91.53	40.88	4.76	85.00	59.95	44.38
	61%	20%	26.02	30.74	47.30	37.97	55.97	20.64	31.19	20.92	21.06	61.50	92.75	40.92	6.18	97.00	31.99	41.12
FastKV	60%	10%	30.54	40.75	54.57	54.33	46.30	30.55	28.15	24.29	21.91	73.00	92.38	42.96	7.36	99.50	60.11	54.79
	60%	20%	30.26	43.70	54.83	54.39	46.42	30.42	30.28	24.88	22.93	73.50	91.97	43.07	7.61	99.50	61.61	55.67
Ministral-8B-Instruct																		
Full-context	100%	100%	24.55	47.99	52.21	60.43	52.64	35.28	32.36	24.16	26.64	74.50	92.04	45.94	9.00	100.00	67.06	67.19
Decoding-Only																		
StreamingLLM	100%	10%	22.52	33.02	27.69	48.68	37.66	21.26	23.55	21.09	18.36	61.50	91.38	43.45	6.00	63.00	62.84	62.55
	100%	20%	23.25	35.91	28.26	51.46	41.13	24.08	26.88	21.38	21.43	67.50	91.88	44.65	5.00	72.00	63.17	63.21
H2O	100%	10%	27.75	40.61	44.04	52.19	48.78	30.68	29.40	23.14	25.35	72.00	90.66	43.26	7.50	70.00	61.99	59.91
	100%	20%	27.28	43.31	44.57	53.41	49.93	30.88	30.47	23.55	25.87	72.00	91.16	44.04	7.50	71.00	65.02	66.47
SnapKV	100%	10%	26.82	43.55	49.53	60.18	51.72	36.38	26.94	23.55	21.27	72.00	92.04	44.01	10.00	100.00	65.02	66.47
	100%	20%	26.72	45.34	49.83	61.11	52.07	36.58	29.36	23.99	23.65	74.50	92.04	45.37	9.00	100.00	65.55	65.65
Prefill-Aware																		
PyramidInfer	60%	60%	24.92	37.60	40.34	48.92	40.84	22.26	27.23	22.99	22.25	68.50	91.06	45.02	7.50	97.00	66.91	64.74
GemFilter	60%	10%	25.70	37.97	39.73	60.62	56.04	37.22	28.65	21.18	19.57	63.00	89.36	42.79	6.50	85.50	59.96	41.79
	70%	20%	28.06	43.81	50.56	63.46	59.43	39.25	30.59	23.31	22.03	70.50	90.84	43.26	5.50	97.00	27.56	46.03
FastKV	60%	10%	25.31	43.71	50.11	61.31	50.85	36.73	26.15	23.45	20.54	75.00	92.04	45.10	10.00	100.00	62.99	65.00
	60%	20%	25.90	46.05	51.29	61.24	50.21	36.15	28.65	23.84	22.57	76.00	92.04	46.20	10.00	100.00	64.68	64.68

D.2 Needle-in-a-Haystack

We provide full breakdown of Needle-in-a-Haystack results for LLaMA-3.1-8B-Instruct in Figure 8. The KV retention rate is fixed to 10% for all methods, while GemFilter and FastKV reduces prefill compute rate to 51% and 60%, respectively.

FastKV achieves the highest score among all evaluated baselines. FastKV even outperforms the full-context, as Token-Selective Propagation helps the model to focus on globally critical tokens by simplifying the input context.

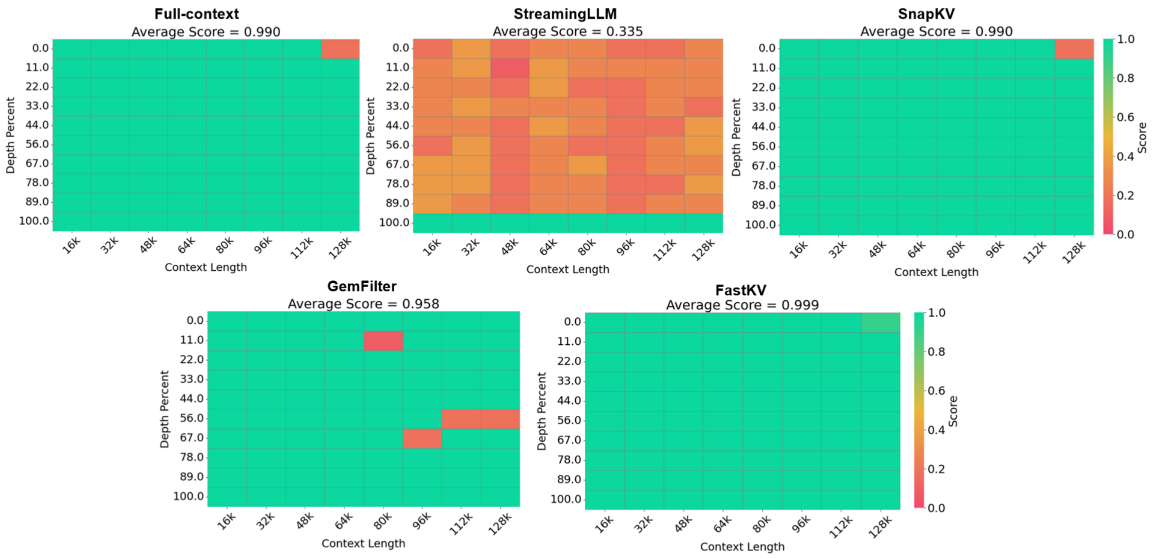

Figure 8: Needle-in-a-Haystack results of LLaMA-3.1-8B-Instruct with 10% KV retention rate.

D.3 Latency

We present the end-to-end inference latency breakdown of Ministral-8B-Instruct in Figure 9. The results were profiled with varying context lengths of 8K, 32K, and 128K tokens, while generation length is fixed to 256 tokens.

Prefill compute rate is set to 60% for PyramidInfer and FastKV, and 51% for GemFilter. The KV retention rate is set to 10% for all methods except PyramidInfer. The KV retention rate of PyramidInfer is identical to its prefill compute rate, which is 60%.

All of the experiments were conducted on a NVIDIA A100 SXM GPU.

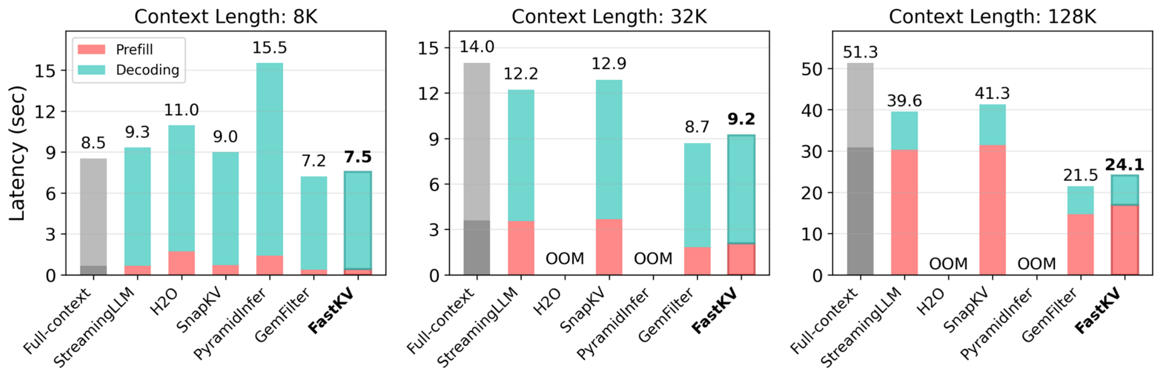

Figure 9: End-to-end inference latency breakdown of Ministral-8B-Instruct at varying input context lengths (generating 256 tokens).

The overall trend is consistent with the results observed on LLaMA-3.1-8B-Instruct. As the context length increases, the end-to-end latency grows, with prefill latency accounting for an increasingly larger proportion of the total.

StreamingLLM and SnapKV effectively reduce decoding latency, but since they do not mitigate prefill latency, their benefits diminish in long-context scenarios where prefill dominates. H2O and PyramidInfer cannot utilize FlashAttention-2 due to implementation constraints, leading to out-of-memory errors beyond 8K tokens, and they are even slower than the full-KV baseline at 8K. PyramidInfer exhibits particularly large decoding latency, as its KV retention rate is 60%, forcing it to maintain a substantially larger KV cache than other methods that operate with only a 10% budget.

By contrast, both GemFilter and FastKV address latency in both the prefill and decoding phases, achieving consistent acceleration across all settings and delivering over 2$\times$ speedup at a 128K context length. For LLaMA-3.1-8B-Instruct, FastKV was slightly slower than GemFilter because the filter layer (layer 13) precedes the TSP layer (layer 15). In Ministral, however, both the filter layer and the TSP layer are located at layer 17, resulting in identical latency for the two methods.