Adaptive Layer Selection for Layer-Wise Token Pruning in LLM Inference

[1]Rei Taniguchi*, [2]Yuyang Dong, [3]Makoto Onizuka, [3,4]Chuan Xiao
[1]NEC Corporation, [2]Initial S, [3]Osaka University, [4]Nagoya University
rei-taniguchi@nec.com, {onizuka, chuanx}@ist.osaka-u.ac.jp, dongyuyang@initial-s.com

Abstract

Due to the prevalence of large language models (LLMs), key-value (KV) cache reduction for LLM inference has received remarkable attention. Among numerous works that have been proposed in recent years, layer-wise token pruning approaches, which select a subset of tokens at particular layers to retain in KV cache and prune others, are one of the most popular schemes. They primarily adopt a set of pre-defined layers, at which tokens are selected. Such design is inflexible in the sense that the accuracy significantly varies across tasks and deteriorates in harder tasks such as KV retrieval. In this paper, we propose ASL, a training-free method that adaptively chooses the selection layer for KV cache reduction, exploiting the variance of token ranks ordered by attention score. The proposed method balances the performance across different tasks while meeting the user-specified KV budget requirement. ASL operates during the prefilling stage and can be jointly used with existing KV cache reduction methods such as SnapKV to optimize the decoding stage. By evaluations on the InfiniteBench, RULER, and NIAH benchmarks, we show that ASL, equipped with one-shot token selection, adaptively trades inference speed for accuracy, outperforming state-of-the-art layer-wise token pruning methods in difficult tasks.

1 Introduction

Large language models (LLMs) have demonstrated remarkable capabilities in processing long contexts, enabling applications such as long document analysis, extended conversations, and software engineering. However, the memory footprint of LLM inference is a critical issue due to the key-value (KV) cache, which stores past tokens' key and value vectors for efficient generation.

To address this challenge, numerous KV cache reduction techniques have emerged (Zhang et al., 2023; Ge et al., 2023; Li et al., 2024; Feng et al., 2024; Fu et al., 2024b). Among these, layer-wise token pruning methods (Shi et al., 2024; Jo et al., 2025; Cai et al., 2024; Yang et al., 2024; Fu et al., 2024a) have recently gained considerable attention by exploiting attention patterns across Transformer layers (Vaswani et al., 2017). To achieve significant memory reduction while reducing accuracy loss, they select a subset of important tokens at particular layers, calculating attention only for these tokens and retaining their KV cache in subsequent layers.

Despite their effectiveness, existing layer-wise token pruning methods suffer from a critical limitation: tokens are selected on pre-defined, fixed layers (referred to as "selection layers") that are determined independently of the task. As illustrated in Figure 1, this inflexible design leads to substantial performance variation across tasks. For simpler tasks like question answering (QA) where relevant information can be identified early, token

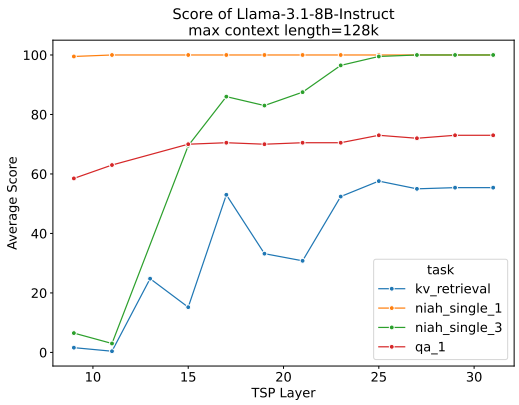


Figure 1: Performance of FastKV (Jo et al., 2025) on four tasks under different selection layer (referred to as "TSP layer" in Jo et al. (2025)) settings: KV retrieval in InfiniteBench (Zhang et al., 2024), single-key NIAH (with varying difficulties) and QA in RULER (Hsieh et al., 2024). KV compression before selection layer is disabled to highlight the impact.

*Work done at Osaka University.
Source code is available at https://github.com/TANIGUCHIREI/ASL.

Findings of the Association for Computational Linguistics: ACL 2026, pages 967–986
July 2-7, 2026

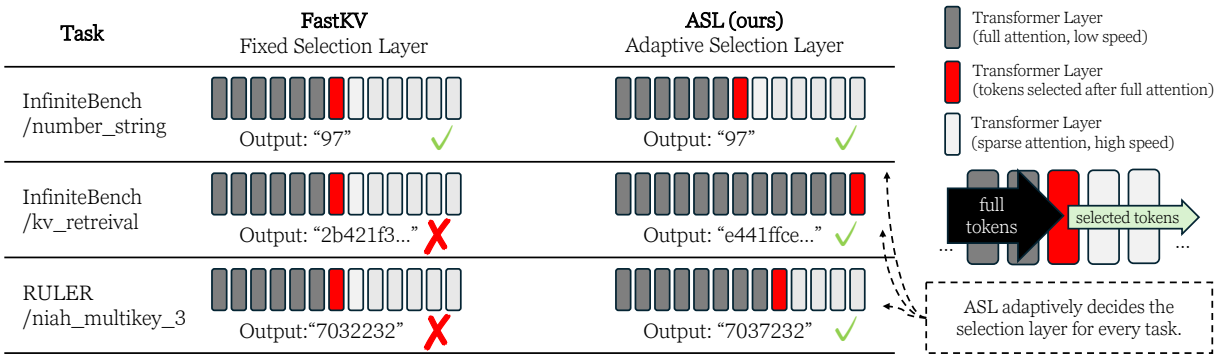


Figure 2: Comparison of FastKV and ASL.

selection at middle layers (e.g., layer 15 for Llama 3.1 8B, as suggested in Jo et al. (2025)) achieves strong performance. However, for harder tasks like KV retrieval, where high semantic similarity between context and query makes early identification difficult, the same selection layers result in severe accuracy degradation. To maintain acceptable performance on challenging tasks, these methods must either postpone token selection to deeper layers or increase the KV budget, thereby compromising their memory reduction benefits.

In this paper, we propose Adaptive Selection Layer (ASL), a task-aware method that adaptively determines the selection layer based on the attention patterns observed during inference. Our idea is to monitor the variance of token ranks ordered by attention score across consecutive layers. When this variance decreases below a threshold, it signals that attention scores have consistently focused on a stable subset of tokens, indicating the moment for token selection. By computing variance and comparing it against a user-specified threshold, ASL adapts to tasks of varying difficulty without requiring manual tuning for each scenario. Figure 2 depicts the comparison of FastKV and ASL.

ASL operates during the prefilling stage with minimal overhead, storing only pooled attention scores from recent layers. Once the selection layer is determined, tokens are selected and propagated to all subsequent layers in a one-shot manner. To meet specific KV budget requirements, ASL can be seamlessly integrated with existing methods. For example, it can be combined with SnapKV (Li et al., 2024) to optimize the decoding stage, and can be integrated into GemFilter (Shi et al., 2024) to enable a two-pass strategy for enhanced accuracy.

We evaluate ASL on three long-context benchmarks, InfiniteBench, RULER, and Needle in a Haystack (NIAH) (Kamradt, 2024), with up to 256k context lengths. Experimental results demonstrate that ASL can outperform state-of-the-art layer-wise token selection methods including FastKV, GemFilter, and PyramidInfer in accuracy, while maintaining comparable decoding speed and KV cache reduction.

Our contributions are summarized as follows: (1) We identify the inflexibility of fixed selection layers in existing layer-wise pruning methods and demonstrate how task difficulty affects optimal layer selection. (2) We propose ASL, a novel adaptive method that determines selection layers by monitoring the variance of token ranks ordered by attention score across layers, enabling task-aware KV cache reduction. (3) We demonstrate that ASL can be integrated with existing KV cache reduction methods to optimize both prefilling and decoding stages while meeting user-specified KV budgets. (4) Through comprehensive experiments on multiple benchmarks, we show that ASL achieves superior accuracy-efficiency trade-offs compared to state-of-the-art methods, particularly excelling on difficult tasks where existing methods fail.

2 Preliminaries

LLM inference typically involves two stages:

- **Prefilling:** This stage occurs when the LLM processes the input prompt at once. For each Transformer layer, the LLM computes query, key, and value vectors for all tokens. Attention is applied across all pairs of input tokens, leading to full self-attention. KV cache is initialized and stores keys and values for all past tokens.
- **Decoding:** This stage outputs tokens one at a time, using the previously generated tokens and their cached keys and values. Only the latest token is passed through the Transformer. The LLM computes the query for this new token, and calculates attention between the new query and all cached keys and values from prior tokens, which are retrieved from the KV cache.

Table 1: List of notable layer-wise token pruning methods. Taxonomy is explained in Section 2.2.

Type	Methods
One-shot	GemFilter (Shi et al., 2024), FastKV (Jo et al., 2025)
Progressive	PyramidKV (Cai et al., 2024), PyramidInfer (Yang et al., 2024), LazyLLM (Fu et al., 2024a), PromptDistill (Jin et al., 2025), SlimInfer (Long et al., 2025)
Sandwiched	OmniKV (Hao et al., 2025)
Grouped	SqueezeAttention (Wang et al., 2024), EvolKV (Yu and Chai, 2025)
Adaptive	DynamicKV (Zhou et al., 2024), CAKE (Qin et al., 2025)

2.1 KV Cache Reduction

To reduce KV cache size, which is essential in long-context scenarios, token eviction techniques have been extensively explored. StreamingLLM (Xiao et al., 2023) keeps only global (first few tokens, a.k.a. attention sinks) and local (most recent) tokens in the sequence. H2O (Zhang et al., 2023), FastGen (Ge et al., 2023), and SnapKV (Li et al., 2024) exploit attention sparsity and retain the most influential tokens' KV cache for the decoding stage, which are decided using heuristics during the prefilling stage. AdaKV (Feng et al., 2024) and HeadKV (Fu et al., 2024b) extend SnapKV with a head-wise budget allocation strategy.

Another line of works does not evict tokens but loads a subset of tokens' KV cache by leveraging attention sparsity (Tang et al., 2024; Singhania et al., 2024) or offloading to CPU memory (Lee et al., 2024; Sun et al., 2024). Besides these methods based on token selection, quantization (Liu et al., 2024c; Hooper et al., 2024) is also an approach to reducing KV cache size. Moreover, LSH (Charikar, 2002) has been utilized to approximate the attention score distribution and estimate attention output (Chen et al., 2024). For other works on KV cache reduction, we refer readers to a GitHub repository (Chen, 2024) for an up-to-date list of papers.

2.2 Layer-Wise Token Pruning

Among the methods for KV cache reduction, many recent ones adopt a layer-wise token pruning paradigm. Observing the attention patterns across Transformer layers, they exploit either the similarity between adjacent layers or dissimilarity in earlier and later layers, and select a subset of tokens—typically by top-k or top-p (cumulative attention score is no less than p percentile of full attention)—to calculate attention and retain in the KV cache. Other tokens are pruned or offloaded to CPU memory (Hao et al., 2025; Long et al., 2025).

Table 1 summarizes a list of notable layer-wise token pruning methods, categorized into five types.

- **One-shot** methods select tokens only once at a specific layer, and all deeper layers process only the selected tokens. FastKV (Jo et al., 2025) and GemFilter (Shi et al., 2024) are two representative methods in this category. FastKV adopts a one-pass strategy: (1) from layer 0 to the selection layer, full attention is calculated; (2) at the selection layer, top-k tokens are selected; (3) for subsequent layers, attention is calculated only for the selected tokens. To meet the KV budget requirement, SnapKV (Li et al., 2024) is jointly used in FastKV to compress the KV cache from layer 0 to the selection layer. GemFilter adopts a two-pass strategy: (1) the first pass calculates full attention and selects top-k tokens at the selection layer; (2) the second pass starts from layer 0 and processes all layers, with attention calculated and KV cache retained for the selected tokens only.
- **Progressive** methods employ a multi-shot token selection scheme, progressively reducing the tokens processed for attention and retained in the KV cache. Most methods in this category, except PyramidKV (Cai et al., 2024), are monotonic methods—a token pruned at a layer is also pruned at all later layers.
- **Sandwiched** methods (in particular, OmniKV (Hao et al., 2025)) calculate attention for all tokens at some layers and selected tokens for others, thereby exhibiting a sandwich shape across layers.
- **Grouped** methods divide layers into several groups and allocate a KV cache budget to each group, such that the allocated values sum up to a user-specified total budget. The allocation is processed online (i.e., during inference) in SqueezeAttention (Wang et al., 2024) and offline in EvolKV (Yu and Chai, 2025).
- **Adaptive** methods directly allocate the KV cache budget to each layer. Online allocation is used in DynamicKV (Zhou et al., 2024) and CAKE (Qin et al., 2025).

In addition to token pruning, layer-wise observations have also been used to build cross-layer merging or sharing methods, such as block-based

layer pruning (BBLP) (Gromov et al., 2024), MiniCache (Liu et al., 2024a), FoldGPT (Liu et al., 2024b), SwiftKV (Qiao et al., 2024), where a small amount of fine-tuning or distillation is needed in BBLP, FoldGPT, and SwiftKV. It is also noteworthy to mention that layer-wise behavior is evaluated in previous studies such as Xiao et al. (2023) and Li et al. (2024), yet they do not belong to the category of layer-wise token pruning because their token selection in each layer is performed independently.

3 Observations

The layer-wise token pruning methods summarized in Table 1, despite achieving significant peak memory reduction and fast decoding speed, only a minority of them, including GemFilter, FastKV, PyramidInfer, LazyLLM, and SlimInfer, optimize time to first token (TTFT), a key metric that evaluates the efficiency of the prefilling stage. They select tokens either at a pre-defined set of layers (GemFilter, FastKV, LazyLLM, and SlimInfer) or with a decay ratio across layers (PyramidInfer).

Such design is inflexible in the sense that the difficulty of tasks is not considered, rendering these methods either incapable of delivering competitive performance for harder tasks such as KV retrieval and multi-key NIAH, or have to compromise its KV cache reduction (by postponing token selection to later layers or increasing the KV budget) to cope with these harder tasks, as we have seen in Figure 1.

As discussed in Jiang et al. (2024a), the difficulty of the task originates from the semantic similarity between the question and the context. A task tends to be easier if the similarity is low, as the LLM can easily identify the context related to the question. In such tasks, it is possible to locate the tokens necessary for the answer in early layers. In contrast, in the harder KV retrieval task, the context consists of key-value pairs, rendering high similarity between the question and the context. As a result, token selection at early layers cannot successfully locate the tokens required for the answer.

Cai et al. (2024) found that in early layers, attention scores are generally distributed in a uniform manner across the tokens in the context, and in later layers, the scores tend to localize to a fixed subset of tokens. While such patterns were observed for an RAG task in Cai et al. (2024), we find that they apply to various tasks. Figure 3 shows that both KV retrieval and QA tasks, whose difficulties significantly differ, exhibit similar attention

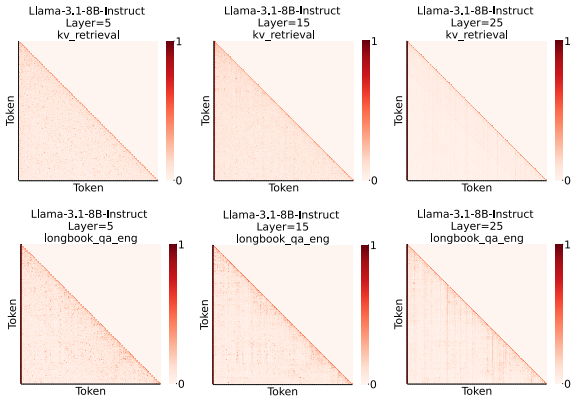


Figure 3: Attention patterns of KV retrieval (upper) and QA (lower). At early layers, the attention scores are roughly uniformly distributed across the context. At middle layers, a subset of tokens exhibit high scores (as shown in stripe-like regions). The scores are more localized at deep layers, focusing to a smaller subset of tokens (as shown in thin vertical lines).

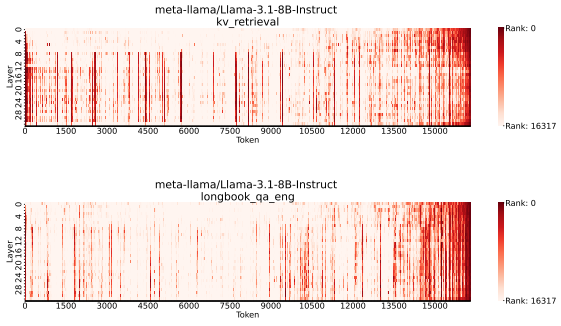


Figure 4: Ranks patterns across layers for KV retrieval (upper) and QA (lower), context length = 16k.

patterns. Based on this observation, we can design an adaptive method that decides the selection layer for layer-wise token pruning: token pruning is performed when attention scores start to focus consistently on a small subset of tokens.

4 Adaptive Selection Layer

4.1 Variance of Token Ranks

To find when attention scores start to focus consistently to a subset of tokens, our idea is to monitor how the tokens in the context, ranked by the attention score, vary across layers. As shown in Figure 4, the top ranks tend to fix at deeper layers for both KV retrieval and QA tasks. Seeing this, we calculate the variance of ranks as an indicator: a small variance of ranks indicates not only the scores are more focused to a fixed subset of tokens, but also the order of these tokens, ranked by the score, is relatively fixed.

Figure 5 depicts the method that calculates the

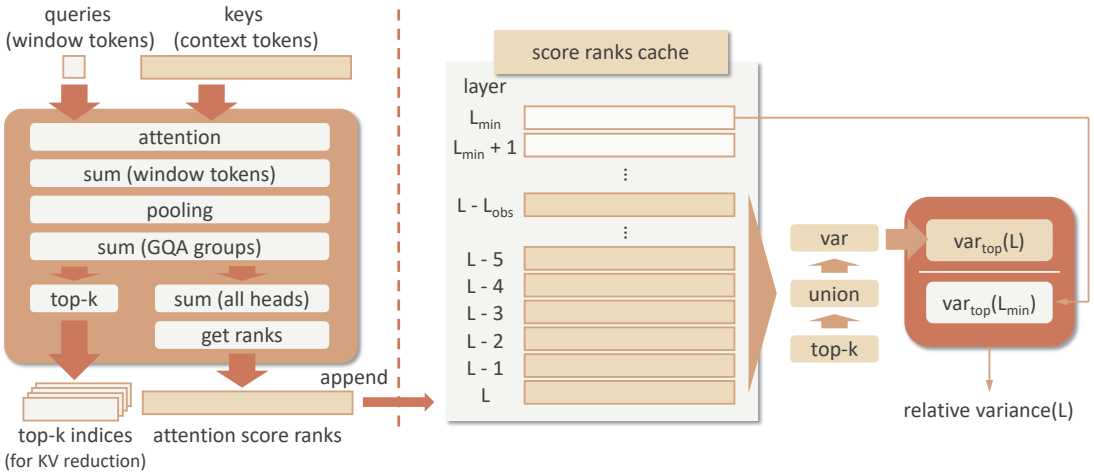


Figure 5: Relative variance calculation.

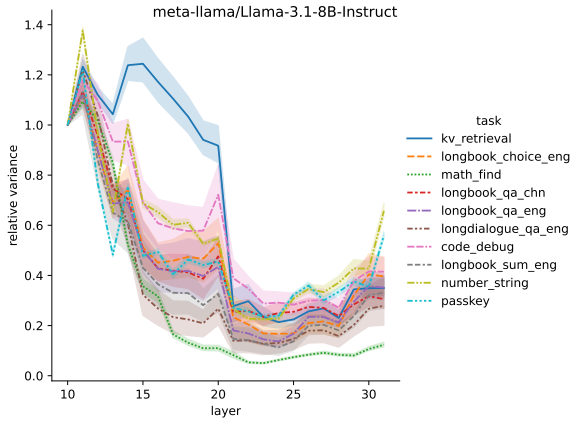


Figure 6: Relative variances across layers for 10 tasks in the InfiniteBench benchmark.

variance. We start with layer L_{min} of the LLM and consider the ranks in every L_{obs} consecutive layers, thereby eliminating the effect of the layers too early or too distant from the current layer. L_{min} and L_{obs} and two hyperparameters. Attention scores are aggregated over attention heads. Instead of using raw attention scores, we perform 1D average pooling to smooth noise and capture contiguous regions in the context, in line with SnapKV (Li et al., 2024). The pooled attention scores, denoted $PA(L)$, are calculated as follows.

$$\text{PA} = \text{pool}\left(\text{softmax}\left(\frac{\mathbf{q}_w\mathbf{k}_c + \mathbf{m}_w}{\sqrt{d}}\right)\right)$$

where $\mathbf{q}_w$ denotes the query vector of current window, $\mathbf{k}_c$ denotes the key vector of the context, and $\mathbf{m}_w$ denotes causal masking.

We sum up the average pooled attention scores over GQA head groups, which are used to order the tokens.

$$\text{scores} = \sum_{i\in\text{groups}}\sum_{j\in\text{heads}} \text{PA}[i,j,:].$$

Due to the attention sparsity, not all tokens in the context need to be considered for variance calculation. From layers $L - L_{\text{obs}}$ to L, we identify the top-k tokens and get a union of the top-k's:

$$\text{top}(L) = \text{sort}\left(\bigcup_{l=L-L_{\text{obs}}}^{L} \text{top-}k(\text{scores}(l))\right).$$

The union serves as a subset of tokens with high attention scores. The variance of the ranks is then calculated for the tokens in the union:

$$\text{var}_{\text{top}}(L) = \frac{1}{|\text{top}(L)|}\sum_{t\in\text{top}(L)} \text{var}\left(R_t[L - L_{\text{obs}}, L]\right)$$

where $R_t[L - L_{\text{obs}}, L]$ denotes the set of token t's ranks from layers $L - L_{\text{obs}}$ to L.

As a task-aware design, we divide each variance by the initial variance at layer L_{min} to obtain a relative variance

$$\text{relative\_variance}(L) = \frac{\text{var}_{\text{top}}(L)}{\text{var}_{\text{top}}(L_{\text{min}})}.$$

The relative variance is then compared with a user-specified threshold τ. If the relative variance is less than τ, the attention scores are regarded as consistently focused, and the current layer becomes the selection layer.

Figure 6 shows how the relative variance changes with layers for 10 tasks in the InfiniteBench benchmark. If we set $\tau = 0.4$, the math find and dialog QA tasks will have the earliest selection layer among the 10 tasks, while the KV retrieval and code debug tasks have the deepest selection layer.

We name this method adaptive selection layer (ASL) and provide pseudo-codes in Algorithms 1 and 2, Appendix C, both acting in the prefilling stage.

4.2 Using ASL for LLM Inference

Prefilling. During the prefilling stage, from layer $L_{\min}$, ASL computes and stores the pooled mean attention scores, which are used in the next L_{obs} layers. Since the pooled scores are aggregated over the attention heads and only the most recent L_{obs} layers need to be kept, they incur only a small amount of peak memory usage. For example, when $L_{\text{obs}} = 8$, for Llama 3.1 8B with 32 layers and 8 KV heads, the pooled scores incurs an additional $\frac{L_{\text{obs}}}{32 \times 8} = \frac{1}{32}$ memory usage compared to attention calculation. When the selection layer is determined, token selection is performed and the prefilling proceeds with only the selected tokens.

Decoding. When new tokens are decoded, only the KV entries for these new tokens are added to the cache and used in subsequent attention calculation.

To meet the KV budget requirement, ASL can work jointly with existing KV cache reduction methods. For example, SnapKV can be used to select KV entries for all the layers prior to the selection layer. Supposing the same k is used for the top-k selection by ASL and SnapKV, this integration guarantees that the KV cache size of each layer is exactly k during decoding. ASL can be also used with the two-pass method GemFilter. In the first pass, tokens are selected once the selection layer is determined. Then, we start the second pass from layer 0 by processing only the selected tokens.

We provide a theoretical analysis in Appendix B for the costs of ASL's prefilling and decoding, comparing with FastKV and full attention.

5 Experiments

5.1 Experimental Setup

Models. We evaluate two long-context LLMs: (1) Llama-3.1-Nemotron-8B-UltraLong-1M-Instruct (Llama-3.1-8B-UL, for short), with 32 layers, and (2) Qwen2.5-7B-Instruct-1M (Qwen2.5-7B, for short), with 28 layers.

Benchmarks. We use three benchmarks for evaluation: (1) InfiniBench (Zhang et al., 2024), with an average context length of 214k. (2) RULER (Hsieh et al., 2024), with context length ranging from 4k to 128k, and (3) Needle in a Haystack (NIAH) (Kamradt, 2024), with context length from 1k to 256k.

Methods. We compare ASL with three layer-wise token pruning methods: (1) FastKV (Jo et al., 2025), (2) GemFilter (Shi et al., 2024), and (3) PyramidInfer (Yang et al., 2024). As introduced in Section 4.1, we select top-k tokens in ASL ranked by pooled mean scores and equip ASL with SnapKV (Li et al., 2024) to optimize decoding. This method, referred to as ASL, serves as a one-pass solution. Moreover, we integrate ASL with GemFilter as a two-pass solution, referred to as ASL_2pass. We set the default KV budget size to 2048. Moreover, to show how ASL compares to FastKV on prefilling, we consider a setting with full KV cache before token selection and a KV budget of 2048 after token selection; i.e., SnapKV is not applied for ASL and KV compression is disabled before token selection in FastKV. $L_{\min}$ is 10 for Llama-3.1-8B-UL and 9 for Qwen2.5-7B. L_{obs} = 8. The default value of $\tau = 0.3$. Other details can be found in Appendix C.

5.2 Accuracy Evaluation

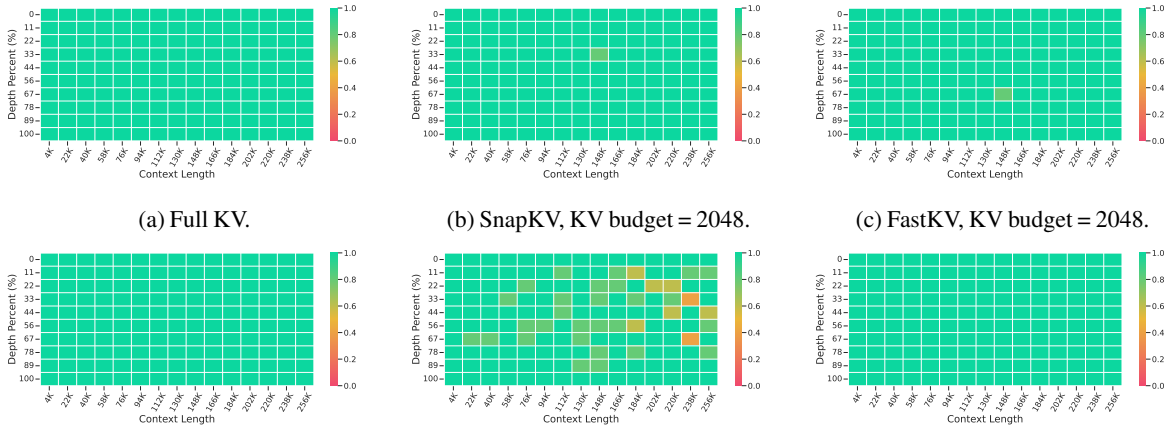
InfiniteBench. Table 2 shows the accuracy on InfiniteBench across 10 tasks. Under the KV budget of 2048, ASL_2pass achieves the highest average score for Llama-3.1-8B-UL. For Qwen2.5-7B, ASL reports the highest average score, tying FastKV. When using full KV before token selection, the gap between adaptive selection (ASL) and user-specified selection (FastKV) is more substantial, especially for hard tasks such as KV retrieval, where the performance of ASL is close to full KV while FastKV is much inferior.

RULER. Table 3 reports the accuracy on RULER with various context lengths. For Llama-3.1-8B-UL, ASL consistently outperforms FastKV. However, ASL_2pass generally performs worse than GemFilter, suggesting less compatibility of ASL with this ultra long variant of Llama 3.1 by NVidia. For Qwen2.5-7B, the advantage of ASL over FastKV is observed when the context length $\geq$ 16k, and ASL_2pass performs better than GemFilter for over 8k contexts. Moreover, the gaps between ASL and existing methods are more substantial for longer contexts such as 128k. Like InfiniteBench, ASL's superiority over FastKV is also more remarkable when using full KV before token selection.

NIAH. Figure 7 shows the results across various context lengths on NIAH, using Qwen2.5-7B. ASL and ASL_2pass report full scores for all the context lengths, delivering the same performance as full KV. SnapKV and FastKV retrieve almost all needles except at the context length of 148k. In contrast, GemFilter reports mediocre performance,

Table 2: Accuracy (↑) comparison on InfiniteBench. PyramidInfer encounters OOM for all tasks and is not reported.

Methods	En.Sum	En.QA	En.MC	En.Dia	Zh.QA	Code.Debug	Math.Find	Retr.PassKey	Retr.Num	Retr.KV	Avg.
Llama-3.1-8B-UL, KV Budget = Full.											
Full KV	27.2	17.73	65.5	11.5	21.26	0	38	100	99.32	16.2	39.7
Llama-3.1-8B-UL, KV Budget = 2048.											
SnapKV	21.86	17.8	65.5	8.5	20.71	0	36.29	100	98.47	2	37.1
FastKV	21.63	17.33	67.69	6	20.14	0	32.29	100	98.47	0.6	36.4
ASL	21.26	18.42	65.5	5.5	20.84	0	35.43	100	98.47	1.8	36.7
GemFilter	5.72	15.78	53.71	13.5	18.88	25.89	36.29	100	100	0	37
ASL_2pass	5.81	18.37	64.63	12	22.3	25.38	27.71	100	100	2.2	**37.8**
Llama-3.1-8B-UL, KV Budget = Full (before selection) & 2048 (after selection).											
FastKV	22.86	16.33	67.69	6.5	20.23	0	32	100	99.49	3.2	36.8
ASL	24.54	17.82	65.5	7	21.66	0	35.71	100	99.49	15.4	**38.7**
Qwen2.5-7B, KV Budget = Full.											
Full KV	33.42	12.63	69.43	9	12.76	0.76	38.57	99.83	100	66.4	44.3
Qwen2.5-7B, KV Budget = 2048.											
SnapKV	28.22	12.02	69.43	3.5	12.02	0.76	35.43	95.93	100	0.4	35.8
FastKV	27.95	11.05	68.56	6.5	11.01	0.51	34.86	99.49	100	1	**36.1**
ASL	28.3	11.89	70.31	3	12	0.25	34.86	99.66	100	0.6	**36.1**
GemFilter	4.69	5.66	27.51	21.5	6.57	3.3	7.71	99.66	100	0	27.7
ASL_2pass	4.81	12.98	63.32	13	11.38	3.3	13.71	98.47	100	5.2	32.6
Qwen2.5-7B, KV Budget = Full (before selection) & 2048 (after selection).											
FastKV	27.99	12.15	68.56	5	11.72	0.51	35.14	99.83	100	1	36.2
ASL	29.52	12.28	70.31	6	11.88	0.25	35.14	99.83	100	52.6	**41.8**



(a) Full KV. (b) SnapKV, KV budget = 2048. (c) FastKV, KV budget = 2048.

(d) ASL, KV budget = 2048. (e) GemFilter, KV budget = 2048. (f) ASL_2pass, KV budget = 2048.

Figure 7: NIAH results of Qwen2.5-7B.

especially for long contexts.

5.3 Efficiency Evaluation

TTFT. Table 4 reports the time to first token (TTFT), varying the context length on RULER. To show how the time scales with the context length, we report the numbers by calculating the ratio to full KV's TTFT. SnapKV, which does not optimize TTFT, report almost the same result as full KV. FastKV, GemFilter, and ASL methods are faster than SnapKV, with GemFilter being the fastest due to its smallest value of selection layer. ASL is slower than FastKV because its token selection is usually at deeper layers than FastKV's.

TPOT. Table 5 reports the time per output token (TPOT), varying context lengths on RULER. Like TTFT, the numbers are also reported in the ratio to full KV. All the KV cache reduction methods exhibit faster decoding speed than full KV. For Llama-3.1-8B, FastKV, GemFilter and ASL methods are faster than SnapKV, and similar TPOTs are observed for these methods under longer contexts. For Qwen2.5-7B, SnapKV, FastKV, and ASL report similar TPOTs, whereas GemFilter becomes the fastest and ASL_2pass shows competitive speed for context lengths over 64k.

Table 3: Accuracy (↑) comparison on RULER, averaged over 13 tasks.

Methods	4k	8k	16k	32k	64k	128k
Llama-3.1-8B-UL, KV Budget = Full.						
Full KV	94.2	92.0	89.5	81.1	73.1	68.6
Llama-3.1-8B-UL, KV Budget = 2048.						
SnapKV	**93.8**	**88.3**	**81.0**	74.8	63.6	55.6
FastKV	93.4	85.8	79.3	69.5	60.1	55.1
PyramidInfer	76.4	74.7	OOM	OOM	OOM	OOM
ASL	93.7	87.0	79.7	73.0	63.2	56.1
GemFilter	92.5	81.1	79.0	76.0	**71.0**	**66.7**
ASL_2pass	82.2	74.6	74.8	**78.5**	69.7	54.5
Llama-3.1-8B-UL, KV Budget = Full (before selection) & 2048 (after selection).						
FastKV	93.7	87.3	82.9	73.0	63.4	60.6
ASL	**94.2**	**90.5**	**87.2**	**80.0**	**70.5**	**69.2**
Qwen2.5-7B, KV Budget = Full.						
Full KV	94.0	93.0	92.7	89.6	86.5	82.3
Qwen2.5-7B, KV Budget = 2048.						
SnapKV	89.9	77.9	75.3	71.8	66.8	65.0
FastKV	**91.0**	**79.7**	75.3	69.9	63.2	59.1
PyramidInfer	37.9	30.2	OOM	OOM	OOM	OOM
ASL	89.9	77.5	**75.5**	**75.1**	**71.7**	**66.4**
GemFilter	79.7	67.7	66.9	63.8	60.4	56.7
ASL_2pass	78.3	70.7	67.9	71.5	69.1	62.5
Qwen2.5-7B, KV Budget = Full (before selection) & 2048 (after selection).						
FastKV	91.3	80.6	75.8	70.6	64.2	59.1
ASL	**94.0**	**91.2**	**88.0**	**83.6**	**80.9**	**74.2**

Table 4: TTFT (↓) comparison on RULER, averaged over 13 tasks. Ratio to Full KV (= 1) is reported.

Methods	4k	8k	16k	32k	64k	128k
Llama-3.1-8B-UL, KV Budget = 2048.						
SnapKV	1.08	1.05	1.04	1.03	1.01	1.01
FastKV	**0.85**	**0.66**	0.58	0.53	0.51	0.50
PyramidInfer	2.09	3.38	OOM	OOM	OOM	OOM
ASL	1.06	0.92	0.82	0.61	0.67	0.79
GemFilter	1.11	0.74	**0.57**	**0.49**	**0.45**	**0.44**
ASL_2pass	1.65	1.17	0.92	0.65	0.69	0.80
Qwen2.5-7B, KV Budget = 2048.						
SnapKV	1.07	1.04	1.03	1.02	1.01	1.01
FastKV	**0.87**	**0.69**	**0.61**	**0.56**	**0.55**	0.54
PyramidInfer	1.94	3.15	OOM	OOM	OOM	OOM
ASL	1.12	1.03	0.93	0.83	0.79	0.81
GemFilter	1.19	0.83	0.66	0.57	**0.55**	**0.53**
ASL_2pass	1.73	1.43	1.25	1.48	0.79	0.95

Table 5: TPOT (↓) comparison on RULER, averaged over 13 tasks. Ratio to Full KV (= 1) is reported.

Methods	4k	8k	16k	32k	64k	128k
Llama-3.1-8B-UL, KV Budget = 2048.						
SnapKV	0.98	0.94	0.83	0.68	0.48	0.30
FastKV	0.91	0.85	0.77	0.62	0.43	0.27
PyramidInfer	1.57	1.63	OOM	OOM	OOM	OOM
ASL	0.89	0.86	0.77	0.62	0.44	0.28
GemFilter	0.81	0.80	0.71	0.58	**0.40**	**0.25**
ASL_2pass	**0.80**	**0.77**	**0.69**	**0.57**	0.41	**0.25**
Qwen2.5-7B, KV Budget = 2048.						
SnapKV	0.98	0.87	0.70	0.48	0.29	0.15
FastKV	0.99	0.88	0.71	0.48	0.30	0.15
PyramidInfer	1.39	1.32	OOM	OOM	OOM	OOM
ASL	1.01	0.89	0.72	0.49	0.30	0.15
GemFilter	**0.76**	**0.68**	**0.55**	**0.37**	**0.23**	**0.11**
ASL_2pass	**0.76**	0.72	0.62	0.55	**0.23**	0.13

Throughput. Table 6 shows the throughputs of ASL, FastKV, and GemFilter on RULER, 128k context, 2048 KV budget. We report mean, median, 95th percentile, and 99th percentile. The latter two refer to the slowest 5% and 1% queries, respectively. ASL trails FastKV and GemFilter, with mean throughputs of 74% (Llama-3.1-8B-UL) and 69% (Qwen2.5-7B) compared to FastKV. ASL_2pass reports mean throughputs of 67% (Llama-3.1-8B-UL) and 55% (Qwen2.5-7B) compared to GemFilter. For the slowest queries on Qwen2.5-7B, the gap between ASL and FastKV/GemFilter becomes smaller, e.g., 72% throughputs compared to FastKV and 79% compared to GemFilter.

ASL trades throughput for accuracy, as shown in Table 8 for the above setting (Qwen-2.5-7B), where substantial accuracy improvement can be seen. In addition, the number of output tokens

Table 6: Throughput (↑).

Methods	Mean	Median	95th	99th
Llama-3.1-8B-UL, KV Budget = 2048.				
FastKV	3.07	1.84	0.49	0.36
ASL	2.27	0.98	0.29	0.26
GemFilter	2.36	1.60	0.28	0.14
ASL_2pass	1.57	0.70	0.18	0.09
Qwen2.5-7B, KV Budget = 2048.				
FastKV	3.43	2.19	0.57	0.29
ASL	2.35	1.54	0.40	0.21
GemFilter	3.32	2.20	0.41	0.14
ASL_2pass	1.82	1.82	0.29	0.11

Table 7: Memory usage (↓) comparison, averaged over 13 tasks, numbers reported in GB.

Models	Full KV	SnapKV	FastKV	ASL	GemFilter	ASL_2pass
InfiniteBench, average context length = 214k, KV budget = 2048.						
Llama-3.1-8B-UL	18.6	**0.3**	**0.3**	**0.3**	**0.3**	**0.3**
Qwen2.5-7B	8.6	**0.2**	**0.2**	**0.2**	**0.2**	**0.2**
RULER, context length = 128k, KV budget = 2048.						
Llama-3.1-8B-UL	17.1	**0.3**	**0.3**	**0.3**	**0.3**	**0.3**
Qwen2.5-7B	7.5	**0.2**	**0.2**	**0.2**	**0.2**	**0.2**

Table 8: Performance of varying KV budgets, Qwen2.5-7B, RULER, 128k. TTFT and TPOT are reported in ratio to Full KV (= 1). Memory usage is reported in GB.

Methods	Accuracy (↑)	TTFT (↓)	TPOT (↓)	Memory (↓)
KV Budget = 2048.				
FastKV	59.1	0.54	0.15	**0.2**
ASL	**66.4**	0.81	0.15	**0.2**
GemFilter	56.7	**0.53**	**0.11**	**0.2**
ASL_2pass	62.5	0.95	0.13	**0.2**
KV Budget = 8192.				
FastKV	67.0	**0.55**	0.18	**0.5**
ASL	**68.5**	0.94	0.18	0.6
GemFilter	59.9	**0.55**	**0.11**	**0.5**
ASL_2pass	66.5	0.96	**0.11**	**0.5**

for the RULER benchmark is small (27 for mean and 120 for 99th percentile). For real-world applications, in which the number of output tokens is often much larger, the gap between the throughputs of ASL and FastKV/GemFilter tends to be even smaller, because most processing time will be spent on the decoding stage and these methods report almost the same TPOT (Table 5).

Memory Usage. Table 7 reports the memory usage. All the KV cache reduction methods consume far less memory than full KV, and their memory usages are almost identical. This showcases that ASL's additional overhead in memory is negligible.

5.4 Varying KV Budget

Table 8 shows the performance under two KV budget settings: 2048 and 8192. By trading TTFT, ASL and ASL_2pass consistently exhibit higher accuracy than their counterparts FastKV and GemFilter for the two budget settings. Moreover, ASL methods report competitive TPOTs and consume around the same size of memory as others.

Table 9: Effect of relative variance threshold τ, RULER, 128k. TTFT is reported in ratio to Full KV (= 1).

Metrics	$\tau = 0.2$	$\tau = 0.3$	$\tau = 0.4$	$\tau = 0.5$	$\tau = 0.6$
Llama-3.1-8B-UL, KV budget = 2048.					
Accuracy (↑)	55.4	56.1	55.1	57.1	**57.9**
TTFT (↓)	0.93	0.79	0.66	0.60	**0.57**
Llama-3.1-8B-UL, KV Budget = Full (before sel.) & 2048 (after sel.)					
Accuracy (↑)	68.6	**69.2**	63.2	63.6	63.8
TTFT (↓)	0.92	0.78	0.66	0.61	**0.57**
Qwen2.5-7B, KV budget = 2048.					
Accuracy (↑)	64.9	66.5	**66.9**	65.8	62.1
TTFT (↓)	0.90	0.83	0.78	0.75	**0.73**
Qwen2.5-7B, KV Budget = Full (before sel.) & 2048 (after sel.)					
Accuracy (↑)	**78.2**	74.2	70.5	67.5	63.3
TTFT (↓)	0.92	0.81	0.76	0.71	**0.67**

5.5 Effect of Relative Variance Threshold

We vary the relative variance threshold τ from 0.2 to 0.6, and report the results on RULER, 128k in Table 9. The accuracy for different tasks is reported in Appendix D.3. TPOT and memory usage are barely affected by τ and not reported. The accuracy fluctuates for Llama-3.1-8B-UL when τ varies from 0.2 to 0.6. For Qwen2.5-7B, under the KV budget of 2048, the accuracy increases and then decreases, and when full KV is available before token selection, a generally decreasing trend of accuracy is observed. TTFT consistently decreases under larger thresholds, because relative variance decreases as layers, meeting a larger threshold first. Seeing the results, we suggest using $\tau = 0.3$ for the trade-off between accuracy and TTFT.

6 Conclusion

We proposed ASL, a KV cache reduction method that adaptively chooses the selection layer for layer-wise token pruning in various tasks. Observing attention patterns across tasks, we exploited the variance of token ranks ordered by attention score. ASL works in a one-shot selection manner, selecting tokens at a layer and propagating only those tokens to deeper layers. To meet the KV budget requirement, we jointly used ASL with existing KV cache reduction methods SnapKV and GemFilter. We evaluated ASL on three benchmarks. The results demonstrated its accuracy-efficiency trade-offs compared to state-of-the-art layer-wise token pruning methods.

Limitations

In this work, we focus on applying ASL to one-shot methods such as FastKV and GemFilter. There are other layer-wise token pruning methods, such as LasyLLM and OmniKV, as summarized in Table 1. These methods select tokens multiple times, and the way of integrating ASL into these methods is yet to be investigated. One possible solution is to use multiple variance thresholds, or a threshold with a decay factor.

There are also numerous methods for KV cache reduction that do not belong to the category of layer-wise token pruning, as noted in Section 2. A more comprehensive comparison with those methods may better reveal the positioning of this work.

Another limitation is that only two LLMs, Llama 3.1-8B-UL and Qwen2.5-7B, have been evaluated for the proposed method. Less competitiveness of ASL, when integrated with GemFilter on Llama 3.1-8B-UL, has been observed.

Ethical Considerations

In this work, we study reducing memory footprint and accelerating LLM inference. To the best of our knowledge, there is no negative societal impact in this research.

The benchmarks used in the experiments are public and have been used in many existing studies on this topic. The materials do not contain personally identifying information or offensive content. We comply with the terms of using the benchmarks.

We used AI assistants to polish the writing of the paper. We are responsible for all the materials presented in this work.

Acknowledgements

This work is supported by JSPS Kakenhi JP23K17456, JP23K25157, JP23K28096, and JP25H01117.

References

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, and 1 others. 2024. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*.

Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388.

Longze Chen. 2024. Awesome-kv-cache-compression. https://github.com/October2001/Awesome-KV-Cache-Compression.

Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, and 1 others. 2024. MagicPig: LSH sampling for efficient LLM generation. *arXiv preprint arXiv:2410.16179*.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.

Huggging Face. 2025. Transformers – attention interface. https://huggingface.co/docs/transformers/en/attention_interface.

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*.

Qichen Fu, Minsik Cho, Thomas Merth, Sachin Mehta, Mohammad Rastegari, and Mahyar Najibi. 2024a. LazyLLM: Dynamic token pruning for efficient long context LLM inference. *arXiv preprint arXiv:2407.14057*.

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

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2023. Model tells you what to discard: Adaptive KV cache compression for LLMs. *arXiv preprint arXiv:2310.01801*.

Andrey Gromov, Kushal Tirumala, Hassan Shapourian, Paolo Glorioso, and Daniel A Roberts. 2024. The unreasonable ineffectiveness of the deeper layers. *arXiv preprint arXiv:2403.17887*.

Jitai Hao, Yuke Zhu, Tian Wang, Jun Yu, Xin Xin, Bo Zheng, Zhaochun Ren, and Sheng Guo. 2025. OmniKV: Dynamic context selection for efficient long-context LLMs. In *The Thirteenth International Conference on Learning Representations*.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Sophia 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*, 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. 2024a. Minference. https://github.com/microsoft/MInference/tree/main.

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. 2024b. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. *Advances in Neural Information Processing Systems*, 37:52481–52515.

Weisheng Jin, Maojia Song, Tej Deep Pala, Yew Ken Chia, Amir Zadeh, Chuan Li, and Soujanya Poria. 2025. PromptDistill: Query-based selective token retention in intermediate layers for efficient large language model inference. *arXiv preprint arXiv:2503.23274*.

Dongwon Jo, Jiwon Song, Yulhwa Kim, and Jae-Joon Kim. 2025. FastKV: KV cache compression for fast long-context processing with token-selective propagation. *arXiv preprint arXiv:2502.01068*.

Greg Kamradt. 2024. Needle in a haystack – pressure testing LLMs. https://github.com/gkamradt/LLMTest_NeedleInAHaystack.

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. *Advances in Neural Information Processing Systems*, 37:22947–22970.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Reza Haffari, and Bohan Zhuang. 2024a. MiniCache: KV cache compression in depth dimension for large language models. *Advances in Neural Information Processing Systems*, 37:139997–140031.

Songwei Liu, Chao Zeng, Lianqiang Li, Chenqian Yan, Lean Fu, Xing Mei, and Fangmin Chen. 2024b. FoldGPT: Simple and effective large language model compression scheme. *arXiv preprint arXiv:2407.00928*.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024c. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. *arXiv preprint arXiv:2402.02750*.

Lingkun Long, Rubing Yang, Yushi Huang, Desheng Hui, Ao Zhou, and Jianlei Yang. 2025. SlimInfer: Accelerating long-context LLM inference via dynamic token pruning. *arXiv preprint arXiv:2508.06447*.

Aurick Qiao, Zhewei Yao, Samyam Rajbhandari, and Yuxiong He. 2024. SwiftKV: Fast prefill-optimized inference with knowledge-preserving model transformation. *arXiv preprint arXiv:2410.03960*.

Ziran Qin, Yuchen Cao, Mingbao Lin, Wen Hu, Shixuan Fan, Ke Cheng, Weiyao Lin, and Jianguo Li. 2025. CAKE: Cascading and adaptive KV cache eviction with layer preferences. *arXiv preprint arXiv:2503.12491*.

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*.

Prajwal Singhania, Siddharth Singh, Shwai He, Soheil Feizi, and Abhinav Bhatele. 2024. Loki: Low-rank keys for efficient sparse attention. *Advances in Neural Information Processing Systems*, 37:16692–16723.

Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. 2024. ShadowKV: KV cache in shadows for high-throughput long-context LLM inference. *arXiv preprint arXiv:2410.21465*.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. Quest: Query-aware sparsity for efficient long-context LLM inference. *arXiv preprint arXiv:2406.10774*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Zihao Wang, Bin Cui, and Shaoduo Gan. 2024. SqueezeAttention: 2d management of KV-cache in LLM inference via layer-wise optimal budget. *arXiv preprint arXiv:2404.04793*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, and 3 others. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online.

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. 2024. PyramidInfer: Pyramid KV cache compression for high-throughput LLM inference. *arXiv preprint arXiv:2405.12532*.

Bohan Yu and Yekun Chai. 2025. EvolKV: Evolutionary KV cache compression for LLM inference. *arXiv preprint arXiv:2509.08315*.

Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Hao, Xu Han, Zhen Thai, Shuo Wang, Zhiyuan Liu, and 1 others. 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.

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*, 36:34661–34710.

Xiabin Zhou, Wenbin Wang, Minyan Zeng, Jiaxian Guo, Xuebo Liu, Li Shen, Min Zhang, and Liang Ding. 2024. DynamicKV: Task-aware adaptive KV cache compression for long context LLMs. *arXiv preprint arXiv:2412.14838*.

Appendix

A Algorithm Pseudo-Codes

Algorithm 1 calculates pooled attention scores over the most recent tokens (specified by a window) and uses them to identify the top-k indices for each GQA head group. In parallel, it computes the score ranks of the pooled attention scores and stores them in a rank cache. Compressed KV entries are returned, in line with SnapKV (Li et al., 2024).

Based on the rank cache, Algorithm 2 obtains the unique top-k tokens across the most recent L_{obs} layers, and then computes the relative variance of their ranks within these L_{obs} layers. If the relative variance falls below a threshold τ, the current layer is determined as the selection layer, and only the selected tokens are propagated to subsequent layers. The rank cache is cleared afterwards.

Algorithm 1: GetRanks

Input: $Q, K, V, layer\_idx, W, S, KV\_budget, rank\_cache$
// W: window_size, S: kernel_size
Output: compressed KV, update $rank\_cache[layer\_idx]$

$N_q \leftarrow Len(Q)$, $B \leftarrow BatchSize$, $H_{q,k} \leftarrow HeadSize$, $G \leftarrow H_q/H_k$; // GQA groups
$I_w \leftarrow \{N_q - W, \ldots, N_q - 1\}$; // window token indices
$A \leftarrow \text{Softmax}\left(\frac{Q_{:,:,N_q-W:N_q} K^\top}{\sqrt{d}} + \text{CausalMaskOnLastBlock}(W)\right)$; // $A \in \mathbb{R}^{B \times H_q \times W \times N_q}$
$U \leftarrow \sum_{t=1}^{W} A_{:,:,t,0:N_q-W}$; // past-only histogram, $U \in \mathbb{R}^{B \times H_q \times (N_q - W)}$
$V \leftarrow \text{AvgPool1D}(U, \text{kernel} = S, \text{stride} = 1, \text{pad} = \lfloor S/2 \rfloor)$;
$P \leftarrow \text{SumGroups}\big(\text{Reshape}(V, [B, H_k, G, N_q - W]), \text{ over } G\big)$; // $P \in \mathbb{R}^{B \times H_k \times (N_q - W)}$
$k \leftarrow \min(KV\_budget, N_q) - W$;
$I \leftarrow \text{Concat}(\text{TopK}(P, k),\ I_w)$; // per (B, H_k), add window tokens
$rank\_cache[layer\_idx] \leftarrow \text{RankDesc}(P)$;
return $K[I]$,$V[I]$;

Algorithm 2: SelectTokens

Input: $layer\_idx, KV\_budget, W, L_{\min}, L_{\text{obs}}, \tau, rank\_cache, init\_var$
// W: window_size
Output: $select\_idx$ (or None), updated $init\_var$

if $layer\_idx < L_{\min}$ **then**
  return None;
$T \leftarrow \text{Len}(rank\_cache[layer\_idx])$; // sequence length
$R \leftarrow \text{Stack}\big(\{rank\_cache[l]\}_{l=layer\_idx-L_{\text{obs}}+1}^{layer\_idx}\big)$; // $R \in \mathbb{R}^{L_{\text{obs}} \times T}$, collect and stack latest L_{obs} layer's rank from rank_cache
$k \leftarrow \min(KV\_budget - W, T)$;
$J \leftarrow \text{TopKIndices}(R, k, \text{smallest})$; // per row, $J \in \mathbb{N}^{L_{\text{obs}} \times k}$
$U \leftarrow \text{Unique}(\text{Flatten}(J))$;
$(var, init\_var) \leftarrow \text{RelNormVar}(R_{:,U}, init\_var)$; // $var = \frac{\mathbb{E}[\text{Var}_{\text{layer}}(R_{:,U})]}{init\_var}$, $init\_var$ is initialized as $\text{Var}_{\text{layer}}(R_{:,U})$ if $init\_var$=None
if $var < \tau$ **then**
  $I_w \leftarrow \{T, \ldots, T + W - 1\}$; // ensure window tokens
  $select\_idx \leftarrow \text{Unique}(J_{last} \cup I_w)$;
  $rank\_cache \leftarrow \emptyset$; // release cache
  return $select\_idx$;
else
  return None;

B Theoretical Analysis

To analyze the performance of ASL, we consider a model with L layers, h KV heads, head dimension d, and context length n. Next, we analyze TTFT (prefilling time), TPOT (decoding time), and memory usage.

B.1 TTFT

For full KV, TTFT is

$$T_{\text{full}} = L \cdot T_{\text{attn}}(n)$$

where $T_{\text{attn}}(n)$ denotes the attention processing time per layer.

For ASL, let L_{select} denote the selection layer.

From layers 0 to L_{select}, the costs are (1) Attention: $(L_{\text{select}}+1) \cdot T_{\text{attn}}(n)$; (2) Pooling: $(L_{\text{select}} - L_{\text{min}}) \cdot O(n)$; (3) Rank computation: $(L_{\text{select}} - L_{\text{min}}) \cdot O(n \log n)$; (4) Variance calculation: $(L_{\text{select}} - L_{\text{min}}) \cdot O(L_{\text{obs}} \cdot m)$, where m denotes the union size and $m \leq k \cdot L_{\text{obs}}$.

At layer L_{select}, the cost is top-k selection: $O(n \log k)$, or $O(n)$ with QuickSelect.

From layers $L_{\text{select}}+1$ to L, the cost is attention with selected tokens: $(L - L_{\text{select}} - 1) \cdot T_{\text{attn}}(k)$.

Therefore, ASL's TTFT is

$$T_{\text{ASL}} = (L_{\text{select}}+1)\cdot T_{\text{attn}}(n)+(L-L_{\text{select}}-1)\cdot T_{\text{attn}}(k)+(L_{\text{select}}-L_{\text{min}})\cdot O(n \log n+L_{\text{obs}}\cdot m)+O(n \log k).$$

Suppose $T_{\text{attn}}(n) = O(n^2 \cdot d + n \cdot d^2)$ for FlashAttention (Dao, 2023). Because $T_{\text{attn}}(n) \gg O(n \log n)$ for large n (e.g., 128k), the third and fourth terms of T_{ASL} (i.e., pooling, rank, variance calculation, and top-k selection) are negligible.

ASL's TTFT ratio to full KV is

$$\frac{T_{\text{ASL}}}{T_{\text{full}}} \approx \frac{L_{\text{select}}+1}{L} + \frac{L - L_{\text{select}} - 1}{L} \cdot \frac{T_{\text{attn}}(k)}{T_{\text{attn}}(n)}.$$

Suppose $k = 2048$ and $n = 131072$ (128k context). $L = 32$ for Llama-3.1-8B-UL and 28 for Qwen2.5-7B.

Under this setting, the mean L_{select} is 23.9 for Llama-3.1-8B-UL) and 21.6 for Qwen2.5-7B (Appendix D.4 reports the the distributions of L_{select}).

Therefore, $\frac{T_{\text{ASL}}}{T_{\text{full}}} \approx 0.78$ (Llama-3.1-8B-UL) and 0.81 (Qwen2.5-7B).

As for comparison, the empirical ratio in Table 4 is 0.79 for Llama-3.1-8B-UL and 0.81 for Qwen2.5-7B, showcasing that the above theoretical prediction well matches the empirical result.

As for comparison, in FastKV, $L_{\text{select}} = 15$ for Llama-3.1-8B-UL and 14 for Qwen2.5-7B. Therefore, $\frac{T_{\text{FastKV}}}{T_{\text{full}}} \approx 0.5$ and 0.54, respectively. Empirical values in Table 4 are 0.5 and 0.54, respectively, exactly matching the predicted values.

B.2 TPOT

For full KV, TPOT is

$$T_{\text{full}} = L \cdot O(n \cdot d).$$

For ASL, TPOT is

$$T_{\text{ASL}} = L \cdot O(k \cdot d).$$

ASL's TPOT ratio to full KV is

$$\frac{T_{\text{ASL}}}{T_{\text{full}}} = \frac{k}{n}.$$

When $k = 2048$ and $n = 131072$ (128k context), $\frac{T_{\text{ASL}}}{T_{\text{full}}} \approx 0.016$.

FastKV shares the same predication as above.

The empirical ratio of ASL to full KV in Table 5 is 0.028 for Llama-3.1-8B-UL and 0.015 for Qwen2.5-7B. For Qwen2.5-7B, it roughly matches the prediction. For Llama-3.1-8B-UL, we suspect that the discrepancy is due to its model-specific implementation (an ultra long variant by NVidia).

B.3 Memory Usage

For full KV, memory usage is

$$M_{\text{full}} = 2 \cdot L \cdot h \cdot d \cdot n$$

where 2 accounts for key and value.

ASL introduces two types of memory overhead: (1) Pooled attention scores: $M_{\text{pool}} = L_{\text{obs}} \cdot \frac{n}{w}$, where w is the pooling kernel size; (2) Rank cache: $M_{\text{rank}} = L_{\text{obs}} \cdot \frac{n}{w}$.

The total additional overhead is

$$M_{\text{ASL-OV}} = 2 \cdot L_{\text{obs}} \cdot \frac{n}{w}.$$

ASL's KV cache size under budget k is

$$M_{\text{ASL-KV}} = 2 \cdot L \cdot h \cdot d \cdot k.$$

ASL's memory usage is

$$M_{\text{ASL}} = M_{\text{ASL-KV}} + M_{\text{ASL-OV}} = 2 \cdot L \cdot h \cdot d \cdot k + 2 \cdot L_{\text{obs}} \cdot \frac{n}{w}.$$

For Llama-3.1-8B-UL, $L = 32$, $h = 8$. For Qwen2.5-7B, $L = 28$, $h = 4$. $d = 128$, $w = 7$, $L_{\text{obs}} = 8$.

Suppose $k = 2048$ and $n = 131072$ (128k context).

Therefore, $\frac{M_{\text{ASL}}}{M_{\text{full}}} \approx 0.016$ for both LLMs.

FastKV's memory usage equals to $M_{\text{ASL-KV}}$, thereby yielding approximately the same prediction as above.

Table 7 shows that the empirical ratio is 0.018 for Llama-3.1-8B-UL and 0.027 for Qwen2.5-7B. The discrepancy can be attributed to additional memory consumption such as workspace for FlashAttention, PyTorch memory management, and activations for FFN.

C Experimental Setup Details

C.1 Datasets

InfiniteBench. (Zhang et al., 2024), is a benchmark testing LLMs in various aspects of long-context processing. We use the version provided in the MInference GitHub repository (Jiang et al., 2024a). There are a a total of 3,992 examples, with an average context length of 214k, evaluating the following 10 tasks: (1) summarization of a fake book created with core entity substitution (En.Sum a.k.a. longbook_sum_eng), (2) free-form question answering based on the fake book (En.QA a.k.a. longbook_qa_eng), (3) multiple choice questions derived from the fake book (En.MC a.k.a. longbook_choice_eng), (4) identification of talkers in partially anonymized scripts (En.Dia a.k.a. longdialogue_qa_eng), (5) question answering on a set of Chinese books (Zh.QA a.k.a. longbook_qa_chn), (6) finding which function in a code repo contains an crashing error (Code.Debug a.k.a. code_debug), (7) finding special integers in a lengthy list (Math.Find a.k.a. math_find), (8) retrieving hidden keys in a noisy long context (Retr.PassKey a.k.a. passkey), (9) locating repeated hidden numbers in a noisy long context (Retr.Num a.k.a. number_string), and (10) finding the corresponding value from a dictionary and a key (Retr.KV a.k.a. kv_retrieval),

RULER. (Hsieh et al., 2024) is a benchmark of synthetic examples for evaluating long-context LLMs with configurable sequence length and task complexity. It consists of 13 tasks, including (1) identifying common words from a mixture of common and uncommon words (cwe), (2) identifying most frequent words from

a Zeta distribution (fwe), (3) single-key NIAH, where a single key-value pair is inserted into noisy text, with varying difficulties (niah_single_1, niah_single_2, and niah_single_3 a.k.a. S-NIAH1, S-NIAH2, and S-NIAH3, respectively), (4) multi-key NIAH, where multiple keys are inserted and a specific value among hard distractors needs to be retrieved, with varying difficulties (niah_multikey_1, niah_multikey_2, and niah_multikey_3 a.k.a. MK-NIAH1, MK-NIAH2, and MK-NIAH3, respectively), (5) retrieving values for multiple keys (niah_multiquery a.k.a. MQ-NIAH), (6) retrieving all values associated with a single key (niah_multivalue a.k.a. MV-NIAH), (7) question answering with distracting paragraphs inserted, with varying difficulties (qa_1 and qa_2 a.k.a. QA1 and QA2, respectively), (8) tracing all variable names pointing to the same value through variable bindings (vt). We test models on 4k, 8k, 16k, 32k, 64k, and 128k context lengths, including 2,600 examples per length.

NIAH. (Kamradt, 2024) is simple needle-in-a-haystack analysis to test in-context retrieval ability of long-context LLMs. The model needs to identify a random fact or statement (the "needle") from a long-context window (the "haystack"). The evaluation iterates over document depths (where the needle is placed) and context lengths. We scale the test from 1k to 256k.

C.2 Methods

We compare our method, ASL, with three layer-wise token pruning methods:

- GemFilter (Shi et al., 2024), a one-shot method using a two-pass strategy for token selection. In the first pass, it calculates full attention for the first L layers, finding the top-k tokens at the L-th layer; in the second pass, it calculates attention for the selected token set for all layers.
- FastKV (Jo et al., 2025), a one-shot method similar to GemFilter but having only one pass. Before reaching the selection layer, full attention is calculated for prefilling but only top-k tokens are retained in the KV cache for decoding. From the selection layer, only top-k selected tokens are carried forward to deeper layers for both prefilling and decoding.
- PyramidInfer (Yang et al., 2024), a progressive method that performs top-p selection and adopts a decay ratio to determine the value of p at each layer.

The selection layer for FastKV is 15 for Llama-3.1-8B-UL and 14 for Qwen2.5-7B. The selection layer for GemFilter is 13 for Llama-3.1-8B-UL and 14 for Qwen2.5-7B. For 1D average pooling in SnapKV and ASL, window size = 32 and kernel size = 7.

LazyLLM (Fu et al., 2024a) and SlimInfer (Long et al., 2025), another two layer-wise token pruning that optimize prefilling, are excluded due to unavailable source codes. MInference (Jiang et al., 2024b), despite optimizing TTFT, is also excluded because it is orthogonal to our method and its block-sparse kernel does not reduce KV cache.

C.3 Environments

The experiments are conducted on an Nvidia H100 GPU with 80GB memory. We run the LLMs with the Hugging Face Transformers library (Wolf et al., 2020). All the methods use FlashAttention-2 (Dao, 2023), except PyramidInfer, which uses eager attention (Face, 2025).

D Additional Experiments

D.1 TTFT and TPOT on InfiniteBench

Tables 10 and 11 report the TTFT and TPOT on the 10 tasks of InfiniteBench. Compared to the results on RULER (Tables 4 and 5), the observations are similar. ASL and ASL_2pass trade prefilling time for higher accuracy, while the decoding speed is approximately the same as other layer-wise methods.

D.2 Effect of $L_{\min}$ and L_{obs}

To study the effect of observation start layer number $L_{\min}$ and lookback layer number L_{obs}, we plot the relative variance under different settings of $L_{\min}$ and L_{obs} in Figure 8.

$L_{\min}$. From Figure 8, we observe that the relative variance generally exhibits a decreasing trend under smaller values of $L_{\min}$. In contrast, the relative variance fluctuates more violently and even rebounds

Table 10: TTFT (↓) comparison on InfiniteBench. Ratio to Full KV (= 1) is reported.

Methods	En.Sum	En.QA	En.MC	En.Dia	Zh.QA	Code.Debug	Math.Find	Retr.PassKey	Retr.Num	Retr.KV	Avg.
Llama-3.1-8B-UL, KV Budget = 2048.											
SnapKV	1.00	1.01	0.99	0.99	0.99	1.01	1.01	1.02	1.01	1.01	1.00
FastKV	0.50	0.50	0.51	0.50	0.51	0.50	0.50	0.52	0.50	0.50	0.50
ASL	0.58	0.73	0.76	0.71	7.70	0.67	0.67	0.47	0.60	0.68	0.68
GemFilter	0.44	0.44	0.44	0.43	0.44	0.45	0.44	0.45	0.44	0.44	**0.44**
ASL_2pass	0.58	0.74	0.77	0.71	0.71	0.69	0.67	0.48	0.62	0.68	0.69
Qwen2.5-7B, KV Budget = 2048.											
SnapKV	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
FastKV	0.54	0.53	0.54	0.53	0.54	0.54	0.54	0.54	0.53	0.53	**0.53**
ASL	0.80	0.82	0.90	0.85	0.85	0.82	0.73	0.77	0.70	0.73	0.82
GemFilter	0.54	0.53	0.53	0.53	0.53	0.53	0.53	0.54	0.52	0.53	**0.53**
ASL_2pass	0.80	0.83	0.90	0.85	0.84	0.83	0.72	0.76	0.70	0.73	0.82

Table 11: TPOT (↓) comparison on InfiniteBench. Ratio to Full KV (= 1) is reported.

Methods	En.Sum	En.QA	En.MC	En.Dia	Zh.QA	Code.Debug	Math.Find	Retr.PassKey	Retr.Num	Retr.KV	Avg.
Llama-3.1-8B-UL, KV Budget = 2048.											
SnapKV	0.31	0.35	0.26	0.20	0.26	0.29	0.35	0.34	0.30	0.29	0.29
FastKV	0.27	0.31	0.23	0.17	0.23	0.25	0.32	0.29	0.27	0.25	0.25
ASL	0.29	0.29	0.24	0.18	0.23	0.25	0.32	0.29	0.26	0.25	0.25
GemFilter	0.32	0.30	0.21	0.17	0.23	0.24	0.25	0.22	0.25	0.21	**0.23**
ASL_2pass	0.32	0.29	0.20	0.17	0.23	0.24	0.24	0.23	0.24	0.21	**0.23**
Qwen2.5-7B, KV Budget = 2048.											
SnapKV	0.16	0.12	0.12	0.09	0.12	0.13	0.18	0.15	0.15	0.15	0.13
FastKV	0.16	0.12	0.12	0.09	0.12	0.13	0.18	0.15	0.15	0.15	0.13
ASL	0.16	0.12	0.12	0.09	0.12	0.12	0.18	0.15	0.15	0.15	0.13
GemFilter	0.15	0.09	0.09	0.07	0.09	0.09	0.13	0.07	0.11	0.11	**0.10**
ASL_2pass	0.14	0.09	0.09	0.07	0.09	0.09	0.14	0.06	0.11	0.12	**0.10**

under larger $L_{\min}$ values. In order to find a threshold to determine the selection layer, we choose to use $L_{\min} = \lfloor L_{\text{model}}/3 \rfloor$ in ASL, where L_{model} is the number of the layers in the LLM, e.g., $\lfloor 32/3 \rfloor = 10$ for Llama-3.1-8B-UL and $\lfloor 28/3 \rfloor = 9$ for Qwen2.5-7B. This is to strike a balance between the identification of the steepest decline in variance and the additional overhead (because we only start the operation of ASL when reaching layer $L_{\min}$).

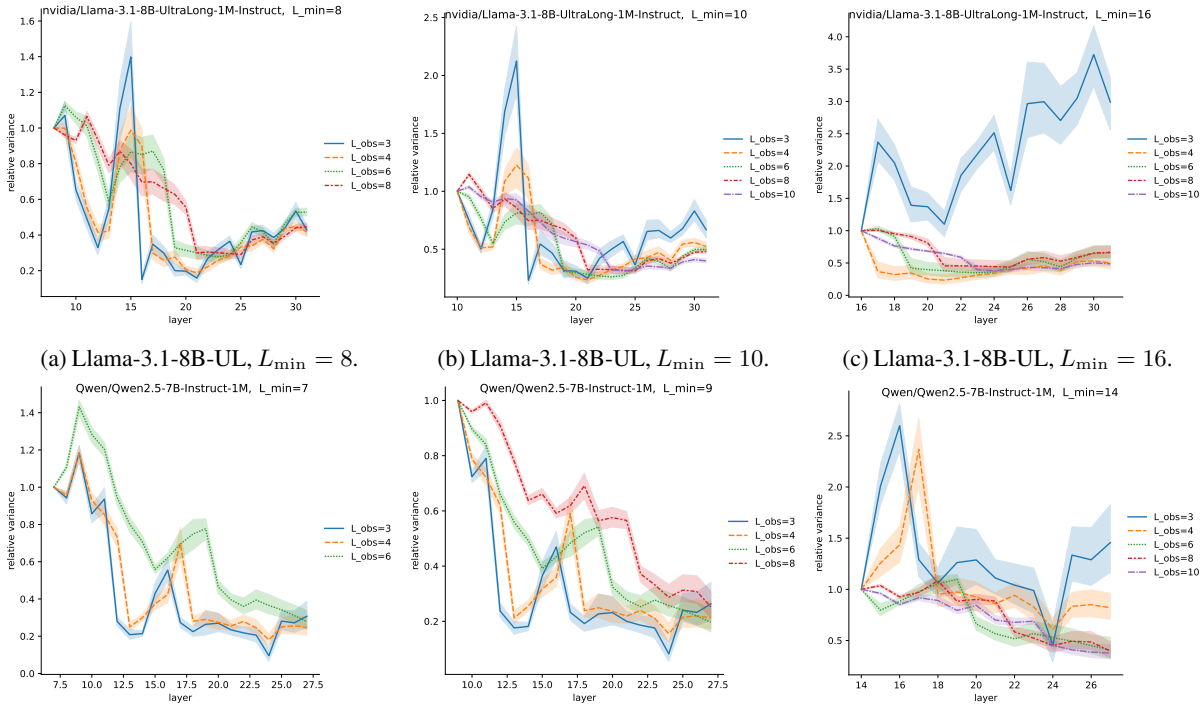
L_{obs}. As shown in Figure 8, L_{obs} represents the sensitivity of the relative variance to attention scores, which change across layers. A smaller L_{obs} suggests that the variance is more responsive to the change of layers. Seeing this, we choose L_{obs} to be 8 control the sensitivity and effectiveness (i.e., the decreasing trend can be identified, so we can use a threshold to determine the selection layer).

D.3 Effect of Relative Variance Threshold

We report the effect of τ across the 13 tasks of RULER in Table 12, where accuracy is measured at a context length of 128k. When the KV Budget is set to 2048, the general trend is that accuracy varies only slightly across different τ settings for all 13 tasks, except for outliers observed on niah_single_3 with Qwen2.5-7B at the smallest ($\tau = 0.2$) and largest ($\tau = 0.6$) values. In contrast, when KV compression methods are not applied (i.e., when the KV Budget is full before the selection layer), the accuracy on tasks such as niah_single_3 and niah_multikeys_3 are highly sensitive to τ. For example, with Qwen2.5-7B on niah_multikeys, the accuracy exceeds 70 at $\tau = 0.3$ but drops sharply to only 3.0 at $\tau = 0.6$.

D.4 Distributions of Selection Layer

Figures 9 and 10 show the frequency of a layer determined as the selection layer RULER, 128k context, and a KV budget of 2048. Each task in RULER consists of 200 questions. Compared to FastKV, which uses fixed layer selection, we observe that in tasks where FastKV reports lower scores, ASL selects a later layer than FastKV, while in tasks where FastKV reports higher scores, ASL picks a selection layer close to FastKV's.



(a) Llama-3.1-8B-UL, $L_{\min} = 8$. (b) Llama-3.1-8B-UL, $L_{\min} = 10$. (c) Llama-3.1-8B-UL, $L_{\min} = 16$.

(d) Qwen2.5-7B, $L_{\min} = 7$. (e) Qwen2.5-7B, $L_{\min} = 9$. (f) Qwen2.5-7B, $L_{\min} = 14$.

Figure 8: Effect of $L_{\min}$ and L_{obs} on relative variance, KV retrieval task.

Table 12: Effect of relative variance threshold τ on accuracy (↑), RULER, 128k.

τ	S-NIAH1	S-NIAH2	S-NIAH3	MK-NIAH1	MK-NIAH2	MK-NIAH3	MV-NIAH	MQ-NIAH	VT	CWE	FWE	QA1	QA2	Avg.
Llama-3.1-8B-UL, KV Budget = 2048.														
0.2	100.0	98.0	63.0	89.5	15.0	0.5	46.6	74.4	75.2	0.15	67.2	58.0	34.5	55.5
0.3	100.0	98.0	44.0	88.5	15.0	0.5	72.6	73.9	75.3	0.05	67.0	60.5	34.5	56.1
0.4	100.0	95.0	45.0	87.5	7.5	0.5	74.9	71.9	75.6	0.05	65.7	58.0	34.0	55.0
0.5	100.0	96.0	64.5	90.0	10.0	0.5	76.9	70.8	75.4	0.05	65.2	58.5	35.0	57.1
0.6	100.0	96.0	71.0	89.0	10.5	0.5	76.6	75.0	76.1	0.05	65.2	58.5	34.0	57.9
Llama-3.1-8B-UL, KV Budget = Full (before selection) & 2048 (after selection).														
0.2	100.0	98.5	99.0	90.0	68.5	15.0	67.6	87.6	83.4	0.05	90.5	56.0	35.0	68.6
0.3	100.0	98.5	90.0	89.5	68.0	14.5	87.5	87.4	83.1	0.05	87.3	60.0	33.5	69.2
0.4	100.0	95.5	63.0	88.5	31.5	11.5	91.5	83.6	83.0	0.05	82.0	58.0	33.0	63.2
0.5	100.0	97.0	72.5	90.0	26.5	10.5	90.9	83.0	83.1	0.05	81.0	58.0	34.0	63.6
0.6	100.0	96.5	76.0	89.0	29.5	8.0	91.3	85.4	82.7	0.05	79.3	58.5	33.5	63.8
Qwen2.5-7B, KV Budget = 2048.														
0.2	100.0	99.5	16.5	99.5	85.0	0.0	82.9	98.9	89.1	10.1	56.5	61.0	44.7	64.9
0.3	100.0	99.5	33.5	99.5	85.0	1.5	83.4	98.9	88.1	11.1	60.7	58.0	44.7	66.4
0.4	100.0	99.5	36.0	100.0	85.0	1.0	83.0	99.0	88.0	11.4	61.2	61.0	44.7	66.9
0.5	100.0	97.5	23.5	99.0	85.5	1.0	79.9	98.8	88.2	11.4	62.5	62.0	45.5	65.7
0.6	100.0	88.5	7.5	92.5	86.0	1.0	70.0	97.5	88.5	11.4	62.2	59.0	43.2	62.1
Qwen2.5-7B, KV Budget = Full (before selection) & 2048 (after selection).														
0.2	100.0	99.5	84.5	99.5	96.5	87.0	85.5	99.6	83.1	9.3	62.3	61.5	47.7	78.2
0.3	100.0	99.5	48.0	99.5	96.5	72.5	85.5	99.6	81.9	11.5	62.3	59.3	48.2	74.2
0.4	100.0	99.5	43.0	100.0	96.5	26.5	86.0	99.6	81.6	12.1	62.7	62.3	46.2	70.5
0.5	100.0	97.0	27.5	99.5	96.5	9.0	83.0	99.6	82.7	12.1	62.5	62.0	45.5	67.5
0.6	100.0	88.5	8.5	93.5	97.0	3.0	74.3	98.3	84.2	12.0	61.3	59.3	43.2	63.3

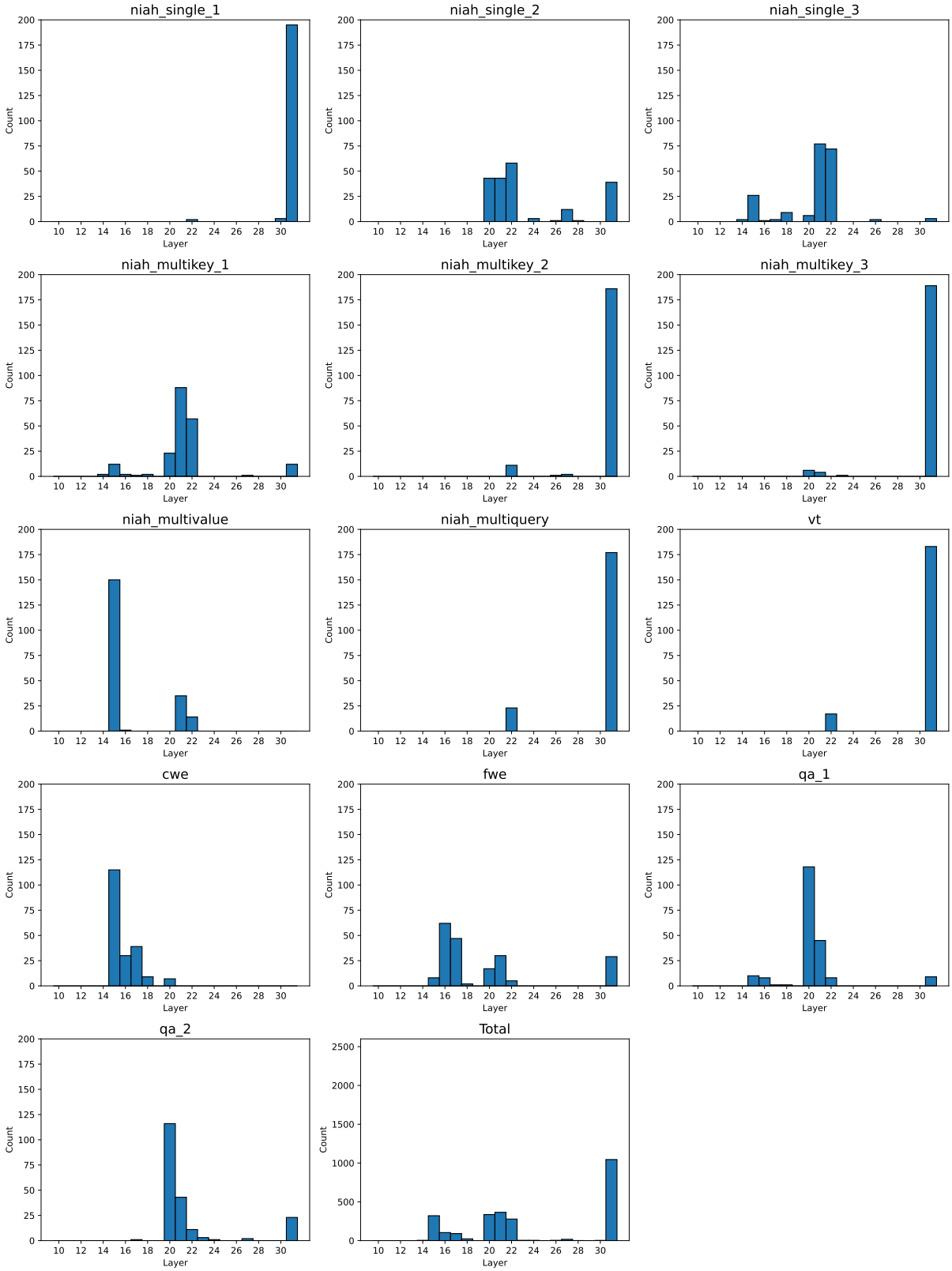


Figure 9: Distributions of selection layer across tasks, Llama-3.1-8B-UL, RULER, 128k, $\tau = 0.3$.

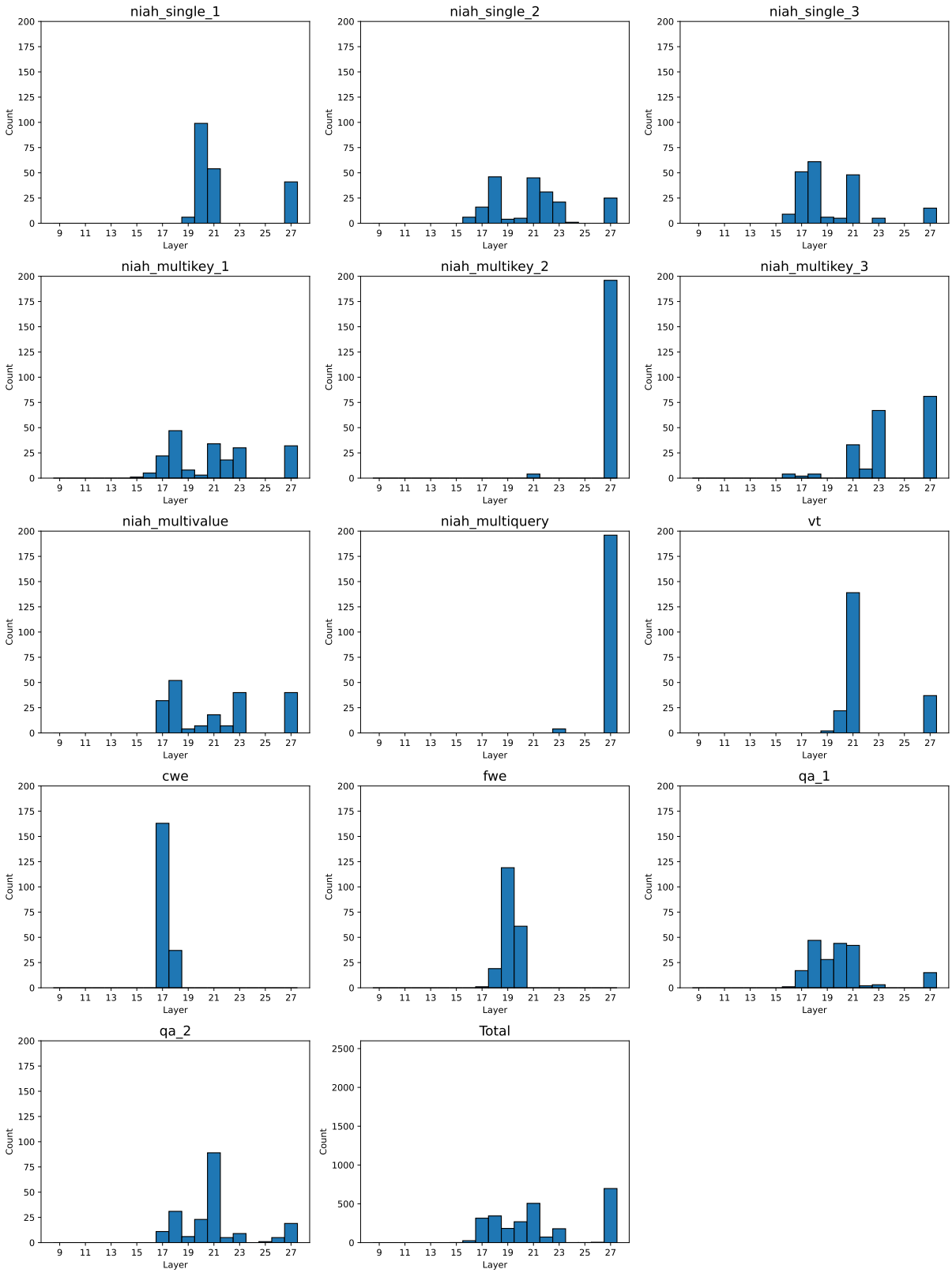


Figure 10: Distributions of selection layer across tasks, Qwen2.5-7B, RULER, 128k, $\tau = 0.3$.