LagKV: Lag-Relative Information of the KV Cache Tells Which Tokens Are Important

Anonymous EMNLP submission

Abstract

The increasing size of the Key-Value (KV) cache during the Large Language Models long-context inference is the main obstacle for its balance between the deployment cost and task accuracy. To reduce the KV cache size in such scenarios, most previous efforts leveraged on the attention weight to evict non-critical cache tokens. But there is a trade-off in those methods, they usually require major modifiation of the inference infrastructure and significant computation overhead. Base on the fact that the Large Lanuage models are autoregresssive models, we propose *LagKV*, a KV compression strategy only relying on straight forward comparison among KV themself. It is a totally attention free method which offers easy integration to the main stream inference platform and comparable performance comparing to other complicated KV compression methods. Results on LongBench and PasskeyRetrieval show that, our approach achieves nearly zero loss when the ratio is $2\times$ and $\approx 90\%$ of the original model performance for $8\times$. Especially in the 64-digit passkey retrieval task, our mehod outperforms the attention weight based method H_2O over 50% with same compression ratios.

1 Introduction

Large Language Models (LLMs) have recently demonstrated remarkable success across diverse text processing tasks, including document retrieval (Laban et al., 2023), code generation (Gu, 2023), and mathematical reasoning (like R1 model (DeepSeek-AI et al., 2025)). The Scaling law (Kaplan et al., 2020) suggests that larger models generally achieve superior performance. The R1-like models further indicates that longer generation sequences with additional 'thinking tokens' can enhance reasoning capabilities. However, these improvements comes at a significant cost: the growing KV cache size poses a major challenge for efficient LLM inference. Many efforts try to mitigate this challenge.

Most of LLMs are totally relying on Self-Attention mechanism (Vaswani et al., 2023) to determine which historical tokens are important in the next token prediction. Therefore, many KV compression approaches are based on it to drop unimportant ones (Zhang et al., 2024; Liu et al., 2024b; Li et al., 2024; NVIDIA, 2024). This kind of algorithms keeps a remarkable performance even when the compression ratio is high. However, most of these importance-based token-dropping approaches depend on the ending query question (Instruction Dependence) to achieve such a performance (Li et al., 2025; Feng et al., 2024; Tang et al., 2024). And aslo, they are incompatible with Flash Attention(FA) (Dao, 2023) because they require computing attention weights to determine token importance. This limitation makes them impractical for deployment, given FA's critical role in long-context inference.

Another prominent direction in KV cache optimization involves quantization techniques (Yang et al., 2024; Liu et al., 2024c), which aim to compress the memory footprint of KV states by representing them with reduced precision. These methods achieve significant memory savings—often by 4× or more—while preserving model performance through careful error mitigation strategies. Beyond memory efficiency, quantization also reduces the bandwidth overhead of transferring KV cache across devices in distributed inference scenarios, accelerating multi-GPU or memory-bound workloads. However, a critical limitation of pure quantization approaches is that they retain all historical tokens, leaving the computational cost of attention unchanged. For long-context tasks, this means the quadratic complexity of attention persists despite the reduced memory usage.

The simple but with limited performance methods are usually based on the sliding window tokens eviction. Sliding window-based eviction methods—such as those used in Infinite-LLM (Han

et al., 2024) and StreamingLLM (Xiao et al., 2023)—retain only the initial cache tokens and those within a fixed sliding window, discarding the rest. However, this indiscriminate eviction strategy often leads to a notable degradation in generation quality.

Recent work by (Liu et al., 2024a,c) addresses the statistical properties of KV states, revealing distinct distribution patterns for keys and values. Their findings suggest that per-channel quantization for keys (which exhibit consistent variance across feature dimensions) and per-token quantization for values (which vary more significantly across sequence positions) yield better fidelity. This observation motivates our key insight: token importance for eviction—traditionally derived from attention weights—can instead be inferred from token- and channel-wise distribution patterns in the KV space. By leveraging these structural properties, we can design a pruning criterion, *LagKV*, that is both hardware-friendly (compatible with FA) and instruction independent, enabling compute savings alongside memory reduction.

2 Methodology

In this section, we formally introduce our KV compression method, *LagKV*. We begin by looking at the autoregressive process of the LLMs. Inspired by this, we propose a simple yet effective strategy to use the subsequent tokens to compress the previous ones.

2.1 Preliminaries

LLMs' next token prediction relies on the previous tokens. First, in the prefill stage, the model uses its tokenizer to convert the words to n indices of the embedding metrics $E \in \mathbb{R}^{V \times d}$ of the model and collects the representations to form a input matrix, $X \in \mathbb{R}^{n \times d}$. This matrix is the initial tokens of the first layer of LLM and then each layer will output a same shape matrix as next layer's input. To depict the operations in each layer, we follow the notation system from (Liu et al., 2023) with h attention heads. For each head $i \in [1, h]$ and head dimension d_h, we focus on the Query, Key, and Value states, which are converted from tokens by three linear transformation matrices W_i^Q, W_i^K, $W_i^V \in \mathbb{R}^{d \times d_h}$ sperately:

$$Q_i = XW_i^Q, K_i = XW_i^K, V_i = XW_i^V \tag{1}$$

The output $Y \in \mathbb{R}^{n \times d}$ is computed using the attention weights $A_i \in \mathbb{R}^{n \times n}$ and the final output matrix $W^O \in \mathbb{R}^{d \times d}$:

$$Y = Concat_{i \in [1,h]}(A_i V_i) W^O \text{where} A_i = \text{softmax}(\frac{Q_i K_i^T}{\sqrt{d_h}}) \tag{2}$$

When the new tokens are generated subsequently in the autoregressive inference, which named as decode stage, the embedding of generated token x is mapped to its respective Query, Key, and Value states for each head, and the previous KV cache is updated accordingly:

$$q_i = xW_i^Q, k_i = xW_i^K, v_i = xW_i^V \tag{3}$$

$$K_i = Cat[K_i : k_i], V_i = Cat[V_i : v_i], A_i = \text{softmax}(\frac{q_i K_i^T}{\sqrt{d_h}}) \tag{4}$$

Since $q_i \in \mathbb{R}^{1 \times d_h}$, the computation will be much faster because of the KV cache.

2.2 LagKV

Since the intrinsic property of autoregressive model, the next token representation will not change abruptly from the previous one. As observed in (Liu et al., 2024a), the called token-wise locality will show that the tokens in closer proximity have more similar K/V tensor values compared to tokens that are further apart.

And also, the StreamingLLM method (Xiao et al., 2023) has demonstrated that the head portion and sliding window of the KV cache are crucial. This suggests that cache compression should use subsequent tokens to assess whether prior tokens remain in the cache, rather than relying on the competition between them—as done in many attention-weight-based methods.

Inspired by above insights, we proposed our LagKV method as:

- After the prefill is done, start to apply the compression dynamically.
- Always keep the attention sink with size S and the already compressed part if had unchanged.
- Skip the compression if the length of the rest KV after the static part is less than $2L$, where we denote the lag size as L.
- Partition the rest KV with L. If it's not divisible by L, the modulo of it will be added to the sliding window.
- Recursively compute the KV cache score. Use the next partition as a reference, calculate token-wise max and min from the reference then use max-min to normalize the Key and Value states respectly. After the KV are normalized, calculate the channel-wise standard deviation then softmax. The equations are formally like:

$$min_i^{p,Z} = min_{seq}(Z_i^{p+1}) \tag{5}$$

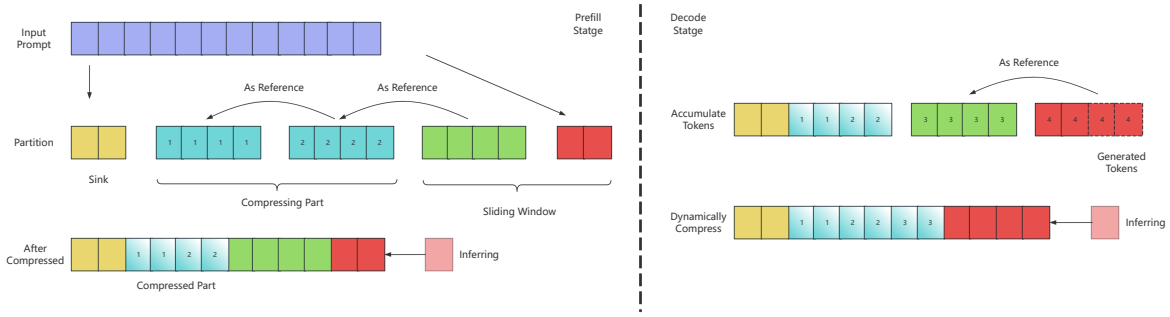


Figure 1: *LagKV* recursively compression process: partition the KV cache and use the next joint chunk as reference to compress the current one. Keep the rest of them as the sliding winow.

$$max_i^{p,Z} = max_{seq}(Z_i^{p+1}) \tag{6}$$

$$\bar{Z_i^p} = \frac{Z_i^p - min_i^{p,Z}}{max_i^{p,Z} - min_i^{p,Z}} \tag{7}$$

$$score(Z_i) = Softmax(Std.(\bar{Z}_i)) \tag{8}$$

where Z is one of $\{K, V\}$, p denotes the partition index, i represents the head index and seq for the sequence axis. Since the last partition has no reference can be used, our method will naturely have a sliding window with at least size L.

- Sum the scores of Key and Value to get the final score of each token:

$$score_i = score(K_i) + score(V_i) \tag{9}$$

- Base on the $score_i$, use the top-K strategy to select tokens in each partition and each head and add them to the compressed part.

The max-min normalization is applied along the sequence dimension, meaning each channel is normalized using statistics from lag-L tokens. Due to token-wise locality, the channel-specific norms of K_i and V_i are largely eliminated. The resulting normalized representations, $\bar{K}_i$ and $\bar{V}_i$, retain the original channel-wise variance, allowing the standard deviation to serve as a measure of token importance. The softmax operation then identifies and separates outliers, while the summed scores $score(K_i)$ and $score(V_i)$ determine their relative contributions.

As showed in Fig. 1, our method is recursively compressing KV cache in both prefill and decode parts, which is essential for the token-wise locality as mentioned above. It requires relative short distance to keep the similarity among the KV states. Subsequently, another benefit, it also avoids the bias from the long context with length much larger than L and the case when the question is at the end of the prompt.

We do not compare the *LagKV* score to the attention weights here. The attention weights vary on different incoming queries. But our scoring method does not depend on the query states or the tokens after the next joint chunk. It mainly finds the tokens that are not coherent to the next chunk and keep them in the cache. As in KIVI (Liu et al., 2024c) qunatization method, we need a rightful mean to find the correct variance and then prune the small ones. However, we use this strategy to evict tokens instead of quantizing them.

To caculate the compresstion ratio, we set the retained token ratio as r in each partition. In the partition chunk, only rL tokens will be kept and others are evicted. Therefore, the compression ratio C for the token sequence length $L_s \geq S + 2L$ can be expressed as:

$$L_R = S + rL(\lfloor \frac{L_s - S}{L} \rfloor - 1) + L + Mod(L_s - S, L) \tag{10}$$

$$C = 1 - \frac{L_R}{L_s} \tag{11}$$

Where L_R is the length of the KV cache after compression. For the case $L_s < S + 2L$, the compression ratio is zero.

3 Experiments

3.1 Settings

Base Models. We employ two open-source base models: Llama-3.1-8B-Instruct (Grattafiori et al., 2024) and Qwen2.5-7B-Instruct (Qwen et al., 2025). These models are main stream LLMs with moderate size and both leverage the GQA (Ainslie et al., 2023) technique to reduce the KV cache size.

Datasets. We use the facility in (Yuan et al., 2024) to extensively test our method. It contains two benchmarks: LongBench (Bai et al., 2024) and Needle-in-a-HaystackTest with Passkey-Retrieval (Kamradt, 2023; Mohtashami and Jaggi, 2023). We only test the 64-digit passkey retrieval task which is much more challenging. And because we are using a recursive and evicting compression strategy, it's easier to illustrate some insights with the partial match score other than the exact one in their report. Therefore, the default needle score will be the partial one across the whole work unless specified.

Parameter Spaces. Across the whole paper, we fix the sink size to $S = 16$ and vary the lag size L and ratio r. The values of L will be $L = 128$, 512 and 1024. The values of r will be $2\times$, $4\times$, $6\times$ and $8\times$ which correspond to $r = 0.5, 0.25, 0.167$, and 0.125 respectly.

Prefill stage. By default, like many other compression methods, compression begins after prefill completes for each layer. This is an efficient and accurate approach—preserving both the KV cache values and the first generated token (FGT) while reducing KV cache size. However, since we lack a reliable benchmark for long-context and long-generation scenarios, we will extend the passkey retrieval task by enabling chunk-by-chunk compression during prefill. This will help us evaluate how compression impacts long-generation performance, especially the FGT. Also, this chunked prefilling method will be useful for extreme long context processing.

Main results. The main results of this work is Table 1.

3.2 LongBench

For the LongBench dataset, the *LagKV* method performs very well across different ratios and lag sizes. When $r = 2\times$, for different L, it shows nearly zero loss ($< 5\%$) compared to the baseline in both models. When $L = 1024, r = 8\times$, the method still retains approximate 90% of the baseline performance. Since the compression ratio will increase when L decreases, the worse case is $L = 128, r = 8\times$ for both models but the method maintains at least 85% of the baseline performance.

3.3 Passkey Retrieval

The 64-digit passkey retrieval task is a challenging one for most token eviction strategies. As discussed in (Yuan et al., 2024), the most succesful eviction strategy H_2O (Zhang et al., 2024) performs well in 7-digit task (scoring 100% for all compression ratios) but degrades a lot in the 64-digit one (for $4\times$ in Llama-3, exact match score is 35% and partial match score is 70.8%). It happens because the strategy applies its compression after the prefill is done which means the FGT is not affected by the compression and the 7-digit passkey usually takes only 2 or 3 tokens. When the passkey size increases to 64, much more generated tokens are impacted by the compression. Many token-evict algorithms are struggling to maintain their performance in this case. In contrast, our method performs very well when the product of r and L is sufficient large enough (for $L = 1024, r = 4\times$ in Llama model, exact math score is 89% and partial match score is 96.57%). In this comparison, it should be noticed that H_2O is compressing with question-aware which will boost its performance significantly as discussed in (Feng et al., 2024; NVIDIA, 2024) while our method is totally instruction free.

Our recursive compression strategy will not perform well for the setups with small rL due to the fact that when the recursive window size is compressed to be close to or less than the length of the queried content, it's highly possible that only a small portion of the wanted information will be kept. In the task of 64-digit passkey retrieval, because digitals usually require more tokens to be represented than the same length words, the number of expected tokens is much larger than the similar tasks in LongBench sub tasks like Document QA, that leads to its results are more sensitive to small rL. As shown in Fig. 2, the Qwen model which uses one token for one digit degenerates faster than the Llama model which represents three digits by one token with smaller rL. It hints us that we must choose the compression ratio and the lag size carefully in considering the length of the expected content and the tokenizer of the LLM. A.1 shows all the details of the needle results.

3.4 Chunk-by-Chunk Compression in Prefill Stage

To enable chunk-by-chunk compression during prefill, we have to split the retrieval tokens like our recursive compression for long context with prefilling the first $S + 2L$ tokens and then L each time until all input tokens are prefilled. In such a way, the hidden values after the first chunk will be different from default prefill ones since less tokens

Table 1: Performance of LagKV.

Model	Method	Single. QA	Multi. QA	Summ.	Few-shot	Synthetic	Code	**LB Avg.**	**Needle**
Llama-3.1-8B-Instruct	Baseline	40.71	37.90	28.29	68.49	68.00	58.70	47.44	99.44
	L=1024,r=2x	39.42	37.12	27.38	67.71	68.50	58.83	46.74	99.27
	L=1024,r=4x	37.06	36.77	26.79	66.96	63.50	58.42	45.54	96.57
	L=1024,r=6x	35.74	36.08	26.33	66.33	60.50	57.91	44.65	91.77
	L=1024,r=8x	35.49	35.99	25.90	65.21	61.00	57.95	44.31	86.26
	L=512,r=2x	39.43	37.45	27.35	67.82	67.50	58.66	46.73	97.02
	L=512,r=4x	37.39	36.27	26.19	66.56	62.50	57.86	45.16	85.73
	L=512,r=6x	34.95	35.62	25.52	65.87	59.50	58.14	44.11	75.67
	L=512,r=8x	34.03	36.12	25.25	64.94	56.00	57.50	43.47	68.25
	L=128,r=2x	38.56	36.80	27.20	67.64	68.00	59.27	46.48	92.76
	L=128,r=4x	36.66	36.58	25.62	66.78	66.50	57.90	45.28	73.41
	L=128,r=6x	34.57	35.41	24.59	63.59	64.00	56.97	43.49	38.48
	L=128,r=8x	33.78	34.60	23.91	62.21	61.50	55.68	42.42	25.01
Qwen-2.5-7B-Instruct	Baseline	41.62	45.00	26.41	68.91	100.00	63.60	51.53	100.00
	L=1024,r=2x	39.80	42.85	26.11	67.66	99.50	63.12	50.33	99.75
	L=1024,r=4x	36.92	40.39	24.81	65.91	95.00	61.60	48.15	96.98
	L=1024,r=6x	35.77	39.74	24.68	65.28	93.50	61.45	47.52	77.47
	L=1024,r=8x	34.60	39.10	24.18	64.74	90.50	61.30	46.73	66.88
	L=512,r=2x	38.72	42.79	25.91	67.98	98.50	62.00	49.91	97.07
	L=512,r=4x	35.42	39.12	24.49	64.59	94.00	60.16	47.01	75.89
	L=512,r=6x	34.00	38.04	23.72	64.31	87.50	58.80	45.69	42.70
	L=512,r=8x	32.14	37.83	23.11	63.48	82.50	58.71	44.64	30.00
	L=128,r=2x	38.67	42.49	25.69	67.75	99.00	60.64	49.61	65.93
	L=128,r=4x	34.47	39.78	24.07	65.13	96.00	58.67	46.91	20.83
	L=128,r=6x	32.83	38.15	22.95	62.23	90.50	56.25	44.76	16.18
	L=128,r=8x	32.47	37.10	22.20	60.24	88.50	56.10	43.78	15.07

are seen in the forwarding. Then, the FGT may be different too. With the chunk-by-chunk prefill compression, we calcullated the FGT accuracy which is defined as the ratio of FGT same as the default prefill ones and also the overall needle scores, shown in Fig. 3.

The chunked prefill definitely diminishs the FGT accuracy as it drops from 100% to around 80% for $r = 8\times$ in both models. But we do not see it has a strong dependence on sequence lengths or needle depths in Fig. 4. These confirm that our method is able to retain the major part of the baseline capabilities in the case with long sequence hidden values impacted by the compression. It ensures that *LagKV* will deliver a good performance in the long generation scenarios.

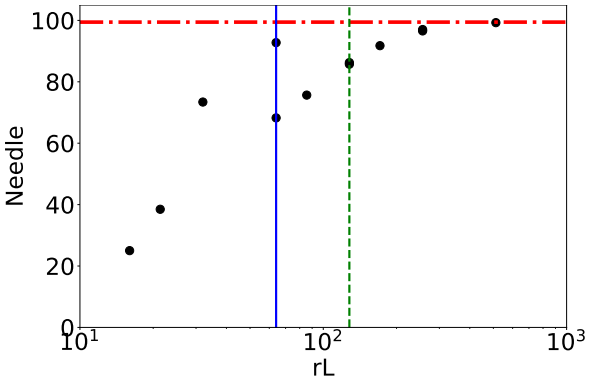
Meanwhile, we also notice that the FGT accuracy and overall needle scores suffer more degradation in Llama model with chunked prefill. It is mainly because different models exhibit various abilities of stable long generation (Quan et al., 2024).

3.5 Efficiency

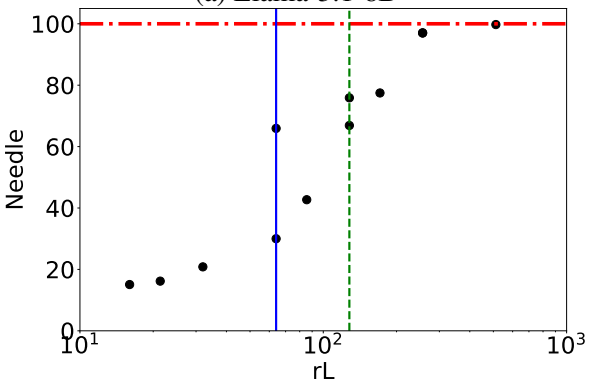
To evaluate the efficiency and memory saving of our algorithm, we test setups with $L = 1024$ and varying r in different context lengths $4K, 8K, 12K, 16K, 20K$ with the additional sink size. Two metrics, Time to First Token(TTFT) and Time Per Output Token(TPOT), are measured only with Llama-3.1-8B-Instruct and batch size 1 in a NVIDIA A800 GPU. By the default prefilling method, we see no difference in the TTFT measurements across all context lengths and ratios compared to the baseline ones which means the computation overhead is negligible. Instead, we measure TTFT by chunk-by-chunk prefilling in Fig. 5. The TPOT measurements which are averaged over 30 new generated tokens are present in Fig. 6. The overhead memory size[1] is in Fig. 7. For other L, see Appendix A.3.

In the longest context $20K$ tests, we observe $\approx$ 1.5 times speedup for TTFT and $\approx$ 2 times

[1]Measured by memory usage after model loaded subtracting from memory usage after prefill. Memory usage is calculated by calling function $torch.cuda.memory\_allocated$ after empty cache.

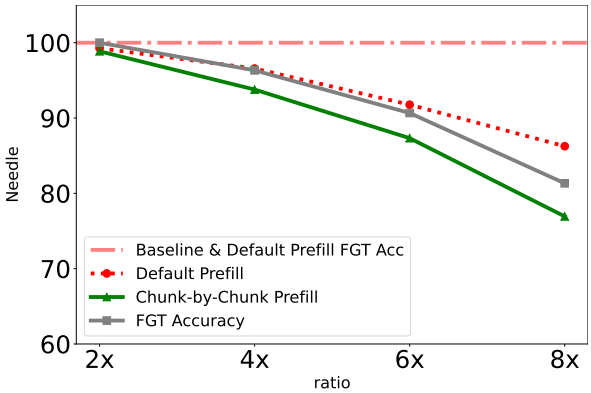


(a) Llama-3.1-8B

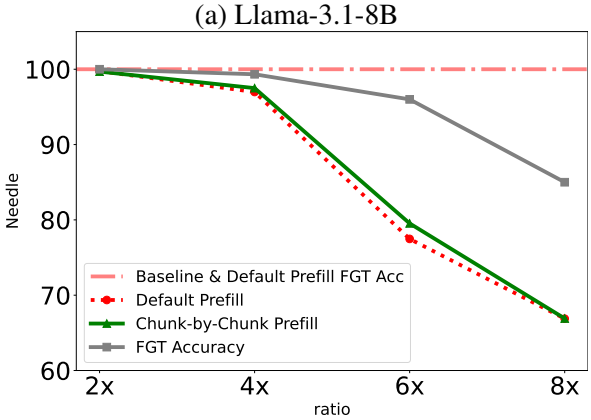


(b) Qwen-2.5-7B

Figure 2: The needle score vs different setups of rL. The horizontal dash-dot line is the baseline for each model. The x-axis is in log scale. We put two vertical lines $rL = 64$ (solid blue) and $rL = 128$ (dash green) for guidelines.



(a) Llama-3.1-8B



(b) Qwen-2.5-7B

Figure 3: The needle score and FGT accuracy for different prefill methods with $L = 1024$ only. The horizontal dash-dot line is the baseline for both needle scores and FGT accuracy since they are overlapping.

for TPOT (the baseline TTFT/TPOT devided by the compressed ones) in the highest compression ratio $8\times$. Meanwhile, the overhead memory usage drops to $\approx 17\%$ of the baseline with full KV cache, which aligns with the compression ratio formula Eq. 11.

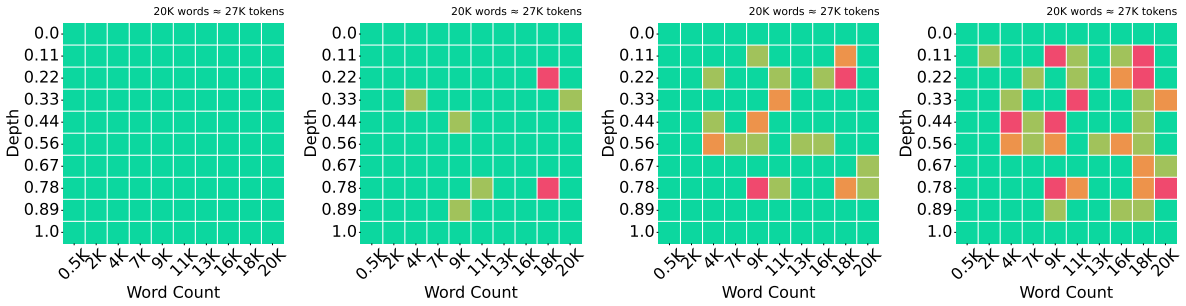
4 Related Works

The L_2 Norm-Based KV compression (Devoto et al., 2024) is an existing eviction approach that relies solely on KV information to compress the KV cache. This method computes token scores using the negative norm of key states. In contrast to our derivation from the autoregressive process and the token-wise locality, their method is formed by comparing the attention loss. Also, unlike our approach, it does not employ recursive compression during the prefill stage. As discussed in A.2, this method shows limitations in the 64-digit passkey retrieval task when the method is adapted to a recursive framework.

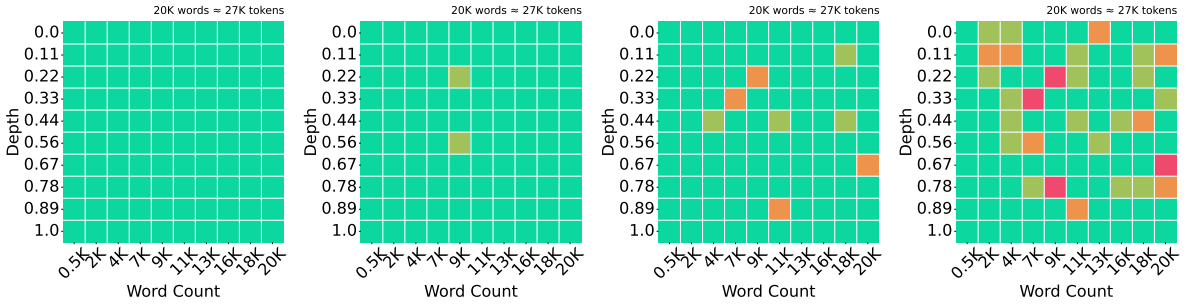
FINCH (Corallo and Papotti, 2024) introduces a prompt-guided KV compression method for the prefill stage, employing a chunk-by-chunk approach with instruction tokens appended to each document chunk. This design ensures the computation of attention submatrices between instructions and document chunks, enabling subsequent KV cache filtering. In contrast, our proposed chunked prefilling method operates without instructions, making it compatible with multi-turn queries. In other words, our approach transforms a causal LLM into a compressor capable of condensing long documents into compressed KV sequences, which can later be decompressed under varying instructions without reconstruction.

5 Conclusion

In this study, we propose *LagKV*, an attention-weight-free token eviction method. It achieves comparable performance on long-context tasks while significantly outperforming mainstream eviction strategies in 64-digit passkey retrieval tasks. These



(a) Llama-3.1-8B-Instruct



(b) Qwen-2.5-7B-Instruct

Figure 4: First Generated Token Accuracy for different setups, sequence lengths and needle depths with chunked prefill. It tests three trials on each depth.

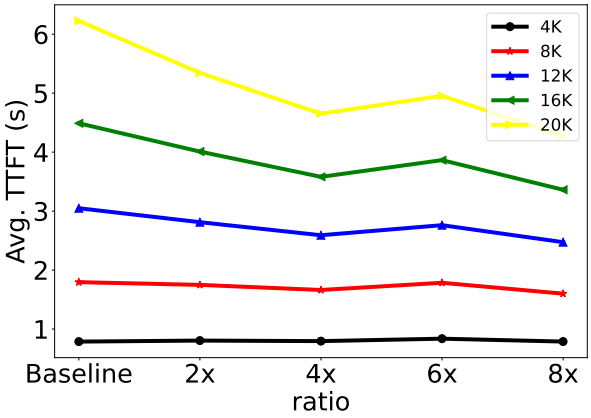


Figure 5: Chunked Prefilling Average TTFT (s) vs r for different context lengths with Llama-3.1-8B-Instruct and $L = 1024$.

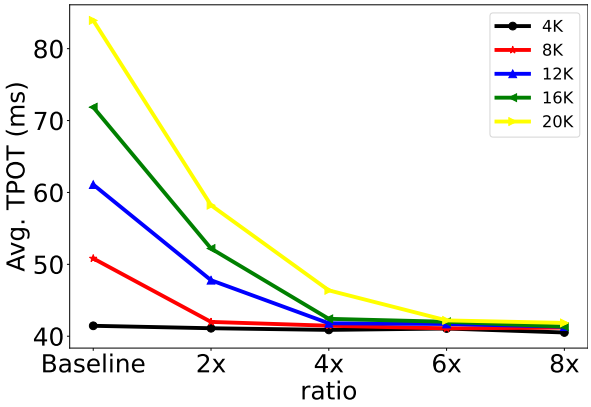


Figure 6: Average TPOT (ms) vs r for different context lengths with Llama-3.1-8B-Instruct and $L = 1024$.

results demonstrate that our method maintains robust long-text retrieval capabilities even at high compression ratios.

Unlike existing approaches, *LagKV* employs a recursive attention-weight-free strategy in both prefill and decode stages to determine token importance for future processing. It's indepent from query states and the rest part of the long prompt. Therefore our method offers a novel perspective on LLM mechanisms, shedding light on their inner workings in a fundamentally different way.

Limitations

Models: Our work tests only two popular models with median sizes. It's supposed to work on other models with different sizes but that is absent in this work due to the limitted computing power resources.

Methodology: With the significant results by the proposed framework, we still need a comprehensive theorectical explanation for the final form meanwhile the attention-weight based methods usually have a clear and straight one. Our method is

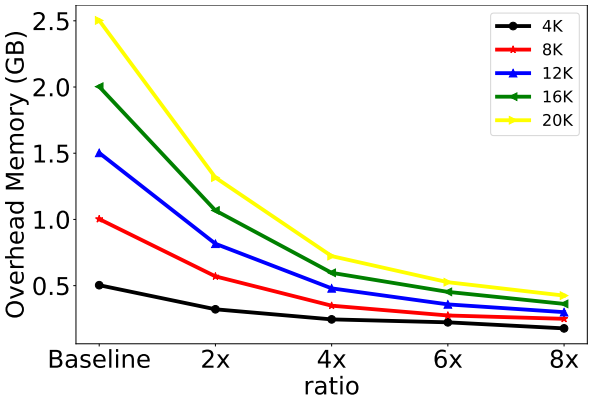


Figure 7: Overhead Memory (GB) vs r for different context lengths with Llama-3.1-8B-Instruct and $L = 1024$.

more intuitive and emprical which may vary on different models or tasks. The lag information in the next token partition used in this work still has many details to be explored, such as the lag size, weights between the key and value states, other variants of the scoring methods, etc.

Quantization: Apprently, our approach is totally compatible with KV quantization methods, like KIVI. With them, the memory size can be furthur reduced. We do not present here because we want to focus on the novellity of the recursive partition, lag reference and scoring method in KV cache.

Compression Budget: Now, our approach is evenly allocating KV cache in each head. As many methods proposed, different heads with different compression budgets can improve performance. This may be achieved in our method simply by setting a score threshold since the score is completely normalized and comparable among all heads, even layers. But it will take more work to handle the irregular shapes.

References

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. 2023. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901, Singapore. Association for Computational Linguistics.

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

Giulio Corallo and Paolo Papotti. 2024. FINCH: Prompt-guided key-value cache compression for large language models. *Transactions of the Association for Computational Linguistics*, 12:1517–1532.

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

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu

Zhang, and Zhen Zhang. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.

Alessio Devoto, Yu Zhao, Simone Scardapane, and Pasquale Minervini. 2024. A simple and effective l_2 norm-based strategy for KV cache compression. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 18476–18499, Miami, Florida, USA. Association for Computational Linguistics.

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.

Aaron Grattafiori, Abhimanyu Dubey, and Abhinav Jauhri .et al. 2024. The llama 3 herd of models.

Qiuhan Gu. 2023. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 2201–2203.

Chi Han, Qifan Wang, Hao Peng, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. 2024. LM-infinite: Zero-shot extreme length generalization for large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3991–4008, Mexico City, Mexico. Association for Computational Linguistics.

Gregory Kamradt. 2023. Needle In A Haystack - pressure testing LLMs. *Github*.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models.

Philippe Laban, Wojciech Kryscinski, Divyansh Agarwal, Alexander Fabbri, Caiming Xiong, Shafiq Joty, and Chien-Sheng Wu. 2023. SummEdits: Measuring LLM ability at factual reasoning through the lens of summarization. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9662–9676, Singapore. Association for Computational Linguistics.

Yucheng Li, Huiqiang Jiang, Qianhui Wu, Xufang Luo, Surin Ahn, Chengruidong Zhang, Amir H. Abdi, Dongsheng Li, Jianfeng Gao, Yuqing Yang, and Lili Qiu. 2025. SCBench: A KV cache-centric analysis of long-context methods. 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. *arXiv preprint arXiv:2404.14469*.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024a. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 38–56, New York, NY, USA. Association for Computing Machinery.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2024b. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 36.

Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. 2023. Deja vu: Contextual sparsity for efficient llms at inference time.

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*.

Amirkeivan Mohtashami and Martin Jaggi. 2023. Landmark attention: Random-access infinite context length for transformers.

NVIDIA. 2024. Llm kv cache compression made easy.

Shanghaoran Quan, Tianyi Tang, Bowen Yu, An Yang, Dayiheng Liu, Bofei Gao, Jianhong Tu, Yichang Zhang, Jingren Zhou, and Junyang Lin. 2024. Language models can self-lengthen to generate long texts.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 technical report.

Hanlin Tang, Yang Lin, Jing Lin, Qingsen Han, Shikuan Hong, Yiwu Yao, and Gongyi Wang. 2024. Razorattention: Efficient kv cache compression through retrieval heads.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention is all you need.

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. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 3258–3270, Bangkok, Thailand and virtual meeting. Association for Computational Linguistics.

Jiayi Yuan, Hongyi Liu, Shaochen Zhong, Yu-Neng Chuang, Songchen Li, Guanchu Wang, Duy Le, Hongye Jin, Vipin Chaudhary, Zhaozhuo Xu, Zirui Liu, and Xia Hu. 2024. Kv cache compression, but what must we give in return? a comprehensive benchmark of long context capable approaches. In *The 2024 Conference on Empirical Methods in Natural Language Processing*.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2024. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36.

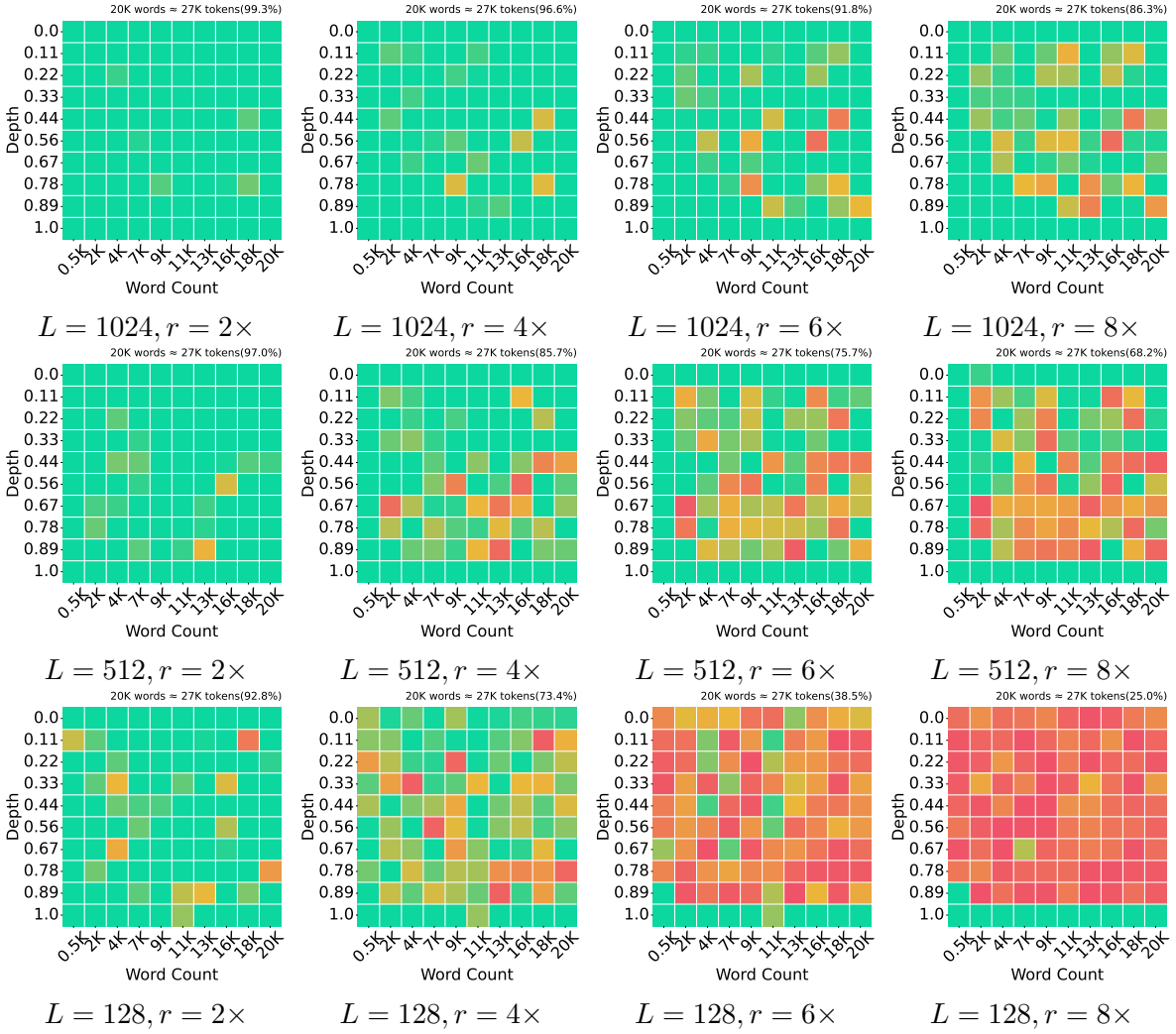


Figure 8: The 64-digit Passkey Retrieval of Llama-3.1-8B-Instruct for different setups with **partial** matching.

A Appendix

A.1 Detail Rresults of Passkey Retrieval

Here, we present all the Needle-in-a-Haystack results with 64-digit Passkey Retrieval for different setups. The partial matching results are in Fig.8 and 9 while Fig.10 and 11 with exact matching. Overall accuracies are noted within parentheses on the top-right corner of each sub graph.

A.2 Variants From *LagKV* Framework

Here we present two variants from *LagKV*. Both of them will only change the scoring methods but keep the attention sink and sliding window unchanged. And we only use the 64-digits passkey retrieval task which can easily distinguish eviction strategies as the detector. Among these tests, we keep $S = 16, L = 1024$ as constant.

The first one is called *LocalKV* which only skips using the reference from the next joint chunk tokens but replacing the equation Eq. 5 and 6 by the following equations:

$$min_i^{p,Z} = min_{seq}(Z_i^p) \tag{12}$$

$$max_i^{p,Z} = max_{seq}(Z_i^p) \tag{13}$$

Therefore, the min-max is totally from the local chunk instead of the remote one.

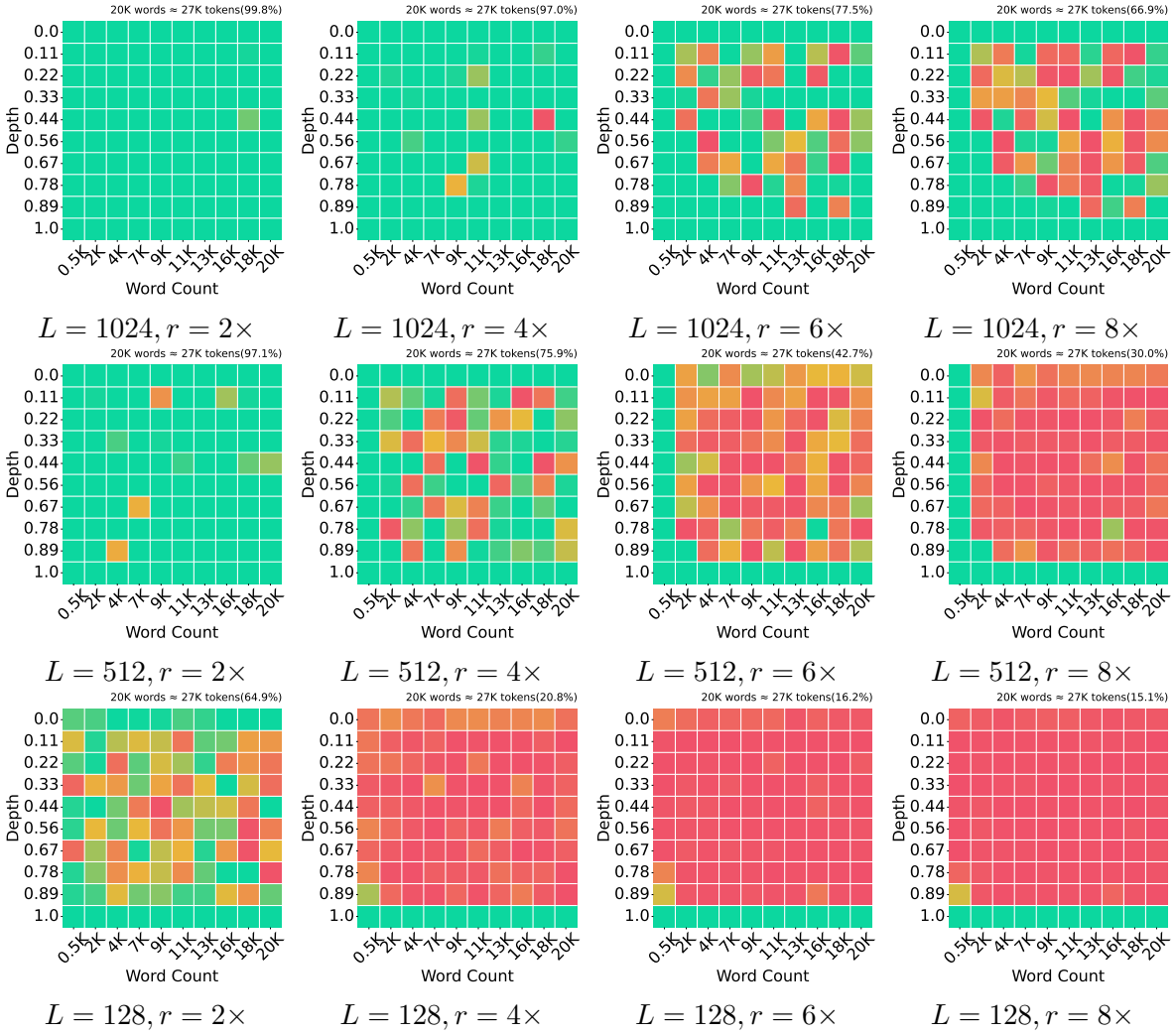


Figure 9: The 64-digit Passkey Retrieval of Qwen-2.5-7B-Instruct for different setups with **partial** matching.

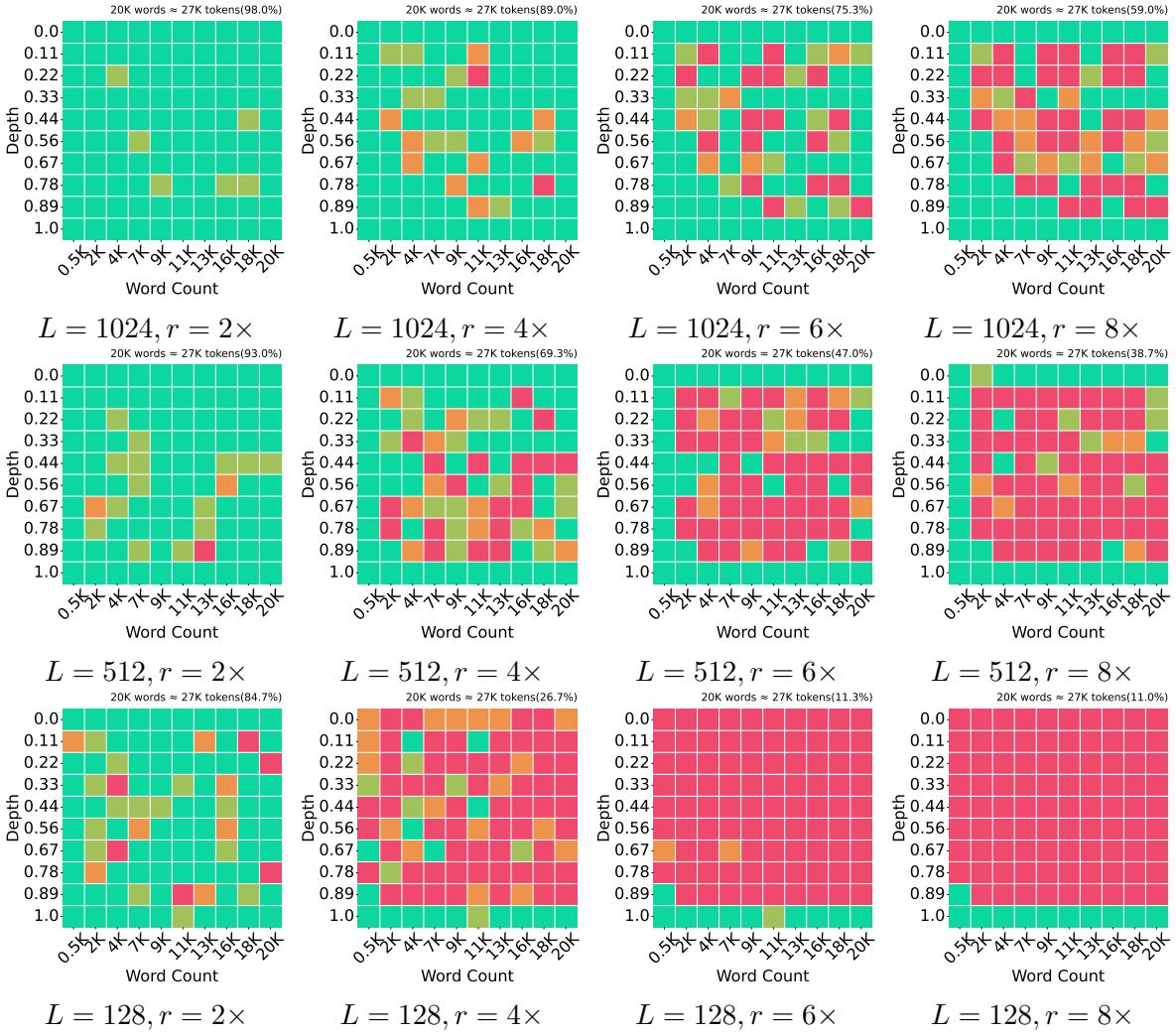


Figure 10: The 64-digit Passkey Retrieval of Llama-3.1-8B-Instruct for different setups with **exact** matching.

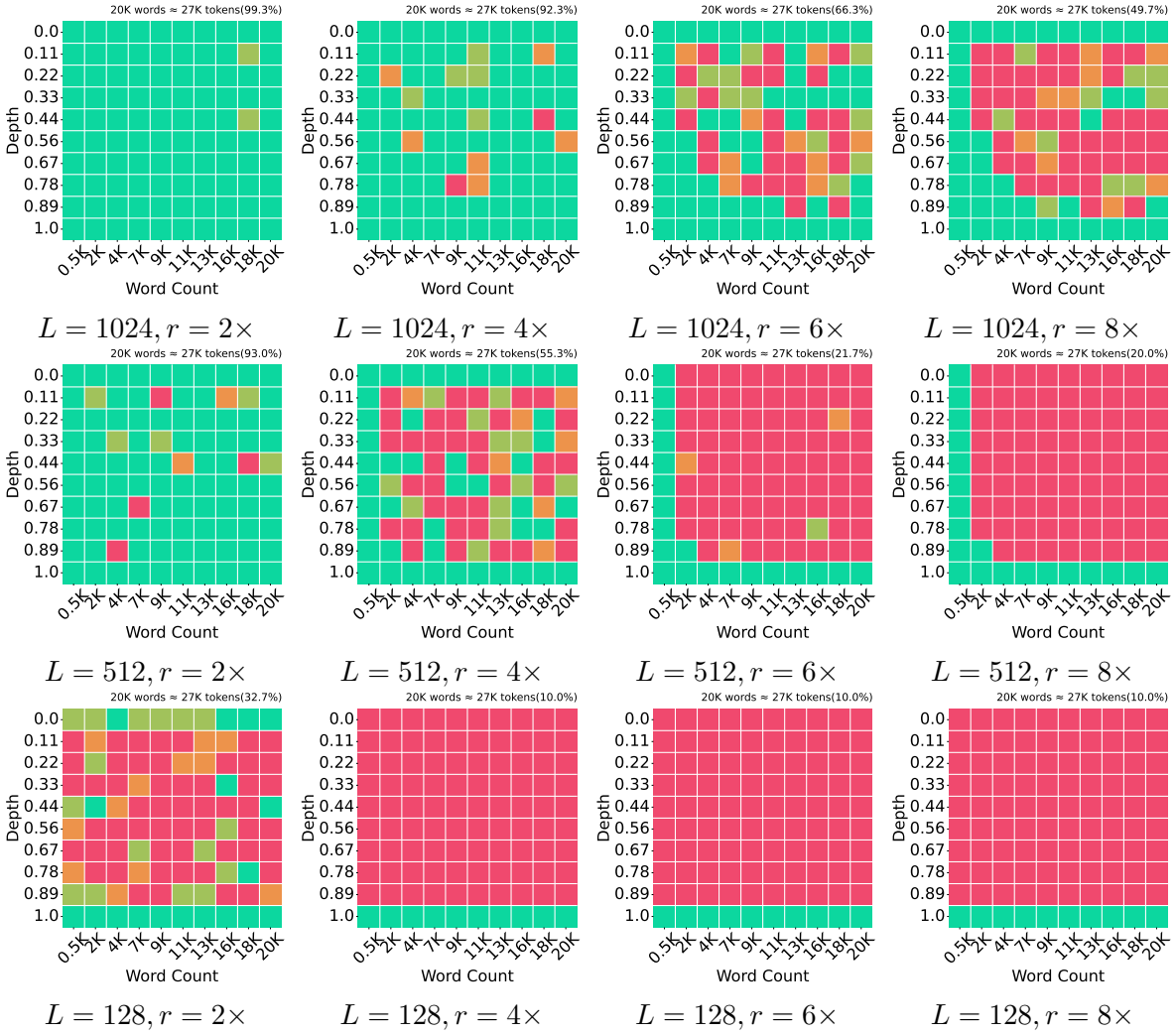


Figure 11: The 64-digit Passkey Retrieval of Qwen-2.5-7B-Instruct for different setups with **exact** matching.

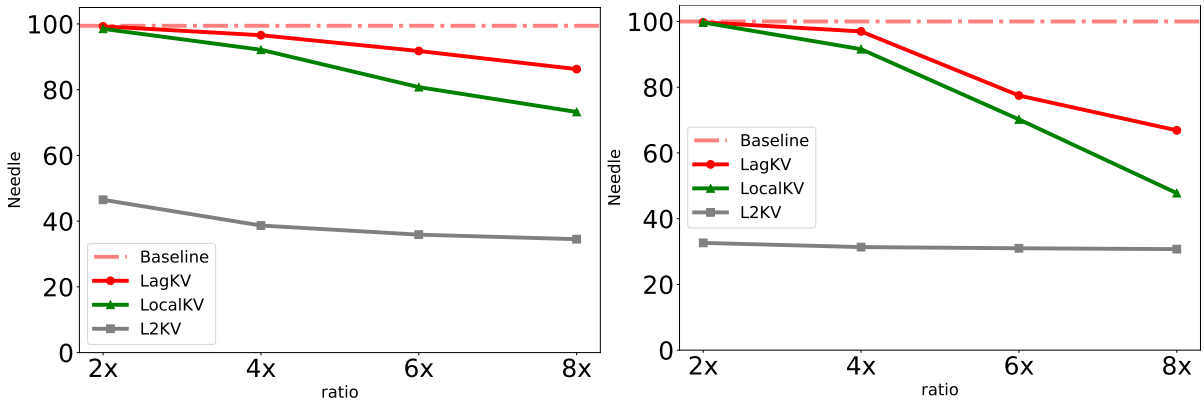


Figure 12: The 64-digit Passkey Retrieval **partial** match scores of different variants and compression ratios.

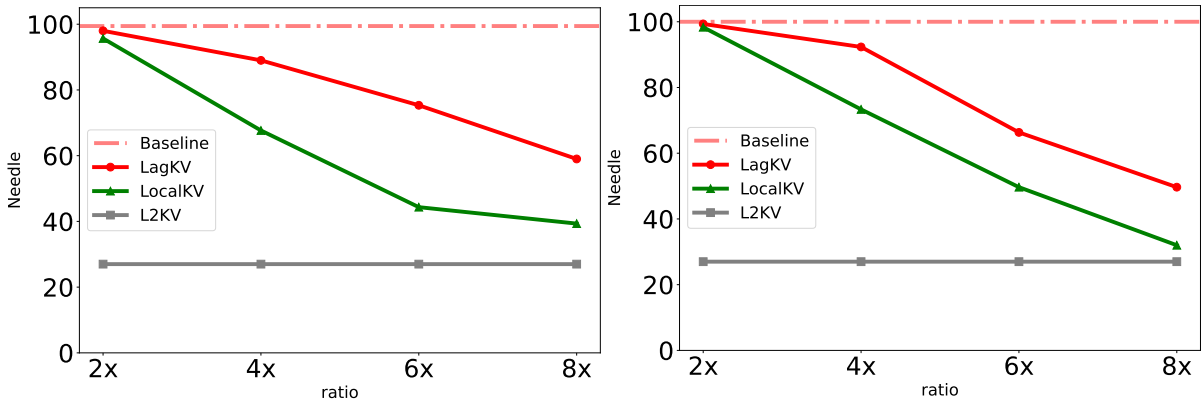


Figure 13: The 64-digit Passkey Retrieval **exact** match scores of different variants and compression ratios.

The second one is L_2 norm from (Devoto et al., 2024). We adapt the low key states norm method into the recursive framework by replacing Eq.9 by:

$$score_i = -Norm(K_i) \tag{14}$$

As suggested in their work, we skip the compression of the first two layers in this variant too.

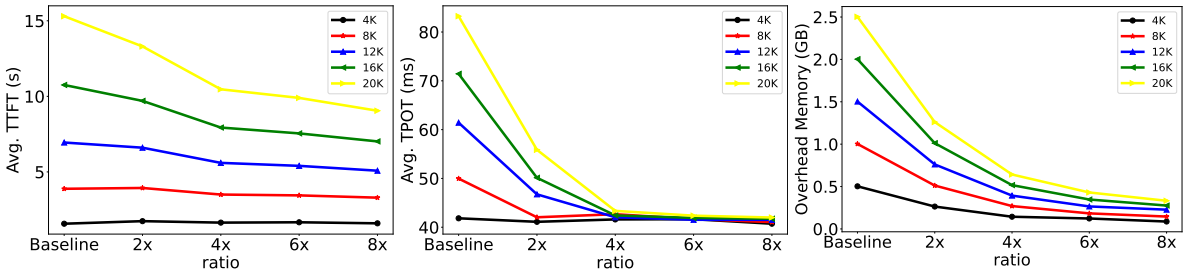
The results of the 64-digit passkey retrieval task are present in Fig. 12 and 13 with partial match scores and exact match scores. As we can see, the *LagKV* method is always the best one especially in the high compression ratios and the exact match cases.

The *LocalKV* variant performs closely to *LagKV* at low compression ratios but degrades significantly at higher ones. This behavior stems from the similarity between local and remote max-min statistical values, which aligns with the token-wise locality. Meanwhile, since all values of channels are strictly normalized to the range $[0, 1]$, the intrinsic property of each channel will be dropped with only token-wise information kept, resulting in the significant degradations in the high compression ratios.

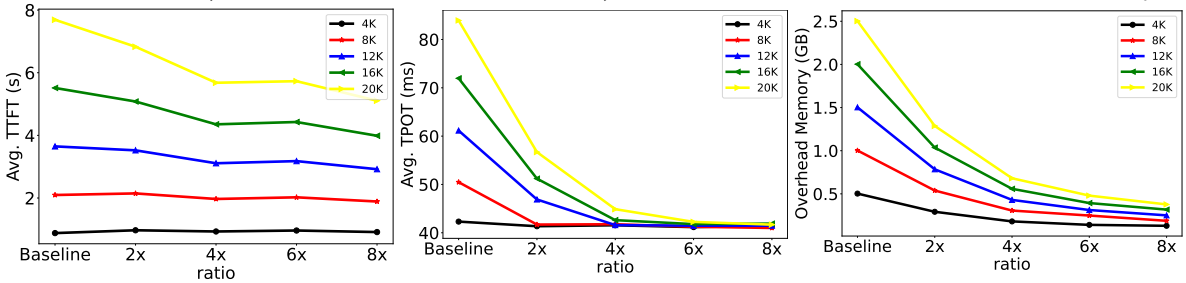
Since the setup of $L = 1024$ will have a chunk fully coverring the passkey when the context is shorter than 2K or the passkey is at 100% depth, the bottom line of the exact match score will be about 27% if the selected tokens did not mess up the output. That means the L_2 norm variant shows very limited performance with a constant exact match score 27% for all compression ratios and models.

A.3 Efficiency and Memory Usage

The measurements of other L are present in Fig. 14. The fluctuations of TTFT/TPOT in the ratio $6\times$ are likely related to the issue of pruned tensor shapes not being powers of 2.



$L = 128$, TTFT $L = 128$, TPOT $L = 128$, Overhead Memory



$L = 512$, TTFT $L = 512$, TPOT $L = 512$, Overhead Memory

Figure 14: Efficiency Measurement and Overhead Memory of Llama-3.1-8B-Instruct.