LAVa: Layer-wise KV Cache Eviction with Dynamic Budget Allocation

Anonymous ACL submission

Abstract

KV Cache is commonly used to accelerate LLM inference with long contexts, yet its high memory demand drives the need for cache compression. Existing compression methods, however, are largely heuristic and lack dynamic budget allocation. To address this limitation, we introduce a unified framework for cache compression by minimizing information loss in Transformer residual streams. Building on it, we analyze the layer attention output loss and derive a new metric to compare cache entries across heads, enabling layer-wise compression with dynamic head budgets. Additionally, by contrasting cross-layer information, we also achieve dynamic layer budgets. LAVa is the first unified strategy for cache eviction and dynamic budget allocation that, unlike prior methods, does not rely on training or the combination of multiple strategies. Experiments with benchmarks (LongBench, Needle-In-A-Haystack, Ruler, and InfiniteBench) demonstrate its superiority over strong baselines. Moreover, our experiments reveal a new insight: dynamic layer budgets are crucial for generation tasks (e.g., code completion), while dynamic head budgets play a key role in extraction tasks (e.g., extractive QA). As a fully dynamic compression method, LAVa consistently maintains top performance across task types.

1 Introduction

Large language models (LLMs) have shown remarkable capability in handling long-text scenarios, enabling advancements in tasks such as question answering (Kamalloo et al., 2023), code generation (Guo et al., 2023), and multi-turn dialogues (Chiang et al., 2023). To further enhance external knowledge integration, state-of-the-art models like Claude 3.5 (Anthropic and et al.), GPT-4 (OpenAI and et al., 2024), and Qwen2.5 Max (Qwen and et al., 2025) have extended their context lengths beyond 128K tokens. However, supporting such long contexts comes with increased computational challenges. One common approach to accelerating LLM inference is caching Key and Value vectors (KV Cache), but its high memory demand necessitates efficient cache compression techniques.

While existing compression methods have shown promise, they are largely heuristic, relying on statistical measures such as accumulated attention scores (Zhang et al., 2023; Oren et al., 2024; Li et al., 2024). These metrics are derived from empirical observations rather than a theoretical foundation. Additionally, although dynamic head allocation (Feng et al., 2024) and dynamic layer allocation (Qin et al., 2025) have been explored, no method, to our knowledge, fully adapts head and layer budgets.

To address this gap, we propose a unified framework for cache compression and budget allocation, which is formulated through the lens of minimizing information loss in Transformer residual streams (see Figure 1, and Sec. 3). Many existing methods can be formulated within our framework. Specifically, context compression methods (Qin et al., 2024a,b) aim to minimize global information loss at the logits layer. In contrast, KV Cache compression methods (Zhang et al., 2023; Cai et al., 2024; Qin et al., 2025) primarily focus on local information loss at the head or layer levels.

Our framework provides a principled approach to designing new algorithms. This paper introduces a novel method based on *Layer Attention Output Loss*, which measures the impact of compression on the information retained in each layer after multi-head attention. The layer-wise loss function provides a balanced perspective on both local information within layers and global information flow across layers. Within each layer, the loss function guides the design of a scoring mechanism to assess token importance across heads, allowing for simultaneous head budget allocation and cache eviction.

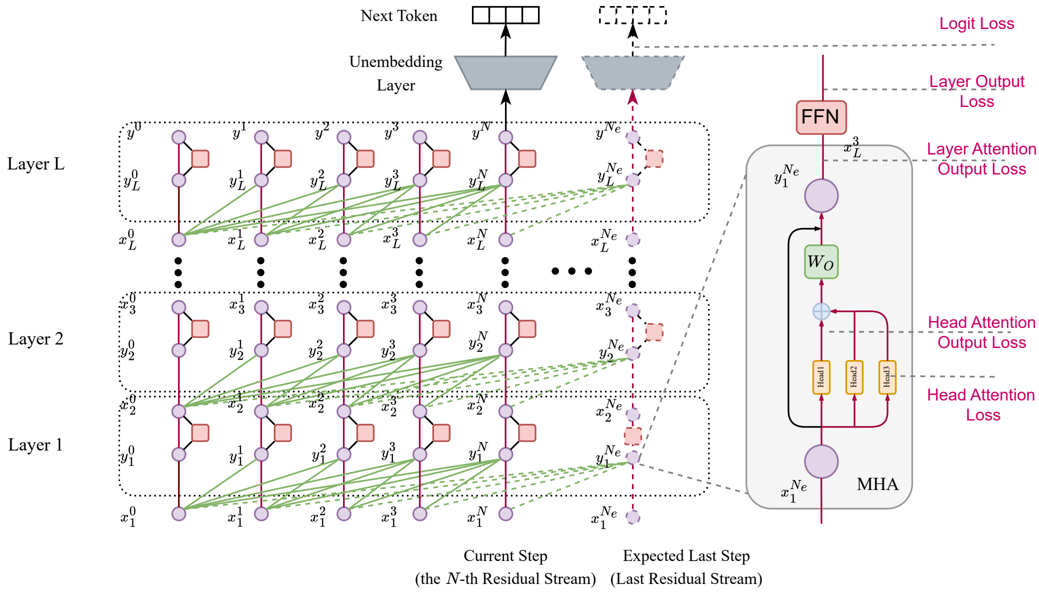

Figure 1: Information flow in decoder-only LLMs. The decoding process can be seen as operating on the current *residual stream*. Each residual stream (red lines) corresponds to one token, and is considered as a *communication channel*. Attention heads copy information from past residual streams to the current one (green lines) .

Across layers, it enables dynamic layer budget allocation by comparing information between layers. Our method is theoretically grounded, and significantly simpler than CAKE, the only training-free method with dynamic layer budgets.

Extensive experiments were conducted using various LLM series on the LongBench and Needle in a Haystack benchmarks. The results consistently demonstrate LAVa's strong ability to preserve the model's long-text comprehension under various memory constraints. Additionally, compared to a full cache implementation of FlashAttention-2, LAVa significantly reduces memory consumption while simultaneously reducing latency (9× faster decoding for 128K-token sequences). Our empirical findings highlight that dynamic layer budgets are essential for generation tasks, while dynamic head budgets are crucial for text extraction tasks. Achieving dynamic budget allocation at both the head and layer levels is key to optimizing performance across different tasks.

Our Contributions: 1) We introduce a **principled framework for KV Cache eviction** by analyzing the information flow through Transformer residual streams, accounting for information loss at various points during decoding. 2) Building on this framework and the notion of information loss at the layer-wise attention output, we propose LAVa—a unified method that simultaneously performs KV cache eviction and dynamic budget allocation. To the best of our knowledge, LAVa is the first training-free method to achieve dynamic budget allocation without relying on multiple combined metrics, making it simple for practical purposes. 3) Evaluations on LongBench, Needle in a Haystack, Ruler and InfiniteBench demonstrate that our simple method **outperforms strong baselines**. 4) Experiments reveal new insights into the role of dynamic budget allocation across different tasks, offering guidance for the adaptive selection of strategies.

2 The Information Flow of LLM Decoding Process with KV Cache

KV Cache is initialized at *prefilling* stage, which basically computes the Key and Value for tokens in the initial prompts in the standard way (Vaswani, 2017). In the following, we assume that there exists a KV Cache of $(N-1)$ previous tokens and demonstrate how decoding is performed at step-N.

Notations The LLM has L layers, each has H heads. The model and head dimensions are d and $d_h = d/H$; K_l, V_l are the KV Cache for the l-th layer up to the current time step (the N-th token), which are of $[H, (N-1), d_h]$ sizes. The full notation Table 3 is in Appendix A.

Decoding Process According to (Ferrando and Voita, 2024), LLM decoding can be viewed as operating on the current (N-th) *residual stream*, as illustrated in Figure 1. Specifically, suppose that

x_l^N is the current input for layer l, we first calculate the corresponding Q_l^N, K_l^N, V_l^N as follows:

$$Q_l^N = x_l^N W_l^Q; K_l^N = x_l^N W_l^K; V_l^N = x_l^N W_l^V$$

where Q_l^N, K_l^N, V_l^N are of size $(H \times 1 \times d_h)$, containing H head-wise caches. The layer-wise KV Cache is then updated as follows:

$$K_l = Cat[K_l, K_l^N], V_l = Cat[V_l, V_l^N]$$

where K_l, V_l are tensors of size $(H \times N \times d_h)$, and Cat indicates the concatenation operation.We then calculate the attention scores of step-N for layer-l:

$$A_l^N = Cat_{h\in[H]}\left(A_{l,h}^N\right)$$

where $A_{l,h}^N = Softmax(\frac{Q_{l,h}^N (K_{l,h})^T}{\sqrt{d_h}})$. Here, $A_{l,h}^N[i]$ ***indicates how much the token at step-N attends to the token-i ($i \le N$).*** Layer-l attention output is calculated as follows:

$$y_l^N = Cat_{h\in[H]}(A_{l,h}^N V_{l,h}) W_l^O \in \mathbb{R}^{1\times d}$$

The layer output x_{l+1}^N is calculated as $x_{l+1}^N = y_l^N + FFN(y_l^N)$, which is then passed as the input the next layer $l+1$. In the last layer, we exploit an un-embedding layer ($W^M \in \mathbb{R}^{d\times|\mathcal{V}|}$) to get the probability vector p^N for next token sampling.

3 A Principled Framework for KV Cache Eviction based on Information Loss

Given the KV Cache, compression can be seen as masking entries in the KV tensors so that the attention heads cannot copy masked information to the later residual streams. Formally, one can define the attention mask $\mathcal{I}_{l,h}$ for layer-l and head-h:

$$\mathcal{I}_{l,h}[i] = \begin{cases} 1 & \text{if } K_{l,h}[i] \text{ and } V_{l,h}[i] \text{ are retained} \\ 0 & \text{evict } K_{l,h}[i] \text{ and } V_{l,h}[i] \end{cases}$$

The goal is to find a KV Cache eviction policy so that to minimize the information loss for the logits at the last layer (p^N) for all subsequent residual streams (from N to N_e; see Figure 1). Let $\mathcal{P}$ denote this logit loss, and $\mathbb{B}$ be the memory constraint. The unified problem for budget allocation and cache eviction can be defined as follows:

$$\min_{\mathcal{I},\mathcal{B}} \mathcal{P}(\mathbf{x}_1^{1\ldots N}, \mathcal{I}, \mathcal{B}) \tag{1}$$

$$\text{st.} \sum_{i\in[N]} \mathcal{I}_{l,h}[i] = \mathcal{B}_{l,h};$$

$$\sum_{h\in[H]} \mathcal{B}_{l,h} = \mathcal{B}_l; \sum_{l\in[L]} \mathcal{B}_l = \mathbb{B}$$

$$\mathcal{I}_{l,h}[k] = 1 \text{ }, \forall l, h; \text{ and } \forall k \in [N-w, N]$$

Here, $\mathcal{B}_{l,h}$ represents the budget for layer-l and head-h, $\mathcal{B}_l$ denotes the total budget for layer-l. The final constraint ensures that the most recent tokens within a window of size w are retained for all heads, aligning with the common practice in the literature.

As computing the loss over future, unseen tokens is impractical. To address this, we approximate the loss by considering only residual streams up to the current step N. Considering the current step-N, one can define $\mathcal{P}$ as the cross-entropy loss between p^N and $\hat{p}^N$, which is the logit obtained with the attention mask (Qin et al., 2024a). Additionaly, since the search space for the mask matrix is combinatorial, we instead search for a scoring function s, where $s_{l,h}[i]$ assigns an importance score to token i at layer l and head h. This scoring function allows us to greedily choose the least important entries to be masked $\mathcal{I} = Select(s, \mathcal{B})$. All in all, we have the following (surrogate) optimization problem:

$$\min_{\mathcal{B}, s\in\mathcal{F}} \mathcal{P}(\mathbf{x}_1^{1\ldots N}, s, \mathbb{B}) \tag{2}$$

where $\mathcal{F}$ denotes the space of all scoring functions. The scoring function can be parameterized by a network ϕ, which is then found through offline training. This is the common approach employed in context compression methods (Qin et al., 2024a,b).

The aforementioned approach to minimizing *Global Logit Loss* can be impractical for online inference when the scoring function is computationally expensive. A more feasible alternative is to focus on local information and apply localized KV Cache eviction. For instance, *Head Attention Loss* can be used for head-wise eviction, a strategy adopted by most existing methods (Zhang et al., 2023; Li et al., 2024; Qin et al., 2025). In this case, the scoring functions are lightweight, relying on simple statistical features, like head-wise attention weights. Table 1 summarizes how existing methods can be formalized within our framework, with further details provided in Appendix B.

4 LAVa: Layer-wise Cache Eviction with Dynamic Budget Allocation

4.1 Layer Attention Output Loss and the Scoring Function

The aforementioned framework provides a principled approach to designing new algorithms for KV Cache eviction. This section demonstrates the design of our novel algorithm based on *Layer Attention Output Loss* (see Figure 1). Specifically, we

Methods	Budgets		Scoring Function	Loss
	$\mathcal{B}_{l,h}$	$\mathcal{B}_l$		
SnapKV (Li et al., 2024)	$\mathcal{B}_l/H$	$\mathbb{B}/L$	Recent attention scores $s_{l,h}[i] = \frac{1}{w}\sum_{j=N-w}^{N} A^j_{l,h}[i], \forall i < N-w$	Head Attention
CAKE (Qin et al., 2025)	$\mathcal{B}_l/H$	Dynamic	Recent attention scores + attention shifts $s_{l,h}[i] = \gamma \text{VAR}_{j=N-w}^{N}([A^j_{l,h}[i]])$ $+\frac{1}{w}\sum_{j=N-w}^{N} A^j_{l,h}[i], \forall i < N-w$	
AdaKV (Feng et al., 2024)	Dynamic	Fixed	Recent attention scores (like SnapKV)	Layer Attention Output
LAVa (Ours)	Dynamic	Dynamic	Recent attention scores $\times$ value norm $s_{l,h}[i] = \frac{max_k \|V_{l,h}[k]\|_1}{w}\sum_{j=N-w}^{N} A^j_{l,h}[i]$	

Table 1: Summary of representative methods for KV Cache compression. LAVa is the only method to support dynamic head ($\mathcal{B}_{l,h}$) and layer (B_l) budgets. For the full table and more comparison, please refer to Appendix B.

.

show how our scoring function is designed based on analyzing the upper bound of the loss and how we can exploit the scoring function for layer-wise cache eviction with dynamic budget allocation.

Lemma 1. *Based on the L_p norm, the layer attention output loss due to the attention mask $\mathcal{I}$ is measured for layer-l at the current (N-th) residual stream as follows:*

$$\mathcal{P}(\boldsymbol{x}_1^{1\ldots N}, \mathcal{I}, \mathcal{B}) = \|y_l^N - \hat{y}_l^N\|_p \tag{3}$$
$$= \left\| Cat_h \left[\left(A_{l,h}^N - \frac{A_{l,h}^N \odot \mathcal{I}_{l,h}}{\|A_{l,h}^N \odot \mathcal{I}_{l,h}\|_1} \right) V_{l,h} \right] W_l^O \right\|_p$$

where $\odot$ indicates element-wise multiplication and $\hat{y}_l^N$ indicates the layer attention output obtained by masking the KV Cache with $\mathcal{I}$ (equivalently, after KV Cache eviction).

We then develop a new upper bound for the L_1 norm and provide the result in Theorem 1. The proof of these are both provided in Appendix C.

Theorem 1. *The L_1 norm of the layer attention output loss can be bounded by:*

$$\|y_l^N - \hat{y}_l^N\|_1$$
$$\leq 2\hat{C} \sum_{h\in[H]} \sum_{i\in[N]} A_{l,h}^N[i] \bar{V}_{l,h} \left(1 - \mathcal{I}_{l,h}[i]\right) \tag{4}$$

where $\hat{C} = \|W_l^{O^T}\|_1$ is a constant independent of any head or token within layer-l; $\bar{V}_{l,h} = max_{k\in[N]} \|V_{l,h}[k]\|_1$ is a head-dependent value.

Given a fixed budget B_l, we consider a greedy algorithm that iteratively evicts one cache entry at a time until the cache budget is met. We evict the entries with the smallest scores, given by the scoring function $s_{l,h}[i] = A_{l,h}^N[i]\bar{V}_{l,h}$ to minimize the upper bound. Notably, this function incorporates a head-dependent value $\bar{V}_{l,h}$, which should not be ignored when comparing KV Cache entries across different heads. This is different from AdaKV (Feng et al., 2024), which considers the layer attention output loss yet does not take into account the values. This also provides a theoretical justification for the introduction of values into the scoring, which has been exploited heuristically in VATP (Guo et al., 2024). 4It is noted that we derive our metric through a detailed reasoning process, independently from VATP. The process is key to understanding the approximations we introduce, which enable future improvements. Moreover, recognizing that the metric is inherently grounded in a layer-wise perspective enables the design of dynamic budget allocation strategies, as demonstrated below. Empirical comparison to VATP is given in Table 5.

The scoring function $s_{l,h}[i] = A_{l,h}^N[i]\bar{V}_{l,h}$ described earlier is based solely on analyzing the current residual stream (the N-th decoding step). To improve the performance for KV Cache eviction, we can incorporate information from all past residual streams similarly to H2O (Zhang et al., 2023). However, doing so introduces more computational overhead. Inspired by SnapKV (Li et al., 2024), we instead incorporate information from recent w residual streams, yielding a new scoring function.

Definition 1. *Layer-wise Attention and Value (LAVa) score for the token-i at layer-l, head-h is defined as follows:*

$$s_{l,h}[i] = \frac{max_{k\in[N]} \|V_{l,h}[k]\|_1}{w} \sum_{j=N-w}^{N} A_{l,h}^j[i] \tag{5}$$

Based on this scoring function, we develop the layer-wise KV Cache eviction as outlined in Algorithm 1. Notably, we only evict entries outside

Algorithm 1 LayerEvict: Layer-wise KV Cache Eviction based on LAVa Score

1: **Input:** Budget $\mathcal{B}_l$, KV Cache K_l, V_l
2: **Output:** Compressed KV Cache $\hat{K}_l, \hat{V}_l$
3: $s_l = [\,]$
4: **for** $h = 1$ to H **do**
5:     Calculate $s_{l,h}[i], \forall i \notin [N-w, N]$ based on Eq. 5
6:     $s_l.extend(s_{l,h})$
7: **end for**
8: **function** EVICT($\mathcal{B}_l, s_l, K_l, V_l$)
9:     $\mathcal{S}_l \leftarrow \mathcal{B}_l$ largest entries based on s_l
10:     $\mathcal{I}_{l,h}[k] = 0, \forall (h,k) \notin \mathcal{S}_l$
11:     **for** $h = 1$ to H **do**
12:         $\hat{K}_{l,h} = K_{l,h} \odot \mathcal{I}_{l,h}$
13:         $\hat{V}_{l,h} = V_{l,h} \odot \mathcal{I}_{l,h}$
14:     **end for**
15:     **Return** $\hat{K}_l, \hat{V}_l$
16: **end function**
17: **Return** EVICT($\mathcal{B}_l, s_l, K_l, V_l$)

Algorithm 2 LAVa: Dynamic Budget Allocation and Cache Eviction based on LAVa Score

Input: Total Budget $\mathbb{B}$, KV Cache K, V Number of Layers L
Output: Compressed KV Cache $\hat{K}, \hat{V}$
$s = [\,], e = [\,], \hat{K} = K, \hat{V} = V$
for $l = 1$ to L **do**
    Calculate s_l based on Eq. 5
    Calculate e_l based on Eq. 6, 7
    $s.append(s_l)$
    $e.append(e_l)$
    for $\tilde{l} = 1$ to l **do**
        $\mathcal{B}_{\tilde{l}} = \frac{e_{\tilde{l}}}{\sum_l e_l}\mathbb{B}$
        $\hat{K}_{\tilde{l}}, \hat{V}_{\tilde{l}} = $ EVICT($\mathcal{B}_{\tilde{l}}, s_{\tilde{l}}, \hat{K}_{\tilde{l}}, \hat{V}_{\tilde{l}}$)
    end for
end for
Return $\hat{K}, \hat{V}$

the recent window $[N-w, N]$, effectively retaining the most recent tokens as specified by the final constraint in the optimization problem (Eq. 1).

Dynamic Head Budget. Our eviction method operates across attention heads within layer-l. Specifically, we flatten the LAVa scores from all heads in the layer into a one-dimensional array s_l (Algorithm 1, lines 3–6). We then compare and rank $\mathcal{B}_l$ cache entries across all heads for layer-wise eviction, effectively obtaining dynamic head budget while performing eviction.

4.2 Layer Budget Allocation

Recently, CAKE (Qin et al., 2025) and PyramidKV (Cai et al., 2024) have demonstrated the potential of allocating different budgets across layers. PyramidKV, however, is suboptimal as it assigns a fixed allocation pattern regardless of the input. In contrast, CAKE is prompt-dependent allocation (dynamic) but combines different scores for cache eviction and budget allocation, which requires tuning three hyperparameters, hindering its practical application. Below, we describe our hyperparameter-free algorithm based on the LaVa score.

Our key idea is that layers with greater uncertainty in *determining which cache entry to evict* should be allocated a larger budget. Specifically, based on the LAVa score, *the probability of evicting token-k at layer-l and head-h* is obtained by normalizing the LAVa scoring values:

$$\hat{s}_{l,h}[i] = \frac{s_{l,h}[i]}{\sum_{k,h} s_{l,h}[k]} \tag{6}$$

The uncertainty for layer-l is then measured by the *normalized entropy* as follows:

$$e_l = \frac{-\sum_{h,i}(\hat{s}_{l,h}[i] \log \hat{s}_{l,h}[i])}{H \times N} \tag{7}$$

With such a measure, we can first initialize all KV Cache through prefilling, followed by cache compression. Unfortunately, this approach results in a high memory peak after prefilling (and before compression). To address this, the common practice is that we perform prefilling and cache eviction layer by layer. For dynamic layer budget allocation, we draw inspiration from CAKE: after prefilling layer-l, the lower layers ($< l$) are recompressed. As a result, a lower layer is compressed multiple times using the same LAVa scores, but the budget is adjusted, becoming smaller over time as the memory is shared with more layers being prefilled. The complete algorithm is outlined in Algorithm 2.

4.3 LLMs with GQA

Group Query Attention (GQA) (Ainslie et al., 2023) is the technique most modern LLMs adopt due to its balance between performance loss and memory efficiency. In GQA, the KV Cache is compressed by sharing a single KV Cache among all heads within a group. When applying LAVa scores to GQA, we take a conservative approach:

the group-wise score for a token is determined as the maximum of its head-wise scores within the corresponding group. In other words, we tend to retain the entry as long as it is important for at least one head within the group.

5 Experiments

5.1 Experimental Settings

Backbone LLMs. We evaluate two series of LLMs: Mistral-7B-Instruct-v0.2 (Jiang et al., 2023), Qwen2.5-7/14/32B-Instruct (Qwen and et al., 2025), all with a context length of 32k. These models are widely adopted for their moderate parameter sizes and strong performance all utilizing GQA (Ainslie et al., 2023).

Evaluation Benchmarks. To validate the effectiveness of our algorithm, we perform evaluation ***LongBench*** (Bai et al., 2024), a bilingual, multi-task benchmark for long-context understanding. It comprises 21 datasets across six task categories in both English and Chinese, with an average length of 6,711 words (English) and 13,386 characters (Chinese). LongBench covers key long-text application areas, including single-document QA, multi-document QA, summarization, few-shot learning, synthetic tasks, and code completion. We also conduct experiments on ***Needle In A Haystack*** (Cai et al., 2024; Liu et al., 2024; Fu et al., 2024), ***Ruler*** (Hsieh et al., 2024) and ***InfiniteBench*** (Zhang et al., 2024), of which the results are given in Appendix D.

Baseline Methods. We compare our methods against several baselines: PyramidKV, SnapKV, Ada-SnapKV, Ada-PyramidKV, and CAKE. Among these, PyramidKV and CAKE allow different layer budgets. AdaKV is derived from the layer attention output loss but relies solely on attention for its scoring function and does not incorporate dynamic layer budget allocation. Ada-SnapKV employs the same scoring function and uniform layer allocation as SnapKV but allows dynamic head budgets. Ada-PyramidKV follows the same approach but assigns fixed, varying budgets across layers like PyramidKV.

Pooling operators, such as max pooling or average pooling, can be applied to token score vectors to smooth score variations across adjacent tokens (Li et al., 2024; Cai et al., 2024; Qin et al., 2025). This strategy is also employed in the implementation of LAVa and all the baselines. For pooling operation, for all methods, we adopt maxpool function and set kernel size as 7. More information is given in Appendix B, and for implementation details, please refer to Appendix D.

5.2 Main Results

Table 2 presents the results of Mistral-7B with different eviction policies on LongBench, revealing several key observations. First, *LAVa outperforms all baselines across different budgets*, with a more pronounced advantage at smaller budgets. Second, among methods requiring no hyperparameter tuning (SnapKV, Ada-SnapKV, and LAVa), LAVa achieves the best performance, significantly surpassing others. For instance, at $\mathbb{B} = 128\text{HL}$, LAVa achieves an average score of 36.74, compared to Ada-SnapKV's 35.82. And finally, *LAVa and CAKE excel in code-related tasks*. On RepoBench-P with a 128HL budget, LAVa (48.92) and CAKE (48.53) outperform Ada-SnapKV (46.85) by a significant margin. This is interesting given that Ada-SnapKV surpasses CAKE on average over 20 datasets. Similar trends are observed with the Qwen series and presented in Appendix D.

To further investigate the last observation, we categorize the 20 LongBench datasets into two types: extraction tasks, which require extracting answers from the context (e.g., QA tasks evaluated with F1 or Accuracy), and generation tasks (e.g., summarization and code completion). For each category, we then compute the average scores obtained with Qwen and Mistral under varying cache budgets and eviction policies. Figure 2 highlights several key findings: 1) *Extraction tasks are generally less affected by compression*, as LLM performance with a compressed cache remains closer to that with a full cache; 2) *The performance gap among different eviction policies is greater on generation tasks.*; 3) *CAKE and LAVa outperform Ada-SnapKV and methods with fixed-layer budgets on generation tasks*, though CAKE performs significantly worse than Ada-SnapKV on extraction tasks with Mistral-7B. This suggests the importance of (dynamic) layer budget allocation for generation tasks. LAVa, however, consistently achieves top performance across both task types and language models.

5.3 Evaluation of Latency and Memory Peak

We evaluate LAVa's efficiency during LLM inference by analyzing peak memory usage and decoding latency on Mistral-7B-Instruct-v0.2, implemented with FlashAttention-2 (Dao, 2023). Our

	Single-Doc. QA				Multi-Doc. QA				Summarization				Few-shot Learning				Synthetic			Code		
	NrtvQA	Qasper	MF-en	MF-zh	HotpotQA	2WikiMQA	Musique	Dureader	GovReport	QMSum	VCSUM	MultiNews	TREC	TriviaQA	SAMSum	LSHT	PCount	PR-en	PR-zh	Lcc	RepoBench-P	Avg
Full Cache	26.77	32.34	49.63	48.42	43.43	27.89	18.61	30.85	32.92	24.54	15.04	27.20	71.00	86.23	43.41	39.00	2.81	86.56	89.75	55.29	52.55	45.07
											$\mathbb{B} = 128$HL											
PyramidKV	20.01	19.23	43.81	32.37	35.62	22.34	14.38	17.53	18.95	21.91	11.07	20.87	47.00	**85.34**	**40.21**	19.25	2.86	65.60	59.49	49.52	45.67	34.51
SnapKV	20.99	19.65	**45.04**	32.02	36.48	22.19	14.04	17.68	18.83	21.36	10.91	20.29	45.00	84.10	40.01	19.75	3.06	64.48	60.50	49.84	45.27	34.42
Ada-PyramidKV	20.21	20.80	43.82	33.65	37.21	22.99	14.93	18.06	**19.41**	22.02	11.16	20.97	52.00	83.93	39.97	20.00	2.81	**72.73**	72.89	51.00	46.62	36.22
Ada-SnapKV	20.61	20.56	44.03	**34.03**	36.39	23.66	**16.15**	17.82	19.21	21.73	11.25	20.35	50.00	84.32	39.82	19.75	3.87	69.11	70.52	50.21	46.85	35.82
CAKE	**21.01**	20.16	44.08	32.52	36.16	**23.89**	15.32	17.67	18.82	**22.62**	10.93	21.03	47.00	85.14	39.90	21.25	3.02	63.65	65.96	51.81	48.53	35.06
LAVa (Ours)	19.57	**21.11**	44.29	33.91	**38.29**	23.59	15.32	**18.56**	19.33	22.32	**11.42**	**21.07**	**53.50**	85.20	40.16	**21.75**	2.88	69.87	**74.75**	**51.94**	**48.92**	**36.74**
											$\mathbb{B} = 256$HL											
PyramidKV	20.79	22.74	45.90	35.72	38.63	24.02	15.97	18.99	21.61	22.34	11.02	22.24	58.00	84.06	40.52	22.75	2.96	74.70	83.83	51.85	48.86	38.23
SnapKV	21.39	22.15	46.50	34.77	**39.68**	25.01	14.86	19.11	21.61	23.04	11.46	22.67	57.00	85.04	40.81	23.25	3.18	76.49	83.60	51.99	49.42	38.49
Ada-PyramidKV	22.61	23.84	47.65	36.56	39.33	24.86	**17.22**	19.65	21.22	22.54	11.82	22.29	64.00	84.93	40.36	24.50	3.40	77.39	85.83	52.48	49.43	39.43
Ada-SnapKV	21.63	23.55	47.51	37.42	38.89	23.65	16.06	19.34	**21.98**	**23.21**	11.49	22.39	64.00	**86.33**	40.54	25.25	2.23	**77.44**	85.42	52.31	49.62	39.40
CAKE	21.37	23.40	46.84	35.02	38.10	24.50	14.81	19.40	21.59	22.77	11.32	22.68	55.00	85.46	**41.92**	24.75	2.96	75.66	**86.46**	**54.29**	51.38	38.84
LAVa (Ours)	**22.70**	**24.67**	**48.62**	**37.81**	**39.68**	**25.96**	16.77	**20.26**	21.92	22.48	**11.88**	**22.91**	**65.00**	85.24	41.28	**26.75**	2.88	76.76	85.75	54.17	**51.77**	**40.12**
											$\mathbb{B} = 512$HL											
PyramidKV	23.57	24.84	48.74	39.54	38.90	25.22	17.40	20.42	23.04	23.24	11.91	24.19	66.50	86.07	41.06	28.00	3.29	87.29	88.83	53.77	50.42	41.15
SnapKV	23.67	**28.08**	49.40	40.25	40.14	25.58	16.97	20.49	**23.75**	**23.69**	12.03	24.31	65.00	86.29	41.98	28.50	3.22	85.79	88.67	53.99	51.02	41.48
Ada-PyramidKV	24.37	27.30	48.01	40.88	39.75	25.96	**18.58**	20.90	23.59	23.33	12.07	24.04	67.50	**86.44**	**42.58**	31.50	3.38	85.88	89.67	54.15	51.30	41.89
Ada-SnapKV	24.63	27.48	48.90	41.28	39.84	26.33	18.26	20.91	23.59	23.51	12.27	24.32	67.50	86.38	42.34	32.50	2.98	**87.65**	89.17	54.39	51.03	42.11
CAKE	22.76	27.54	**49.47**	41.27	38.17	25.85	17.26	20.60	23.72	23.65	11.95	24.50	66.00	86.01	42.56	29.50	3.45	86.79	88.75	**56.40**	52.37	41.76
LAVa (Ours)	**25.01**	27.84	48.97	**42.14**	**40.95**	**26.88**	18.33	**21.12**	23.59	23.59	**12.28**	**24.51**	**68.50**	86.34	42.48	**33.50**	2.90	87.23	**89.83**	55.83	**52.85**	**42.59**
											$\mathbb{B} = 1024$HL											
PyramidKV	**25.62**	28.96	48.35	42.18	40.89	26.65	19.69	21.96	25.10	23.57	12.58	25.42	68.50	**86.30**	41.92	35.50	2.98	86.77	89.50	55.26	51.03	42.79
SnapKV	24.80	30.17	49.13	43.23	41.16	26.92	17.89	22.58	25.75	23.64	12.88	25.85	67.50	86.25	42.56	36.00	2.88	88.10	88.92	55.23	51.38	43.00
Ada-PyramidKV	24.98	29.92	47.97	41.43	40.83	26.98	19.42	22.45	25.46	23.58	12.94	25.61	68.50	**86.30**	**42.84**	35.50	2.89	88.18	89.25	54.51	51.32	42.90
Ada-SnapKV	24.84	29.99	**49.21**	42.55	41.00	**27.39**	19.23	23.23	25.89	**24.18**	13.13	25.85	69.00	86.23	**42.84**	36.25	2.90	**89.02**	**89.75**	55.38	51.93	43.34
CAKE	25.15	30.34	49.00	43.08	40.86	26.70	**19.93**	23.07	25.82	23.72	13.16	**26.05**	68.00	86.25	42.70	36.00	2.91	88.60	88.75	56.75	53.26	43.36
LAVa (Ours)	25.59	**31.21**	48.27	**43.43**	**41.92**	27.38	19.48	**23.48**	**26.06**	23.86	**13.38**	26.00	**70.00**	86.22	42.43	**38.00**	2.73	87.01	88.75	**57.31**	**53.28**	**43.65**

Table 2: Final comparison based on Mistral-7B-Instruct-v0.2 among 21 datasets of LongBench. (Note: The best result is highlighted in **bold**, and the second is in underline. Due to the negligible numerical values obtained from the passage count dataset, its results were excluded from the computation of the average scores.)

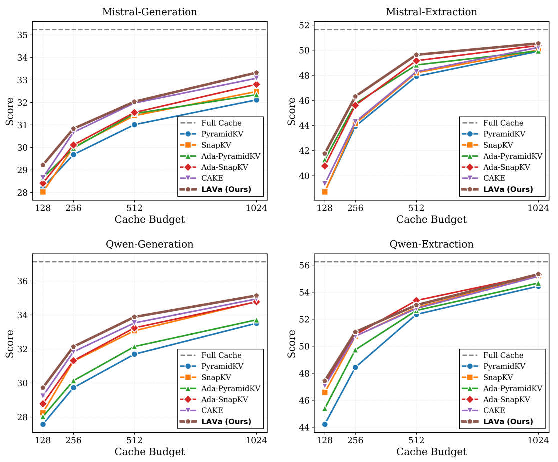

Figure 2: Results of generation and extraction tasks.

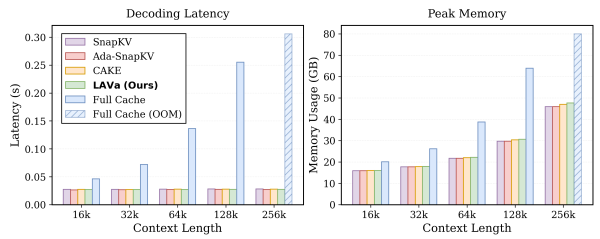

Figure 3: Peak memory usage and decoding latency in A800 80GB based on Mistral-7B-Instruct-v0.2.

comparison includes Full Cache, SnapKV, Ada-SnapKV and CAKE, all using allocation budget 1024HL. We set input at varying lengths while keeping the output length fixed at 128.

Decoding Latency. By analyzing the decoding latency in Figure 3, we observe that our scoring function and dynamic budget allocation introduce negligible decoding cost, achieving over a 9× speedup compared to Full Cache at a 128K context length. Notably, our method is easier to deploy than PyramidKV, Ada-PyramidKV, and CAKE, as these baselines require parameter tuning.

Peak Memory Usage. The peak memory usage of all methods generally increases with context length due to prefilling. Our method effectively maintains peak memory at a reasonable level, particularly compared to Full Cache, which encounters OOM issues at higher context lengths. CAKE and LAVa, both employing dynamic layer budgets, generally have slightly higher peak memory usage. Compared to CAKE, LAVa requires additional storage for the norms of head-wise value vectors, but this extra memory overhead remains minimal.

Theoretical Analysis. We provide the theoretical analysis of time complexity and memory usage in Appendix D. The time complexity and peak memory usage of SnapKV is $O(HN(Nd_h + wd_h + logB_{l,h}))$ and $O(HNd_h + LHB_{l,h}d_h)$, while that of LAVa is $O(HN(Nd_h + wd_h + d_h + logB_l)$ and $O(HNd_h + LHB_{l,h}d_h + LHB_{l,h}d_h)$. Setting

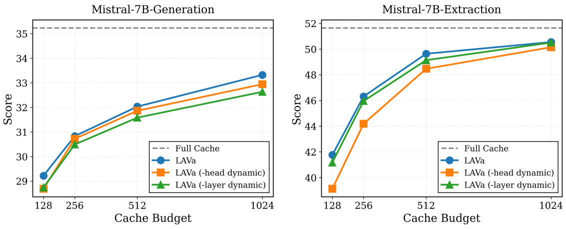

Figure 4: Ablation study on LongBench.

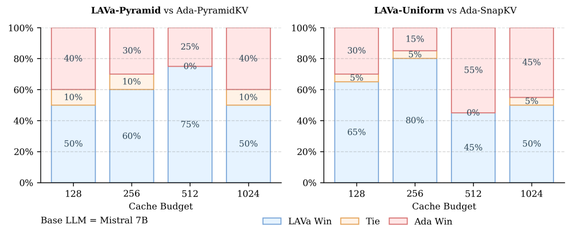

Figure 5: LaVa score vs AdaKV score on LongBench.

context length N as 10,000, head budget $B_{l,h}$ as 1024, the extra computation of LAVa compared to SnapKV is 0.01% and the extra memory usage is 0.6%, which is consistent with Figure 3.

5.4 Further Analysis

Dynamic Budget Allocation To examine the impact of dynamic budget allocation, we introduce two modifications: **LAVa (-layer dynamic)**, which enforces a uniform layer budget of $\mathbb{B}/L$, and **LAVa (-head dynamic)**, which fixes the head budget at $\mathcal{B}_l/H$ after dynamically determining the layer budget $\mathcal{B}_l$, performing head-wise cache eviction without cross-head comparisons. Results in Figure 4 demonstrate that dynamic budget allocation at both the head and layer levels is essential for performance. Furthermore, it reinforces the finding that dynamic layer budgets are essential for generation tasks, whereas dynamic head budgets play a crucial role in text extraction tasks. Detailed results are provided in Appendix D, where we also analyze the influence of different layer allocation approaches.

Analysis of LAVa Score. To validate the effectiveness of LAVa score, we replace our dynamic layer budgets with fixed ones with PyramidKV or Uniform allocation. For different total budgets, we then compare LAVa-Pyramid with Ada-PyramidKV and LAVa-Uniform with AdaKV on LongBench. For each comparison, we count the number of tasks in LongBench where one method outperforms the other. Figure 5 presents the final winning rates. The results show that our scoring function yields a significantly higher number of wins in most cases, validating its effectiveness.

6 Related Work

Recently, various KV Cache compression methods have been proposed, leveraging different policies such as recency (Xiao et al., 2024), accumulated attention scores (Zhang et al., 2023), last-token attention scores (Oren et al., 2024), and recent attention scores (Li et al., 2024; Dai et al., 2024). While most approaches assume a uniform budget, recent efforts have been made for dynamic budget allocation across layers (Qin et al., 2025) and heads (Feng et al., 2024). Some methods aim at layer-dependent budgets but fix the patterns across all samples (Cai et al., 2024; Yang et al., 2024). In general, KV Cache eviction and budget allocation are typically treated as separate problems, requiring a combination of independent strategies. In contrast, we develop a principled framework based on information loss in the residual stream and propose a unified method for both cache compression and dynamic budget allocation.

Closely related to LAVa is (Feng et al., 2025, 2024), which aims at minimizing the layer output perturbation. However, this study only applies the derived metric locally for head budget allocation. In contrast, we propose a metric for layer-wise cache eviction with dynamic layer budgets.

7 Conclusion

This paper provided a comprehensive of current KV Cache compression into a unified framework, grounded in the principle of minimizing information loss in Transformer residual streams. By analyzing the *Layer Attention Output Loss*, we proposed LAVa, a novel layer-wise compression method that enables fully dynamic head and layer budget allocation. Our experiments demonstrate that *dynamic layer budgets are crucial for generation tasks*, whereas *dynamic head budgets are important for extraction tasks*. As a fully dynamic compression method, LAVa consistently maintains top performance across task types and LLM architectures, while achieving the same speedup of 9× with 128K context length compared to full cache.

Future directions include exploring new compression algorithms based on our framework, as well as extending our framework for model compression. By advancing efficient methods for LLMs, our work contributes to making LLM more accessible and scalable for diverse applications.

Limitations

There are several limitations to our work. While we propose a unified framework with multiple optimization opportunities, our theoretical analysis and experiments focus on only one direction. Although LAVa's simplicity is a key advantage, other approaches should be explored to further close the performance gap with a full-cache setup, particularly for generation tasks. Additionally, further research is needed to better understand why dynamic layer budget is crucial for generation tasks. Lastly, apart from FlashAttention-2 (Dao, 2023), our method has not yet been integrated into other widely used inference frameworks, such as vLLM (Kwon et al., 2023). We believe that such integration is essential for broader adoption and real-world deployment of our algorithm.

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 *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Anthropic and et al. The claude 3 model family: Opus, sonnet, haiku.

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

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

Guanzheng Chen, Xin Li, Michael Shieh, and Lidong Bing. 2025. LongPO: Long context self-evolution of large language models through short-to-long preference optimization. In *The Thirteenth International Conference on Learning Representations*.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality.

Jincheng Dai, Zhuowei Huang, Haiyun Jiang, Chen Chen, Deng Cai, Wei Bi, and Shuming Shi. 2024. Corm: Cache optimization with recent message for large language model inference. *Preprint*, arXiv:2404.15949.

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

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*.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S Kevin Zhou. 2025. Identify critical kv cache in llm inference from an output perturbation perspective. *Preprint*, arXiv:2502.03805.

Javier Ferrando and Elena Voita. 2024. Information flow routes: Automatically interpreting language models at scale. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 17432–17445, Miami, Florida, USA. Association for Computational Linguistics.

Yao Fu, Rameswar Panda, Xinyao Niu, Xiang Yue, Hannaneh Hajishirzi, Yoon Kim, and Hao Peng. 2024. Data engineering for scaling language models to 128k context. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. Longcoder: A long-range pretrained language model for code completion. In *International Conference on Machine Learning*.

Zhiyu Guo, Hidetaka Kamigaito, and Taro Watanabe. 2024. Attention score is not all you need for token importance indicator in KV cache reduction: Value also matters. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 21158–21166, Miami, Florida, USA. Association for Computational Linguistics.

Roger A Horn and Charles R Johnson. 2012. *Matrix analysis*. Cambridge university press.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. 2024. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *Preprint*, arXiv:2310.06825.

Ehsan Kamalloo, Nouha Dziri, Charles Clarke, and Davood Rafiei. 2023. Evaluating open-domain question answering in the era of large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5591–5606, Toronto, Canada. Association for Computational Linguistics.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA. Association for Computing Machinery.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. SnapKV: LLM knows what you are looking for before generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

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

OpenAI and et al. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Matanel Oren, Michael Hassid, Nir Yarden, Yossi Adi, and Roy Schwartz. 2024. Transformers are multistate rnns. *arXiv preprint arXiv:2401.06104*.

Guanghui Qin, Corby Rosset, Ethan Chau, Nikhil Rao, and Benjamin Van Durme. 2024a. Dodo: Dynamic contextual compression for decoder-only lms. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9961–9975.

Guanghui Qin, Corby Rosset, Ethan C. Chau, Nikhil Rao, and Benjamin Van Durme. 2024b. Nugget 2d: Dynamic contextual compression for scaling decoder-only language models.

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. In *The Thirteenth International Conference on Learning Representations*.

Qwen and et al. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems*.

Wenhao Wu, Yizhong Wang, Guangxuan Xiao, Hao Peng, and Yao Fu. 2025. Retrieval head mechanistically explains long-context factuality. In *The Thirteenth International Conference on Learning Representations*.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, junxian guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. 2025. Duoattention: Efficient long-context LLM inference with retrieval and streaming heads. In *The Thirteenth International Conference on Learning Representations*.

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

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. Association for Computational Linguistics.

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

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

A Extension of The Information Flow of LLM Decoding Process with KV Cache

KV Cache is initialized at *prefilling* stage, which basically computes the Key and Value for tokens in the initial prompts in the standard way (Vaswani, 2017). In the following, we assume that there exists a KV Cache of $(N-1)$ previous tokens and demonstrate how decoding is performed at step-N.

Notation Table The LLM has L layers, each has H heads. The model and head dimensions are d and $d_h = d/H$; K_l, V_l are the KV Cache for the l-th layer up to the current time step (the N-th token), which are of $[H, (N-1), d_h]$ sizes. The notations for the theoretical analysis are listed in Table 3.

Notation	Explanation	Notation	Explanation
N	Current token length	$A_{l,h}^N[i]$	Attention weight of position i at layer l, head h and step N
N_e	Expected token length	y_l^N	Attention output of layer l and step N
L	Total number of layers	$\hat{y}_l^N$	Modified attention output of layer l and step N after eviction
H	Total number of heads per layer	p	Logits after last layer for next token
l	Layer index, $l \in [L]$	$\hat{p}$	Modified logits after last layer for next token after eviction
h	Head index, $h \in [H]$	$\mathcal{P}$	Information loss function of Transformer residual streams
d	The model embedding dimension	w	Sliding window size
d_h	The head embedding dimension $d_h = d/H$	$\mathcal{B}_{l,h}$	Budget for head h of layer l
x_l^N	The input hidden states of step N and layer l	$\mathcal{B}_l$	Budget for layer l
Q_l^N	The query vector of step N and layer l	$\mathbb{B}$	Fixed total budget for KV Cache, $\mathbb{B} = \sum_{l \in [L]} \mathcal{B}_l$
K_l^N	The key vector of step N and layer l	$s_{l,h}[i]$	Score of position i at layer l and head h
V_l^N	The value vector of step N and layer l	e_l	The uncertainty of layer l for dynamic layer budget allocation
$K_{l,h}$	Key cache of layer l and head h	$\mathcal{I}_{l,h}$	Attention mask for the head h of layer l, $\mathcal{I}_{l,h} \in [1,0]^N$
$V_{l,h}$	Value cache of layer l and head h	$\mathcal{I}$	Attention mask $\mathcal{I} \in [1,0]^{L \times H \times N}$

Table 3: Notation table.

Decoding Process According to (Ferrando and Voita, 2024), the decoding process of large language models (LLMs) can be viewed as a series of operations on the current *residual stream*, as illustrated in Figure 1. In each layer, information is read from the residual stream, updated, and then written back. Specifically, supposing that x_l^N is the current input for layer l, we first calculate the corresponding Q_l^N, K_l^N, V_l^N as follows:

$$Q_l^N = x_l^N W_l^Q; K_l^N = x_l^N W_l^K; V_l^N = x_l^N W_l^V$$

where Q_l^N, K_l^N, V_l^N are of size $(H \times 1 \times d_h)$, containing H head-wise caches. The layer-wise KV Cache is then updated as follows:

$$K_l = Cat[K_l, K_l^N], V_l = Cat[V_l, V_l^N]$$

where K_l, V_l are tensors of size $(H \times N \times d_h)$, and Cat indicates the concatenation operation.We then calculate the attention scores of step-N for layer-l:

$$A_l^N = Cat_{h \in [H]} \left(A_{l,h}^N \right)$$

where $A_{l,h}^N = Softmax(\frac{Q_{l,h}^N K_{l,h}}{\sqrt{d_h}})$. Here, textbf$A_{l,h}^N[i]$ *indicates how much the token at step-N (the N-th token) attends to the i-th token ($i <= N$).* Layer-l attention output is calculated as follows:

$$y_l^N = Cat_{h \in [H]}(A_{l,h}^N V_{l,h}) W_l^O \in \mathbb{R}^{1 \times d}$$

where $W_l^O \in \mathbb{R}^{d \times d}$. The layer output x_{l+1}^N, which is also the input for the layer-$(l+1)$, is calculated as $x_{l+1}^N = y_l^N + FFN(y_l^N)$.

In the last layer, we exploits an un-embedding layer ($W^M \in \mathbb{R}^{d \times |\mathcal{V}|}$) to get the probability vector p for next token sampling:

$$p^N = \left(y_L^N + FFN(y_L^N \right) W^M \quad (8)$$

Head-wise vs Layer-wise Cache Current query matrix and KV Cache on head h of layer l are :

$$Q_{l,h}^N = Q_l^N[:, d_h * h : d_h * (h+1)] \in \mathbb{R}^{1 \times d_h} \quad (9)$$

$$K_{l,h} = K_l[:, d_h * h : d_h * (h+1)], \quad (10)$$

$$V_{l,h} = V_l[:, d_h * h : d_h * (h+1)] \in \mathbb{R}^{N \times d_h} \quad (11)$$

Henc, the layer-wise KV Cache can be treated as concatenation of head-wise elements where we just change the order of dimensions:

$$K_l = Cat_{h \in [H]}[K_{l,h}] \in \mathbb{R}^{H \times N \times d_h}, \quad (12)$$

$$V_l = Cat_{h \in [H]}[V_{l,h}] \in \mathbb{R}^{H \times N \times d_h} \quad (13)$$

And the same to the query matrix:

$$Q_l^N = Cat_{h \in [H]}[Q_{l,h}^N] \in \mathbb{R}^{H \times 1 \times d_h} \quad (14)$$

B Extension of A Principled Framework for KV Cache Eviction based on Information Loss

The unified problem for budget allocation and cache eviction can be defined as follows:

$$\min_{\mathcal{I}, \mathcal{B}} \mathcal{P}(\mathbf{x}_1^{1 \ldots N}, \mathcal{I}, \mathcal{B}) \quad (15)$$

$$\text{st.} \sum_{i \in [N]} \mathcal{I}_{l,h}[i] = \mathcal{B}_{l,h};$$

$$\sum_{h \in [H]} \mathcal{B}_{l,h} = \mathcal{B}_l; \sum_{l \in [L]} \mathcal{B}_l = \mathbb{B}$$

$$\mathcal{I}_{l,h}[k] = 1 \text{ , } \forall l, h; \text{ and } \forall k \in [N-w, N]$$

The optimization problem in Eq. 15 is infeasible to solve for several reasons. We can instead search for a scoring function s, where $s_{l,h}[i]$ assigns an importance score to token i at layer l and head h. This scoring function allows us to greedily choose the least important entries to be masked until the budget is met $\mathcal{I} = Select(s, \mathcal{B})$. Bringing everything together, we arrive at the following (surrogate) optimization problem:

$$\min_{\mathcal{B}, s \in \mathcal{F}} \mathcal{P}(\mathbf{x}_1^{1\ldots N}, s, \mathbb{B}) \tag{16}$$

Current various kv cache eviction methods can be adapted into our framework, just defining several significant functions and parameters (including $P, \mathcal{I}, \mathcal{B}$ and s) and introducing additional constraints, which will result in suboptimal performance. In addition, they adopt many heuristic techniques based on observations to simplify the problem. The full summarization of how existeing methods can be formalized within our framework is presented in Table 4.

H2O. (Zhang et al., 2023) Allocation budgets $\mathcal{B}$ are all fixed before generation. The budgets of all layers are the same and the budgets of all heads are also the same.

$$\mathcal{B}_{l,h} = \frac{\mathcal{B}}{HL} \tag{17}$$

H2O uses **head attention loss** and adopt accumulated attention scores as score function.

$$s_{l,h}[i] = \sum_{j=i+1}^{N} A_{l,h}^{j}[i], \mathcal{I}_{l,h} = Select(s_{l,h}, \mathcal{B}_{l,h}) \tag{18}$$

H2O claimed that the accumulated attention score can preserve the future attention pattern better. This technique is heuristic and based on observations of experiments in several methods like H2O and SnapKV (Li et al., 2024), but it is valid and actually can improve the performance, mitigating the impact of absolutism of only current attention scores (Oren et al., 2024).

TOVA. (Oren et al., 2024) The difference between TOVA and H2O is that TOVA uses current attention scores as score function.

$$s_{l,h}[i] = A_{l,h}^{N}[i], \mathcal{I}_{l,h} = Select(s_{l,h}, \mathcal{B}_{l,h}) \tag{19}$$

SnapKV. (Li et al., 2024) The difference between SnapKV and H2O is that SnapKV uses recent attention scores as score function, which means SnapKV only utilizes tokens within sliding window to calculate accumulated attention scores. We set sliding window size as w:

$$\begin{aligned} s_{l,h}[i] &= \sum_{j=N-w}^{N} A_{l,h}^{j}[i] \\ \mathcal{I}_{l,h} &= Select(s_{l,h}, \mathcal{B}_{l,h}) \end{aligned} \tag{20}$$

SnapKV claims that the accumulated attention scores of the recent sliding window is enough to represent the significance of tokens. Furthermore, SnapKV adopts pooling operation to preserve the completeness of the information. In our view, better protecting the coherence of the text is the reason for the effectiveness of pooling operation.

PyramidKV. (Cai et al., 2024) The difference between PyramidKV and SnapKV is that considering the different significance of layers in the long-context setting, PyramidKV set the budgets of layers in a descending order like a pyramid. It uses a hyper-parameter β to control the shape of pyramid.

$$\begin{aligned} \mathcal{B}_{L-1} = \frac{\mathcal{B}}{\beta * L}, \mathcal{B}_0 &= \frac{2 * \mathcal{B}}{L} - \mathcal{B}_{L-1} \\ \mathcal{B}_l = \mathcal{B}_0 - \frac{\mathcal{B}_{L-1} - \mathcal{B}_0}{L-1} * l \end{aligned} \tag{21}$$

And the budgets of heads in one layer are the same: $\mathcal{B}_{l,h} = \frac{\mathcal{B}_l}{H}$.

Hence, compared with SnapKV, PyramidKV consider about different budgets of layers in a heuristic way.

CAKE. (Qin et al., 2025) Allocation budgets $\mathcal{B}$ are generated through the online prefilling stage. All heads of one layer have the same budget. So CAKE do not consider the level of head (such as using mean information across heads).

Considering spatial and temporal information, CAKE allocates different budgets to different layers. And not adopting the fixed pattern like PyramidKV, CAKE claims that for different samples, the allocation pattern also needs to be adapted. It defines functions of spatial and temporal information for one layer l, the spatial information function $\mathcal{H}$ is formed as entropy of attention scores (larger values means more even distribution) and the temporal information function $\mathcal{V}$ (larger values means more distribution shift) is formed as variance of attention scores ($A^{(n)}$ means the attention scores

Methods	Budgets $\mathcal{B}_{l,h}$	Budgets $\mathcal{B}_l$	Scoring Function	Loss
H2O (Zhang et al., 2023)	$\mathcal{B}_l/H$	$\mathbb{B}/L$	Accumulated attention scores $s_{l,h}[i] = \sum_{j=i+1}^{N} A_{l,h}^{j}[i]$	Head Attention
SnapKV (Li et al., 2024)	$\mathcal{B}_l/H$	$\mathbb{B}/L$	Recent attention scores $s_{l,h}[i] = \frac{1}{w}\sum_{j=N-w}^{N} A_{l,h}^{j}[i], \forall i < N-w$	
TOVA (Oren et al., 2024)	$\mathcal{B}_l/H$	$\mathbb{B}/L$	Last-token attention scores $s_{l,h}[i] = A_{l,h}^{N}[i]$	
CAKE (Qin et al., 2025)	$\mathcal{B}_l/H$	Dynamic	Recent attention scores + attention shifts $s_{l,h}[i] = \gamma \text{VAR}_{j=N-w}^{N}([A_{l,h}^{j}[i]]) + \frac{1}{w}\sum_{j=N-w}^{N} A_{l,h}^{j}[i], \forall i < N-w$	
VATP (Guo et al., 2024)	$\mathcal{B}_l/H$	$\mathbb{B}/L$	Recent attention scores + value vectors $s_{l,h}[i] = \frac{\lVert V_{l,h}[i]\rVert_1}{w}\sum_{j=N-w}^{N} A_{l,h}^{j}[i]$	Head Attention Output
Dodo (Qin et al., 2024a)	Dynamic	$\mathbb{B}/L$	Neural Network (LoRA)	Logits
DuoAttention (Xiao et al., 2025)	w or full	-	Head classifier (retrieval vs non-retrieval)	Layer Attention Output
AdaKV (Feng et al., 2024)	Dynamic	Fixed	Recent attention scores	
LAVa (Ours)	Dynamic	Dynamic	Recent attention scores + value vectors $s_{l,h}[i] = \frac{max_k \lVert V_{l,h}[k]\rVert_1}{w}\sum_{j=N-w}^{N} A_{l,h}^{j}[i]$	

Table 4: Comparison between different methods; Dodo and DuoAttention require training; The layer cache budget $\mathcal{B}_l$ of AdaKV is based on the method it is integrated with.

distribution in the n-th step of prefilling stage):

$$\mathcal{H}_l = -\sum_{j=1}^{N} A_l^j \log(A_l^j),$$

$$\mathcal{V}_l = \sum_{j=1}^{N} \text{VAR}([A_l^t[j]]^{t\in[j,N]}) \tag{22}$$

Then CAKE uses these two functions to determine the budget of layers, where γ_1 and γ_2 are two hyper-parameters to control the influence of two functions:

$$\mathcal{P}_l = \mathcal{H}_l^{\frac{1}{\gamma_1}} \mathcal{V}_l^{\frac{1}{\gamma_2}}, \mathcal{B}_l = \frac{\mathcal{P}_l}{\sum^{l\in[L]} \mathcal{P}_l}\mathcal{B}, \mathcal{B}_{l,h} = \frac{\mathcal{B}_l}{H} \tag{23}$$

CAKE also uses **head attention loss** function as optimization objective but it also introduces temporal information into score function of SnapKV. It adopts variance to represent the distribution shift of attention scores for the same token. Let γ be a hyper-parameter to control the influence of temporal information, and w as the sliding window size, CaKE score is:

$$s_{l,h}[i] = \sum_{j=N-w}^{N} A_{l,h}^{j}[i] + \gamma \text{VAR}([A_{l,h}^{t}[i]]^{t\in[i,N]})$$

$$\mathcal{I}_{l,h} = Select(s_{l,h}, \mathcal{B}_{l,h}) \tag{24}$$

AdaKV. (Feng et al., 2024) The algorithm of AdaKV is based on other methods. It adopts **layer attention output loss** function but not conduct real training. Deriving the upper bound of output loss (as shown in Eq. 25 where $C = max_{h\in[H]}\lVert {W_{l,h}^{O}}^{T} V_{l,h}^{T}\rVert_1$), AdaKV obtains the insight that allocating different budgets to heads of one layer based on the score function just considering about information within attention scores can preserve the performance of model further.

$$\lVert y_l - \hat{y}_l\rVert_1 \leq 2C \sum_{h\in[H]} (\sum_{i\in[N]} A_{l,h}^{N}[i](1-\mathcal{I}_{l,h}[i])) \tag{25}$$

We set $\hat{s}_l$ as the topk results of all $s_{l,h}, h \in [H]$, the budget of one head h can be calculated by:

$$\mathcal{B}_{l,h} = Num(\hat{s}_{l,h}), \hat{s}_l, \mathcal{I}_l = Select(s_{l,h}, \mathcal{B}_{l,h}) \tag{26}$$

AdaKV combines this insight with SnapKV and PyramidKV for better results. So the score function of AdaKV is the same as Eq. 20. However, the bound of AdaKV ignores the influence of value information and just use the max information, which will make the bound too loose. Our framework about output loss is motivated by this research and we conduct some modification and further studies. For the details and how to derive upper bound of output loss, refer to Section 4.

DuoAttention. (Xiao et al., 2025) DuoAttention uses **layer attention output loss** function as optimization objective. Unlike H2O and TOVA, attention mask $\mathcal{I}$ of DuoAttention is constraint to a pattern combined with sink and recent tokens based on allocation budgets $\mathcal{B}$, which means score function s id for tokens are not needed. Here sink tokens means several initial tokens in prompt defined by StreamingLLM (Xiao et al., 2024).

$$\mathcal{I}_{l,h}[i] = \begin{cases} 1 & \text{if position } k \text{ is sink or recent, } k \in [N] \\ 0 & \text{otherwise, evict } K_{l,h}[k] \text{ and } V_{l,h}[k] \end{cases} \tag{27}$$

DuoAttention adopts real optimization method and needs training based on 2-norm of output loss function. The optimization result is to determine the allocation budgets $\mathcal{B}$. In detail, it determines which head was allocated with full budget and which head was allocated with a compressed budget. So besides $\mathcal{I}$ and $\mathcal{B}$, DuoAttention introduces a parameter α to be optimized and finally determines the different functions of heads, including Retrieval Heads (Wu et al., 2025) and Streaming Heads. We define $\hat{w}$ as the numbers of sink and recent tokens.

$$\mathcal{B}_{l,h} = \begin{cases} n & \text{if head } h \text{ of layer } l \text{ is Retrieval Head} \\ \hat{w} & \text{otherwise, Streaming Head} \end{cases} \tag{28}$$

Dodo. (Qin et al., 2024a) Dodo uses **logit loss** function as optimization objective. But not adopting a predefined rule for attention mask $\mathcal{I}$, Dodo uses a score function ϕ implemented by LoRA (Hu et al., 2021) adapters to determine the attention mask for tokens, which is trained along with logits loss. Logits loss is defined by loss of future expected tokens which are not pratical. So Dodo converts the expected tokens into past tokens and the loss function can be formalized as:

$$P(\mathcal{I}, \mathcal{B}) = \sum_{i \in [N]} CE(p, \hat{p})^i \tag{29}$$

The score function ϕ is trained via this loss function and finally determines which tokens will be preserved. The cache budget $\mathcal{B}$ for all heads and layers are the same. Besides, Dodo merges the information within tokens evicted into the preserved tokens similar to KV Cache merging methods.

VATP. (Guo et al., 2024) The difference between LAVa and VATP is shown in Table 4 and explained as follows: (1) VATP directly multiplies each token's value norm with attention scores. In contrast, LAVa calculates the maximum value norm, which serves as scaling factors for heads; (2) VATP has fixed head and layer budgets, while LAVa is totally dynamic. The deeper difference, however, lies in how the two scores are developed. VATP comes with an intuition of "Value also matters" but lacks theoretical analysis. We independently derive from layer attention output with a complete reasoning process: starting from layer attention output, deriving the upper bound, getting an approximate score in greedy solution, smoothing it out based on multiple residual stream.

Budgets	128	256	512	1024
SnapKV	34.42	38.49	41.48	43.00
+VATP	35.34	39.41	41.93	43.32
LAVa	**36.74**	**40.12**	**42.59**	**43.65**
-layer dynamic	36.20	39.77	42.11	43.35

Table 5: Comparison between VATP and LAVa.

This reasoning is very important. As we start from the layer point of view, we can see that such scores can be used to compare entries across heads for layer-wise KV Cache eviction. And we argue that doing so could reduce the information loss at layer attention output. The reasoning process shows what approximation we make and gives room for future improvement.

To validate our elaboration, we compares three configurations: (1) VATP integrated with SnapKV, (2) standard LAVa, and (3) LAVa without dynamic layer budgeting based on Mistral-7B-Instruct-v0.2 in LongBench. The results in Table 5 demonstrate that while VATP shows improvement over baseline SnapKV, it consistently underperforms compared to both LAVa and LAVa (-layer dynamic). From the computational perspective, VATP incurs similar overhead to LAVa(refer to Appendix D, yet delivers suboptimal performance. This verifies our claim that intuition and a theoretical analysis help you get to a more optimal solution.

C Extension of LAVa: Layer-wise Cache Eviction with Dynamic Budget Allocation

Details of Lemma 1. We define and derive the **Layer Attention Output Loss** in this lemma.

Lemma 1. Based on the L_p norm, the layer attention output loss due to the attention mask $\mathcal{I}$ is measured for layer-l at the current (N-th) decoding step as follows:

$$\mathcal{P}(\mathbf{x}_1^{1\ldots N}, \mathcal{I}, \mathcal{B}) = \|y_l^N - \hat{y}_l^N\|_p \tag{30}$$
$$= \left\| Cat_h \left[(A_{l,h}^N - \frac{A_{l,h}^N \odot \mathcal{I}_{l,h}}{\|A_{l,h}^N \odot \mathcal{I}_{l,h}\|_1}) V_{l,h} \right] W_l^O \right\|_p$$

where $\odot$ indicates element-wise multiplication and $\hat{y}_l^N = Cat_h(\hat{A}_{l,h}^N V_{l,h}) W_l^O$
As we mentioned above:

$$\begin{aligned} y_l^N &= Cat_{h\in[H]}(A_{l,h}^N V_{l,h}) W_l^O \\ \hat{y}_l^N &= Cat_{h\in[H}(\hat{A}_{l,h}^N V_{l,h}) W_l^O \end{aligned} \tag{31}$$

And based on the definition of attention mask $\mathcal{I}$, the attention weights after eviction can be calculated as:

$$\hat{A}_{l,h}^N = Softmax(\frac{-\inf \odot (\mathbf{1} - \mathcal{I}_{l,h}) + Q_{l,h}^N K_{l,h}^T}{\sqrt{d_h}}) \tag{32}$$

Hence, Lemma 31 is equal to (Temporarily ignoring the superscript N):

$$\hat{A}_{l,h} = \frac{A_{l,h} \odot \mathcal{I}_{l,h}}{\|A_{l,h} \odot \mathcal{I}_{l,h}\|_1} \tag{33}$$

This theorem has been proved by AdaKV (Feng et al., 2024), so we will not elaborate further here.

Proof of Theorem 1. Then we drive the **upper bound** of Layer Attention Output Loss and give this theorem.

Theorem 1. The L_1 norm of layer attention output loss can be bounded by:

$$\|y_l - \hat{y}_l\|_1 \tag{34}$$
$$\leq 2\hat{C} \sum^{h\in[H]} \bar{V}_{l,h} (\sum^{k\in[N]} A_{l,h}^N[k](1 - \mathcal{I}_{l,h}[k]))$$

where $\bar{V}_{l,h} = max_{k\in[N]} \|V_{l,h}[k]\|_1$ and $\hat{C} = \|W_l^{O^T}\|_1$ is a constant, which is independent of any head or token within layer-l.

Proof. First we need to introduce a lemma:

Lemma 2. *Given a vector $x \in \mathbb{R}^{1\times m}$ and a matrix $W \in \mathbb{R}^{m\times n}$, we can get the relationship between matrix norm and vector norm:*

$$\|xW\|_p \leq \|x\|_p \|W^T\|_p \tag{35}$$

$\|xW\|_p$ and $\|x\|_p$ are vector p-norm, $\|W^T\|_p$ is matrix p-norm which is calculated by the largest sum of column absolute value.

This lemma is derived from Horn and Johnson (2012). Then we can obtain (Temporarily ignoring the superscript N):

$$\|y_l - \hat{y}_l\|_1$$
$$\leq \|Cat_h[(A_{l,h} - \frac{A_{l,h} \odot \mathcal{I}_{l,h}}{\|A_{l,h} \odot \mathcal{I}_{l,h}\|_1}) V_{l,h}]\|_1 \|W_l^{O^T}\|_1 \tag{36}$$

We set $\|W_l^{O^T}\|_1$ as $\hat{C}$ because it is the constant model parameter. Then we know that and set:

$$G_{l,h} = (A_{l,h} - \frac{A_{l,h} \odot \mathcal{I}_{l,h}}{\|A_{l,h} \odot \mathcal{I}_{l,h}\|_1}) V_{l,h} \in \mathbb{R}^{1\times d_h} \tag{37}$$

Thus $\|Cat^{h\in[H]}[G_{l,h}]\|_1$ is the vector 1-norm of a vector $\in \mathbb{R}^{1\times(d_h * H)}$. According to the definition of vector 1-norm, we can transform cat operation to sum and continue derivation based on Theorem 2:

$$\|y_l - \hat{y}_l\|_1$$
$$\leq \hat{C} \|Cat_{h\in[H]}[(A_{l,h} - \frac{A_{l,h} \odot \mathcal{I}_{l,h}}{\|A_{l,h} \odot \mathcal{I}_{l,h}\|_1}) V_{l,h}]\|_1$$
$$= \hat{C} \sum_{h\in[H]} \|(A_{l,h} - \frac{A_{l,h} \odot \mathcal{I}_{l,h}}{\|A_{l,h} \odot \mathcal{I}_{l,h}\|_1}) V_{l,h}\|_1$$
$$\leq \hat{C} \sum_{h\in[H]} (\|A_{l,h} - \frac{A_{l,h} \odot \mathcal{I}_{l,h}}{\|A_{l,h} \odot \mathcal{I}_{l,h}\|_1}\|_1 \|V_{l,h}^T\|_1) \tag{38}$$

Next we will prove that $\|A_{l,h} - \frac{A_{l,h} \odot \mathcal{I}_{l,h}}{\|A_{l,h} \odot \mathcal{I}_{l,h}\|_1}\|_1 = 2 \sum_{if \mathcal{I}_{l,h}[i]=0}^{i\in[N]} A_{l,h}[i]$.

Let $\|A_{l,h} \odot \mathcal{I}_{l,h}\|_1 = \sum_{i\in[N]} \mathcal{I}_{l,h}[i] A_{l,h}[i] = \sum_{if \mathcal{I}_{l,h}[i]=1}^{i\in[N]} A_{l,h}[i]$ as $F \in (0, 1]$:

$$
\begin{aligned}
&\|A_{l,h} - \frac{A_{l,h} \odot \mathcal{I}_{l,h}}{\|A_{l,h} \odot \mathcal{I}_{l,h}\|_1}\|_1 = \|\frac{F - \mathcal{I}_{l,h}}{F} \odot A_{l,h}\|_1 \\
&= \sum_{i \in [N]} |\frac{(F - \mathcal{I}_{l,h}[i]) A_{l,h}[i]}{F}| \\
&= \sum_{if \mathcal{I}_{l,h}[i]=0}^{i \in [N]} A_{l,h}[i] + \sum_{if \mathcal{I}_{l,h}[i]=1}^{i \in [N]} \frac{(1-F) A_{l,h}[i]}{F} \\
&= \sum_{if \mathcal{I}_{l,h}[i]=0}^{i \in [N]} A_{l,h}[i] + \frac{\sum_{if \mathcal{I}_{l,h}[i]=1}^{i \in [N]} A_{l,h}[i]}{F} \\
&- \sum_{if \mathcal{I}_{l,h}[i]=1}^{i \in [N]} A_{l,h}[i] \\
&= \sum_{if \mathcal{I}_{l,h}[i]=0}^{i \in [N]} A_{l,h}[i] + 1 - \sum_{if \mathcal{I}_{l,h}[i]=1}^{i \in [N]} A_{l,h}[i] \\
&= 2 \sum_{if \mathcal{I}_{l,h}[i]=0}^{i \in [N]} A_{l,h}[i]
\end{aligned}
\tag{39}
$$

Then based on the definition of matrix 1-norm and $\|V_{l,h}^T\|_1 \in \mathbb{R}^{d_h \times N}$, we can calculate this as the largest sum of row absolute value of $V_{l,h} \in \mathbb{R}^{N \times d_h}$, which is equals to the largest vector 1-norm of V value of previous tokens, formalized as:

$$
\bar{V}_{l,h} = \|V_{l,h}^T\|_1 = max_{k \in [N]} \|V_{l,h}[k]\|_1 \tag{40}
$$

Now we can obtain:

$$
\begin{aligned}
&\|y_l - \hat{y}_l\|_1 \qquad (41) \\
&\leq 2\hat{C} \sum_{h \in [H]} (\sum_{if \mathcal{I}_{l,h}[i]=0}^{i \in [N]} A_{l,h}^N[i] \|V_{l,h}^T\|_1) \\
&= 2\hat{C} \sum_{h \in [H]} (\sum_{i \in [N]} A_{l,h}^N[i] \bar{V}_{l,h} (1 - \mathcal{I}_{l,h}[i]))
\end{aligned}
$$

Here the proof is done. □

Potential Future Work. Building on our framework, multiple research directions can be further explored. One possible question is whether the *Layer Output Loss*, which takes into account the FFN layer, should be considered. The interaction between the FFN layer and the layer attention output determines what information a layer writes to the residual stream (Ferrando and Voita, 2024). In other words, certain tokens in past residual streams may play a crucial role in activating the layer's knowledge within the FFN. Accounting for these interactions could reduce performance loss, yet the challenge lies in how to do so efficiently.

Another potential avenue is formulating the problem as an online reinforcement learning (RL) task, where the objective is to optimize the policy (i.e., the scoring function) to maximize the expected reward. Here, the expected reward can be cast as minimizing the expected loss in future residual streams, not just the past ones. This direction is potential for the cache-offload and retrieval problem, where we need to decide which parts of the cache to offload to CPU or retrieve from CPU while maintaining the communication cost.

Additionally, this framework could be extended to model pruning, not just masking tokens but also selectively masking model parameters to minimize information flow while preserving efficiency.

D Extension of Experiments

Implementation Details. For SnapKV and Ada-SnapKV, no additional hyperparameters are required. However, for PyramidKV, we must adjust the parameter β to control the shape of the cache budget pyramid. We set β to (5, 10, 20) and select the best-performing result, the same approach to Ada-PyramidKV. For CAKE, three parameters require tuning: γ_1 and γ_2 for layer budget allocation, and γ_3 for the scoring function, as explained in Appendix B. Based on recommendations from (Qin et al., 2025), we set $1/\gamma_1$ to (0.2, 0.3, 0.5, 1, 2), $1/\gamma_2$ to (0.2, 0.3, 0.5, 1, 2), and γ_3 to (0, 5, 10, 200). We then evaluate different combinations and select the one that yields the best overall performance.

Pooling operators, such as max pooling or average pooling, can be applied to token score vectors to smooth score variations across adjacent tokens (Li et al., 2024; Cai et al., 2024; Qin et al., 2025). This strategy is also employed in the implementation of LAVa and all the baselines. For pooling operation, for all methods, we adopt maxpool function and set kernel size as 7.

Results of LAVa in LongBench. The results of Qwen2.5-7B-Instruct are listed in Table 6. The results of Qwen2.5-14B-Instruct and Qwen2.5-32B-Instruct are in Table 7. From all these results, we can obtain the similar conclusion like Mistral in main text. LAVa outperforms all baselines across different budgets, even in models with larger parameter size.

Results of LAVa in Needle In A Haystack. The results of Needle In A Haystack are shown in Table 8. The conclusion is consistent with that of

	Single-Doc. QA				Multi-Doc. QA				Summarization				Few-shot Learning				Synthetic			Code		
	NrtvQA	Qasper	MF-en	MF-zh	HotpotQA	2WikiMQA	Musique	Dureader	GovReport	QMSum	VCSUM	MultiNews	TREC	TriviaQA	SAMSum	LSHT	PCount	PR-en	PR-zh	Lcc	RepoBench-P	Avg
Full Cache	29.05	43.34	52.52	62.27	57.59	47.05	30.24	29.25	31.78	23.64	15.96	23.96	72.50	88.82	45.61	42.75	8.50	100.00	96.50	59.61	67.12	48.96
$\mathbb{B} = 128\text{HL}$																						
PyramidKV	21.96	26.41	42.53	52.77	49.33	42.17	23.48	17.88	16.80	19.29	11.24	14.30	42.50	83.78	41.15	22.39	8.50	95.50	63.50	48.53	51.39	37.88
SnapKV	**25.24**	27.66	43.90	53.53	51.00	42.12	24.59	18.56	18.04	19.85	11.32	15.55	41.00	83.18	40.68	24.88	**9.00**	**98.00**	81.50	49.44	52.58	39.60
Ada-PyramidKV	23.08	27.53	42.07	53.17	50.73	42.03	23.31	18.03	17.48	19.65	11.21	14.71	42.50	83.90	41.25	22.81	**9.00**	94.00	76.00	49.17	52.69	38.78
Ada-SnapKV	25.20	28.45	45.00	54.37	51.08	**44.02**	24.66	18.81	**18.26**	20.09	11.50	16.25	42.50	84.06	41.00	22.49	**9.00**	96.50	**87.50**	49.92	54.32	40.24
CAKE	24.43	**30.15**	45.03	54.86	50.65	42.41	**25.91**	18.89	18.21	**20.66**	11.60	15.84	42.00	84.54	41.95	**26.24**	8.50	95.50	81.50	51.60	55.09	40.26
LAVa (Ours)	23.29	28.87	**46.80**	**56.10**	**52.65**	42.96	25.09	**19.25**	18.24	20.52	**11.80**	**16.28**	**43.00**	**84.56**	**42.18**	23.95	8.50	96.00	85.00	**53.45**	**56.07**	**40.69**
$\mathbb{B} = 256\text{HL}$																						
PyramidKV	24.82	31.13	46.92	56.06	53.07	42.31	25.06	19.54	19.27	20.47	12.01	16.55	50.00	84.88	42.04	25.39	**8.50**	96.00	85.50	52.03	55.82	41.30
SnapKV	26.61	23.77	49.15	58.37	**56.03**	44.18	25.68	**20.96**	20.84	20.99	12.19	18.52	48.50	86.31	43.06	29.89	**8.50**	97.50	**95.00**	54.26	59.42	43.32
Ada-PyramidKV	25.97	31.01	47.31	56.43	54.17	43.03	25.23	19.41	19.60	21.09	11.87	17.07	**54.50**	86.04	42.69	27.28	**8.50**	97.00	90.00	52.78	56.55	42.26
Ada-SnapKV	26.52	34.50	**50.01**	58.28	55.61	43.60	26.14	20.89	**21.30**	20.94	12.51	18.59	52.50	85.50	42.97	28.43	**8.50**	**98.00**	93.50	53.94	59.30	43.41
CAKE	26.59	33.95	49.80	58.25	54.89	**44.42**	26.47	20.35	21.23	**21.94**	12.35	18.53	47.50	85.41	**43.51**	**32.33**	**8.50**	97.50	94.00	55.56	61.13	43.53
LAVa (Ours)	**27.04**	**35.19**	49.36	**59.74**	55.35	44.13	**27.25**	20.88	21.15	21.51	**12.77**	**18.96**	49.00	**86.73**	43.42	30.35	**8.50**	**98.00**	93.00	**56.19**	**62.19**	**43.84**
$\mathbb{B} = 512\text{HL}$																						
PyramidKV	28.02	35.74	**50.84**	58.11	55.26	44.72	25.85	20.94	21.83	21.34	12.33	18.95	59.50	86.13	43.04	32.83	**8.50**	99.00	**96.00**	55.65	59.42	44.48
SnapKV	**28.27**	28.22	50.69	60.27	**56.18**	44.69	27.28	21.98	23.79	21.89	**13.20**	20.64	59.50	84.10	43.68	35.52	**8.50**	**100.00**	94.00	56.66	62.69	45.32
Ada-PyramidKV	27.31	37.36	49.62	58.57	55.40	44.66	26.74	21.35	22.39	21.12	12.42	19.32	**62.00**	**86.29**	43.78	33.33	**8.50**	99.00	95.50	55.78	60.99	44.83
Ada-SnapKV	28.03	38.51	50.06	**60.54**	55.50	45.06	**28.81**	22.04	**23.98**	22.49	13.05	20.80	**62.00**	85.83	44.37	37.10	**8.50**	**100.00**	94.00	56.44	62.71	45.71
CAKE	28.17	**39.09**	50.22	60.00	54.89	45.21	26.31	22.20	23.65	21.98	13.04	20.57	57.50	85.60	44.61	37.23	**8.50**	99.50	94.00	**58.27**	63.95	45.45
LAVa (Ours)	27.21	39.08	50.47	60.09	55.63	**45.25**	27.75	**22.91**	23.83	**22.81**	13.05	**20.84**	58.50	86.15	**45.02**	**37.43**	**8.50**	**100.00**	93.50	58.02	**64.57**	**45.74**
$\mathbb{B} = 1024\text{HL}$																						
PyramidKV	28.06	40.11	51.83	60.22	**57.55**	45.38	29.31	22.42	24.35	22.04	13.12	21.12	68.00	85.27	44.18	36.99	**8.50**	**100.00**	**96.50**	58.29	62.56	46.47
SnapKV	29.01	42.02	51.86	**61.22**	56.82	45.04	28.95	23.97	26.26	22.76	13.66	**22.50**	68.50	86.85	**45.52**	**42.50**	**8.50**	**100.00**	**96.50**	57.94	65.59	47.43
Ada-PyramidKV	28.52	40.50	**51.87**	60.27	56.42	**45.80**	29.18	23.01	24.45	22.10	13.31	21.25	69.00	86.41	45.10	37.79	**8.50**	**100.00**	**96.50**	57.16	63.31	46.69
Ada-SnapKV	29.61	**42.30**	51.79	60.29	56.38	45.75	29.30	23.64	26.21	22.80	**13.85**	22.39	69.00	**88.09**	45.36	41.75	**8.50**	**100.00**	96.00	58.15	65.77	47.47
CAKE	29.70	41.08	51.85	60.64	57.34	45.02	**30.48**	23.82	25.92	**22.95**	13.69	22.45	67.50	86.63	45.22	42.00	**8.50**	**100.00**	**96.50**	59.49	65.99	47.47
LAVa (Ours)	**29.79**	41.68	51.84	60.79	57.04	45.27	30.01	**23.99**	**26.36**	22.90	13.81	22.42	**69.50**	87.42	45.46	41.00	**8.50**	**100.00**	**96.50**	**59.97**	**66.24**	**47.64**

Table 6: Final comparison based on Qwen2.5-7B-Instruct among 21 datasets of LongBench. (Note: The best result is highlighted in **bold**, and the second is in underline.)

	Single-Doc. QA				Multi-Doc. QA				Summarization				Few-shot Learning				Synthetic			Code		
	NrtvQA	Qasper	MF-en	MF-zh	HotpotQA	2WikiMQA	Musique	Dureader	GovReport	QMSum	VCSUM	MultiNews	TREC	TriviaQA	SAMSum	LSHT	PCount	PR-en	PR-zh	Lcc	RepoBench-P	Avg
Qwen2.5-14B-Instruct																						
Full Cache	29.33	45.19	53.59	62.79	62.59	57.69	38.47	29.87	29.74	23.53	14.75	21.90	77.50	90.23	47.27	50.00	9.23	98.67	98.25	62.60	51.13	50.21
Qwen2.5-14B-Instruct, B=128h																						
PyramidKV	19.67	22.26	39.57	50.04	50.75	49.47	30.31	16.67	16.10	19.43	10.53	13.51	42.00	82.29	40.90	27.00	12.12	82.50	56.67	54.52	41.38	37.03
SnapKV	21.04	25.50	42.11	49.89	54.31	51.87	33.60	17.78	17.12	19.95	10.75	14.53	43.50	85.95	41.81	26.75	10.50	89.58	65.00	55.42	43.42	39.07
Ada-PyramidKV	20.85	24.83	40.88	51.78	54.65	52.34	29.78	16.83	16.67	19.59	10.32	13.90	**46.50**	80.76	40.58	25.75	11.18	87.75	63.75	53.72	43.49	37.90
Ada-SnapKV	22.16	25.58	42.80	52.22	55.10	53.21	33.50	17.98	17.69	20.25	10.86	14.81	45.50	85.62	42.49	27.00	9.05	91.33	68.17	**56.26**	43.39	39.76
CAKE	22.20	26.13	42.10	50.83	54.75	53.25	31.77	17.73	17.56	19.98	10.84	**15.44**	44.00	**87.51**	**42.65**	**28.50**	**13.96**	86.50	**78.83**	54.92	43.90	40.16
LAVa (Ours)	**22.24**	**26.52**	**43.09**	**52.39**	**55.97**	**53.43**	**33.68**	**18.23**	**17.94**	**20.57**	**10.98**	15.10	46.00	86.79	42.20	27.17	10.53	**92.00**	73.00	55.74	**44.63**	**40.39**
Qwen2.5-14B-Instruct, B=512h																						
PyramidKV	26.18	38.19	48.71	59.81	**60.74**	55.26	36.82	20.55	21.21	21.27	11.86	18.43	68.50	89.21	45.38	44.25	8.59	**98.33**	96.75	59.71	48.71	46.59
SnapKV	**26.99**	39.34	48.84	59.34	60.20	54.86	37.47	21.43	22.25	21.95	11.93	19.34	66.50	88.78	45.95	45.25	8.22	98.25	**98.58**	61.12	49.42	46.95
Ada-PyramidKV	26.78	40.25	**49.71**	60.40	60.64	**55.69**	37.72	20.75	21.49	21.54	11.67	18.60	**70.00**	88.59	45.70	44.50	**8.77**	**98.33**	96.75	60.23	48.85	47.00
Ada-SnapKV	26.03	**41.56**	49.42	**60.88**	59.99	55.63	38.34	21.33	22.49	22.09	11.96	19.32	69.50	89.01	46.35	**46.75**	7.72	98.17	98.50	**62.21**	49.92	**47.48**
CAKE	25.39	39.92	48.62	60.30	60.42	55.19	**38.37**	21.40	22.56	21.72	**12.31**	**19.57**	**70.00**	**89.03**	46.19	46.25	6.68	98.17	98.25	60.90	49.31	47.17
LAVa (Ours)	26.23	40.65	48.93	59.45	60.34	55.36	37.50	**21.53**	**22.57**	**22.13**	11.91	19.48	67.00	88.68	**46.50**	**46.75**	7.98	97.75	97.75	61.85	**50.38**	47.18
Qwen2.5-32B-Instruct																						
Full Cache	**OOM**																					
Qwen2.5-32B-Instruct, B=128h																						
PyramidKV	21.32	27.86	43.55	56.05	55.74	53.85	32.25	16.74	17.08	18.88	10.71	15.76	48.00	54.41	**40.69**	29.50	11.17	94.00	73.09	48.04	35.36	38.29
SnapKV	21.72	28.31	42.83	56.03	54.43	55.52	30.78	16.94	16.92	19.04	10.53	15.69	48.50	58.30	39.64	27.50	12.00	93.75	74.37	47.15	35.82	38.37
Ada-PyramidKV	21.19	29.67	**45.61**	**58.04**	**57.30**	55.65	32.96	17.45	17.37	19.30	10.89	16.02	**51.50**	56.24	40.24	30.25	12.00	97.00	82.67	**48.14**	35.94	39.78
Ada-SnapKV	21.79	28.64	45.49	56.56	57.12	56.14	32.54	**17.66**	17.63	19.31	10.66	16.12	49.50	**60.07**	40.03	27.50	12.00	96.04	**85.13**	47.96	36.29	39.72
CAKE	21.28	28.40	43.30	55.71	55.93	54.89	32.86	17.04	17.00	19.44	10.50	16.18	46.50	56.35	40.38	**31.88**	**12.50**	94.79	82.92	46.63	36.05	39.07
LAVa (Ours)	**22.29**	**30.12**	45.50	57.06	56.59	**58.51**	**33.72**	17.50	17.42	**19.97**	**11.09**	**16.29**	48.50	57.21	40.23	28.17	10.00	**97.42**	84.09	48.12	**36.68**	**39.83**
Qwen2.5-32B-Instruct, B=512h																						
PyramidKV	26.00	37.40	48.67	61.17	60.60	60.44	34.75	19.37	20.84	20.61	11.64	18.48	66.00	55.11	42.71	39.00	**11.56**	99.75	98.54	50.28	38.12	43.86
SnapKV	25.71	40.23	48.81	62.94	61.16	60.60	34.85	20.64	22.69	21.27	11.61	20.04	66.50	77.77	44.01	41.86	11.19	**100.00**	99.03	52.20	**39.15**	45.82
Ada-PyramidKV	26.41	38.97	50.14	61.50	61.50	**61.86**	**37.55**	19.67	21.49	20.71	11.23	18.68	67.50	60.81	43.40	39.75	11.08	99.75	**99.62**	50.60	38.27	44.79
Ada-SnapKV	**27.51**	39.44	49.21	63.09	61.70	61.60	37.23	20.35	22.69	21.72	11.74	**20.45**	**69.00**	**77.87**	**44.19**	42.04	**11.56**	**100.00**	98.24	52.22	39.14	46.24
CAKE	25.32	40.24	49.66	**63.28**	59.75	61.42	37.11	20.44	22.73	21.22	11.67	20.28	66.50	77.31	43.92	**44.58**	11.19	**100.00**	98.78	**52.36**	38.99	46.04
LAVa (Ours)	26.56	**41.18**	**50.80**	62.49	**61.90**	60.83	37.25	**21.44**	**23.16**	**22.02**	**11.86**	20.30	68.50	77.69	43.97	42.23	11.50	**100.00**	98.53	52.24	38.86	**46.35**

Table 7: Final comparison based on Qwen2.5-14B-Instruct and Qwen2.5-32B-Instruct among 21 datasets of LongBench. (Note: The best result is highlighted in **bold**, and the second is in underline.)

Methods	Mistral-7B	Qwen2.5-7B
Full Cache	99.88	99.66
	$\mathbb{B} = 128HL$	
PyramidKV	91.44	91.10
SnapKV	91.25	93.28
Ada-PyramidKV	92.08	92.70
Ada-SnapKV	92.12	94.30
CAKE	92.79	94.61
LAVa (Ours)	**93.35**	**95.57**
	$\mathbb{B} = 1024HL$	
PyramidKV	97.88	99.56
SnapKV	97.95	99.48
Ada-PyramidKV	98.58	99.58
Ada-SnapKV	98.54	99.53
CAKE	98.32	99.55
LAVa (Ours)	**98.95**	**99.59**

Table 8: Average scores of Mistral-7B-Instruct-v0.2 and Qwen2.5-7B-Instruct in Needle In A HayStack.

LongBench. Our method shows superior overall performance, demonstrating its robust in preserving the model's retrieval capacity.

Results of LAVa in Ruler and InfiniteBench. The results of Ruler and InfiniteBench are shown in Table 10 and Table 11. we set the cache budget as 5%-10% of the task context length, i.e. 1024 and 10000. We use Mistral-7B-Instruct-v0.2 as the backbone of Ruler. For InfiniteBench, we change the backbone into Mistral-7B-LongPO-128K (Chen et al., 2025), which is fine-tuned based on Mistral-7B-Instruct-v0.2, because the task context length of InfiniteBench is much longer than the original maximum model length 32K. The results reconfirm the effectiveness of LAVa.

Results of Dynamic Budget Allocation. The detailed results of ablation study based on Mistral-7B-Instruct-v0.2 in LongBench are listed in Table 9. It demonstrates that dynamic budget allocation at both the head and layer levels is essential for strong performance, with a more pronounced performance drop when head-wise allocation is removed under constrained budgets. This is expected, as LAVa's strength lies in its ability to compare cache entries across heads.

Analysis of Different Layer Allocation. To validate the effectiveness of our layer budget allocation, we modify LAVa to incorporate two alternative strategies: **LAVa-Uniform**, which is equivalent to LAVa (-layer), and **LAVa-Pyramid**, which retains LAVa's head budget allocation and layer-wise cache eviction but adopts Pyramid for layer allocation. The results in Table 12 indicate that our method outperforms these alternatives. Notably, LAVa-Pyramid requires finetuning, whereas the other methods do not. Moreover, LAVa-Pyramid fails to outperform LAVa-Uniform at higher budgets, aligning with the observed comparison between Ada-SnapKV and Ada-Pyramid. This underscores the limitation of heuristic-based designs, which may not always yield optimal results.

Analysis of Time Complexity. Our study builds upon the SnapKV framework with a batch size of 1, consistent with prior works like CAKE and AdaKV. We start with the analysis for SnapKV (the most computationally efficient method among baselines) in computation for one layer as a reference.

- For layer l, SnapKV needs to calculate the layer's original KV Cache with the time complexity of $O(HN^2d_h)$, ignoring the IO operations. Generally, this is done with FlashAttention, which avoids saving the large attention matrix of size $O(N^2)$. The computation cost in practice is high due to IO operations and recomputation (to avoid saving the attention matrix), but we ignore it for simplicity.

- As Flash attention does not save the attention matrix, for calculating the scores to evict KV Cache, SnapKV needs to recompute the attention scores for the recent window of size w in the second pass. The time complexity is $O(HNwd_h)$.

- The top-$B_{l,h}$ selection for head-wise cache eviction with a min-heap takes $O(NlogB_{l,h})$, and for H heads, it takes $O(HNlogB_{l,h})$, where $B_{l,h}H = B_l, B_lL = \mathbb{B}$.

To summarize, SnapKV requires:

- $O(HN^2d_h)$ for original cache for one layer;

- $O(HNwd_h)$ for recomputing the recent attention scores;

- $O(HNlogB_{l,h})$ for cache eviction.

In contrast, LAVa requires the computation for one layer as follows:

	Single-Doc. QA				Multi-Doc. QA				Summarization				Few-shot Learning				Synthetic			Code		
	NrtvQA	Qasper	MF-en	MF-zh	HotpotQA	2WikiMQA	Musique	Dureader	GovReport	QMSum	VCSUM	MultiNews	TREC	TriviaQA	SAMSum	LSHT	PCount	PR-en	PR-zh	Lcc	RepoBench-P	Avg
Full Cache	26.77	32.34	49.63	48.42	43.43	27.89	18.61	30.85	32.92	24.54	15.04	27.20	71.00	86.23	43.41	39.00	2.81	86.56	89.75	55.29	52.55	45.07
$\mathbb{B}$ = 128HL																						
LAVa (Ours)	19.57	21.11	44.29	33.91	**38.29**	23.59	**15.32**	**18.56**	**19.33**	**22.32**	**11.42**	**21.07**	**53.50**	85.20	**40.16**	**21.75**	2.88	**69.87**	**74.75**	**51.94**	**48.92**	**36.74**
− layer	20.32	**21.18**	**45.17**	**35.00**	37.37	**23.62**	15.09	18.20	19.21	22.04	11.35	20.99	48.50	**85.32**	39.33	20.75	3.42	67.93	73.75	51.28	47.52	36.20
− head	**20.33**	20.27	44.06	32.23	36.64	22.84	14.19	18.15	18.88	21.51	11.09	20.89	45.00	84.29	39.57	20.25	3.21	65.23	64.25	51.88	47.51	34.95
$\mathbb{B}$ = 256HL																						
LAVa (Ours)	**22.70**	24.67	**48.62**	**37.81**	**39.68**	**25.96**	**16.77**	**20.26**	**21.92**	22.48	**11.88**	**22.91**	**65.00**	85.24	41.28	**26.75**	2.88	76.76	85.75	**54.17**	**51.77**	**40.12**
− layer	21.78	**24.74**	47.82	37.47	39.06	25.53	16.21	19.94	21.86	**23.22**	11.81	**22.91**	62.00	**85.37**	**41.53**	25.25	2.77	**78.53**	**87.67**	52.78	49.85	39.77
− head	21.34	22.77	47.43	35.87	37.71	25.50	15.47	19.43	21.55	23.06	12.08	22.86	58.00	84.88	41.69	22.25	3.11	74.77	84.18	53.89	51.19	38.80
$\mathbb{B}$ = 512HL																						
LAVa (Ours)	**25.01**	27.84	**48.97**	**42.14**	**40.95**	**26.88**	18.33	**21.12**	23.59	**23.59**	12.28	**24.51**	**68.50**	**86.34**	**42.48**	**33.50**	2.90	87.23	**89.83**	**55.83**	**52.85**	**42.59**
− layer	24.43	**27.98**	48.72	41.00	40.23	26.17	**18.50**	20.74	**24.00**	23.40	**12.68**	24.20	66.50	86.04	42.26	32.75	2.84	**87.89**	89.33	54.11	51.22	42.11
− head	23.59	27.70	48.61	40.61	40.22	25.79	17.87	20.68	23.91	23.39	12.38	24.28	66.50	86.09	41.95	28.50	2.97	86.88	89.17	55.73	52.53	41.82
$\mathbb{B}$ = 1024HL																						
LAVa (Ours)	25.59	**31.21**	48.27	43.43	**41.92**	27.38	**19.48**	**23.48**	26.06	**23.86**	**13.38**	**26.00**	**70.00**	86.22	42.43	**38.00**	2.73	87.01	88.75	**57.31**	**53.28**	**43.65**
− layer	**25.76**	30.38	**49.54**	**43.54**	41.08	27.03	18.83	22.73	25.79	23.69	13.13	25.88	69.50	**86.30**	**43.10**	37.25	2.71	87.56	**89.25**	55.04	51.67	43.35
− head	25.76	29.61	49.31	42.77	40.82	**27.63**	18.59	22.64	**26.29**	23.77	12.70	25.82	68.00	85.82	41.77	35.00	2.63	**89.06**	**89.25**	**57.31**	53.22	43.26

Table 9: Ablation study based on Mistral-7B-Instruct-v0.2 among 21 datasets of LongBench. (Note: The best result is highlighted in **bold**.)

Context Length	4K	8K	16K
PyramidKV	72.55	62.02	55.42
SnapKV	70.71	61.52	55.61
Ada-PyramidKV	70.80	60.83	54.95
Ada-SnapKV	71.14	60.31	55.05
CAKE	72.41	61.55	55.84
LAVa (Ours)	**75.39**	**62.61**	**56.70**

Table 10: Results of Mistral-7B-Instruct-v0.2 in Ruler.

Tasks	En Sum	En MC	En Dia
PyramidKV	25.3	67.2	6.5
SnapKV	25.1	67.2	7.0
Ada-PyramidKV	24.9	67.2	7.0
Ada-SnapKV	24.6	66.8	7.0
CAKE	24.8	**67.8**	6.6
LAVa (Ours)	**25.4**	66.8	**9.5**

Table 11: Results of Mistral-7B-LongPO-128K in InfiniteBench.

- $O(HN^2d_h)$ for the original cache of one layer, same as SnapKV;
- $O(HNwd_h)$ for recomputing the recent attention scores, same as SnapKV;
- $O(HNd_h)$ for computing the value norms for each token;
- $O(HNlogB_l)$ for layer-wise cache eviction because the eviction of LAVa is operated in all cache of one layer.

For one layer l, the difference of time complexity between LAVa and SnapKV is $O(HN(d_h+logH)$. In a long context, N is very large, and thus $O(HN(d_h+logH)$ is much smaller than the dominant factor $O(HN^2d_h)$. Based on the setting of Mistral-7B-Instruct-v0.2, we have $d_h = 128$ and $H = 32$, **the extra computation of LAVa compared to SnapKV is** $HN(d_h + logH)$ **divided by** HN^2d_h**, which is approximately 0.01% when** $N = 10,000$. The computation time increases with the increase of the number of layers and batch size for both SnapKV and LAVa, but the ratio of the extra computation time for LAVa is still 0.01%. A similar analysis can be achieved to see that all the other methods have similar latency, aligning with the latency results in Figure 3.

Analysis of Memory Usage. We analyze the difference between SnapKV and LAVa/CAKE, which are dynamic layer budget methods.

- For SnapKV, the cache size increases from $O(HKd_h)$ in the first layer to the last layer, where it reaches the peak of $O(LHB_{l,h}d_h)$. The memory peaks when the latest (full) layer cache $O(HNd_h)$ is not pruned, and the current retained cache reaches the size of $O(LHB_{l,h}d_h)$. In sum, the peak memory is $O(HNd_h + LHB_{l,h}d_h)$.
- For LAVa and CAKE, the cache size is always $O(LHB_{l,h}d_h)$ from the first layer to the last layer, yet it is distributed among prefilled layers. The memory peak, however, is similar to SnapKV, which is $O(HNd_h + LHB_{l,h}d_h)$, except that for LAVa/CAKE, we need to store the layer scores. As we save only the top scores for each layer, the size for scores is $O(LHB_{l,h})$. Given that the total cache size

	Single-Doc. QA				Multi-Doc. QA				Summarization				Few-shot Learning				Synthetic			Code		
	NrtvQA	Qasper	MF-en	MF-zh	HotpotQA	2WikiMQA	Musique	Dureader	GovReport	QMSum	VCSUM	MultiNews	TREC	TriviaQA	SAMSum	LSHT	PCount	PR-en	PR-zh	Lcc	RepoBench-P	Avg
Full Cache	26.77	32.34	49.63	48.42	43.43	27.89	18.61	30.85	32.92	24.54	15.04	27.20	71.00	86.23	43.41	39.00	2.81	86.56	89.75	55.29	52.55	45.07
										$\mathbb{B} = 128HL$												
LAVa-Pyramid	19.91	20.36	44.32	**35.06**	37.68	23.58	**15.40**	17.99	**19.61**	22.09	10.87	21.05	52.00	84.45	40.09	20.25	2.89	**72.32**	**76.92**	51.81	46.81	36.63
LAVa-Uniform	**20.32**	**21.18**	**45.17**	35.00	37.37	**23.62**	15.09	18.20	19.21	22.04	11.35	20.99	48.50	**85.32**	39.33	20.75	3.42	67.93	73.75	51.28	47.52	36.20
LAVa (Ours)	19.57	21.11	44.29	33.91	**38.29**	23.59	15.32	**18.56**	19.33	**22.32**	**11.42**	**21.07**	**53.50**	85.20	**40.16**	**21.75**	2.88	69.87	74.75	**51.94**	**48.92**	**36.74**
										$\mathbb{B} = 256HL$												
LAVa-Pyramid	21.22	23.96	47.86	37.12	38.92	24.94	16.70	19.11	21.43	22.44	11.20	22.77	62.50	85.17	41.34	23.75	3.34	**79.07**	86.58	52.25	49.70	39.40
LAVa-Uniform	21.78	**24.74**	47.82	37.47	39.06	25.53	16.21	19.94	21.86	**23.22**	11.81	**22.91**	62.00	85.37	**41.53**	25.25	2.77	78.53	**87.67**	52.78	49.85	39.77
LAVa (Ours)	**22.70**	24.67	**48.62**	**37.81**	**39.68**	**25.96**	**16.77**	**20.26**	**21.92**	22.48	**11.88**	**22.91**	**65.00**	85.24	41.28	**26.75**	2.88	76.76	85.75	**54.17**	**51.77**	**40.12**
										$\mathbb{B} = 512HL$												
LAVa-Pyramid	24.59	27.33	48.36	40.24	39.75	26.18	18.26	20.82	23.39	23.38	12.35	24.08	67.00	**86.66**	**42.55**	32.00	2.93	86.13	89.62	53.46	51.53	41.88
LAVa-Uniform	24.43	**27.98**	48.72	41.00	40.23	26.17	**18.50**	20.74	**24.00**	23.40	**12.68**	24.20	66.50	86.04	42.26	32.75	2.84	**87.89**	89.33	54.11	51.22	42.11
LAVa (Ours)	**25.01**	27.84	**48.97**	**42.14**	**40.95**	**26.88**	18.33	**21.12**	23.59	**23.59**	12.28	**24.51**	**68.50**	86.34	42.48	**33.50**	2.90	87.23	**89.83**	**55.83**	**52.85**	**42.59**
										$\mathbb{B} = 1024HL$												
LAVa-Pyramid	24.88	29.51	49.01	42.57	41.16	27.20	19.40	22.61	25.58	**24.00**	13.08	25.71	68.50	86.19	**43.19**	37.00	2.67	**87.73**	**90.25**	54.72	51.53	43.19
LAVa-Uniform	**25.76**	30.38	**49.54**	**43.54**	41.08	27.03	18.83	22.73	25.79	23.69	13.13	25.88	69.50	**86.30**	43.10	37.25	2.71	87.56	89.25	55.04	51.67	43.35
LAVa (Ours)	25.59	**31.21**	48.27	43.43	**41.92**	**27.38**	**19.48**	**23.48**	**26.06**	23.86	**13.38**	**26.00**	**70.00**	86.22	42.43	**38.00**	2.73	87.01	88.75	**57.31**	**53.28**	**43.65**

Table 12: Layer allocation comparison based on Mistral-7B-Instruct-v0.2 among 21 datasets of LongBench. (Note: The best result is highlighted in **bold**.)

Tasks	Qasper	HotpotQA	Gov Report	TriviaQA	Passage Retrieval ZH	LCC
			Layer 0			
AdaKV	1.77	1.63	1.82	2.64	1.59	1.91
LAVa	**1.61**	**1.59**	**1.73**	**2.61**	**1.40**	**1.86**
			Layer 31			
AdaKV	134.69	133.33	107.94	121.53	93.50	149.25
LAVa	**132.97**	**130.02**	**106.06**	**121.31**	**90.50**	**147.16**

Table 13: Results of Layer Attention Output Loss.

is $O(LHB_{l,h}d_h)$, it is sufficient to just keep a total of LHK scores for comparison. Again, the extra factor is dominated by $O(HNd_h + LHB_{l,h}d_h)$. **The extra memory usage of LAVa is 0.6% of SnapKV peak memory** when $L = H = 32, B_{l,h} = 1024, d_h = 128$, and $N = 10,000$. This is small, but not as negligible as in time complexity, consistent with Figure 3. However, dynamic layer budget is important for tasks like summarization or code generation, as shown in Figure 2.

Analysis of Layer Attention Output Loss. To validate the effectiveness of LAVa in minimizing layer attention output loss, we compare LAVa with AdaKV, which also aims to minimize layer attention output loss and its scoring function is the same with SnapKV. We set the cache budget as 128 to make the difference clear and calculate the loss in the first and the last layer. The backbone is Mistral-7B-Instruct-v0.2. The results in Table 13 are consistent with the evaluation of other benchmarks, proving that the upper bound of LAVa is tighter compared to that of AdaKV.