# AUV: Efficient KV Cache Eviction for LLMs via Attention Score Aggregation and Usage Count

Quanli Li*[1,2,3][0009−0004−7397−6802], Qinglin Wang*†[1,2,3][0000−0002−8286−6566], Hongyang Hu[1,2,3][0000−0002−7564−587X], Xiaoyan Luo[1,2,3][0009−0008−2740−2217], and Jie Liu[1,2,3][0000−0003−3745−7541]

[1] College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China
[2] National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, Changsha 410073, China
[3] Laboratory of Digitizing Software for Frontier Equipment, National University of Defense Technology, Changsha 410073, China
{ll,wangqinglin,huhongyang,lxy_,liujie}@nudt.edu.cn

**Abstract.** As transformer-based large language models (LLMs) evolve, the need for efficient inference, particularly in managing the KV cache during the decoding phase, has intensified. However, the increasing sequence lengths in LLMs lead to substantial memory usage, which hinders their practical deployment. Existing cache eviction strategies, primarily based on accumulated attention scores, tend to prioritize early tokens, causing unstable performance and neglecting the integration of key metrics for dynamic cache management. To address these challenges, we propose AUV, an innovative cache eviction framework. AUV introduces a novel attention score aggregation method to mitigate the uneven eviction. It combines two complementary metrics: Total Attention Level and Strong Attention Frequency, by integrating the aggregated attention score with the usage count. Furthermore, AUV adopts a multi-step eviction strategy along with an eviction compensation mechanism to optimize both efficiency and accuracy. Finally, AUV improves KV cache management to avoid fragmentation. Through extensive experiments with OPT models, we demonstrate that AUV outperforms existing methods, including H2O, NACL, and full cache, under low cache budgets, achieving greater accuracy while maintaining throughput. Notably, AUV achieves a 10 percentage point improvement over H2O in the BERTScore-F1 metric when retaining only 2% KV cache budget. These results highlight the potential of AUV to reduce memory consumption without sacrificing performance.

**Keywords:** Large language models · Transformer · KV cache eviction · Attention score aggregation · Usage count.

*Equal contribution.
†Corresponding author.

## 1 Introduction

In the context of rapid advances in artificial intelligence technologies, large language models(LLMs) based on transformer architecture [15] have demonstrated exceptional performance and gained widespread application. Inference with large-scale autoregressive language models can be divided into two stages: prefill and decoding. During the prefill phase, the model processes the entire input sequence in parallel, computing Keys and Values for all input tokens, which allows for high GPU parallelism. In contrast, the decoding phase generates new outputs in an autoregressive manner. To avoid redundant calculations at each step, the common approach is to store all previously computed Keys and Values as KV cache.

However, these models typically contain billions or even trillions of parameters, and as the generated sequence length increases, the size of the KV cache grows linearly with the sequence length. This results in significant strain on GPU memory, becoming a major bottleneck in practical applications. Existing work to mitigate the pressure on the KV cache focuses primarily on improving memory layout to reduce internal and external fragmentation, and on evicting low-importance token states.

A representative work, the vLLM system [7], employs a PagedAttention mechanism, which introduces a paging mechanism similar to operating system paging. Divides the KV cache into fixed-size memory blocks, allowing noncontiguous storage, and thereby eliminating fragmentation. However, while paging alleviates memory fragmentation, it does not actively reduce the actual cache size. As the sequence length grows, the cache will eventually reach its limit. Eviction strategies based on attention values, such as H2O [24], have shown that only a small number of tokens (heavy hitters) contribute most of the attention value. Therefore, it suffices to retain only the Keys and Values for these important tokens, significantly reducing memory consumption.

Although these methods mitigate the KV cache pressure, they still have limitations. First, although studies like NACL [2] observed that the heavy hitters retained by H2O were mainly concentrated at the beginning of the sequence, the reason for this was not fully explored. NACL addresses this by selecting random numbers, but this method lacks interpretability and can result in the eviction of important tokens. Second, current KV cache eviction strategies typically focus only on aggregated attention scores, which is insufficient. We observe two complementary metrics for KV cache eviction: *Total Attention Level* and *Strong Attention Frequency*. *Total Attention Level* refers to the extent to which the current key is tended by subsequent queries. Eviction strategies such as H2O use this metric. *Strong Attention Frequency* tracks how often a key is attended to with high attention by subsequent queries. Methods like Scissorhands [11] employ a similar approach, evicting tokens that are rarely handled with care. To our knowledge, there is no eviction strategy that integrates both of these metrics.

To holistically address these limitations, we propose AUV, a unified eviction framework that incorporates multiple synergistic innovations. We first investi-

gate the root cause of the uneven eviction issue, which stems from the aggregation method used for attention scores. Most eviction methods, including H2O, rely on accumulated attention scores, with methods such as SnapKV [9] using clustering based on accumulated scores, Pyramid [1] employing instruction-based accumulated attention scores, and NACL using proxy token-based accumulated scores. We introduce a new attention score aggregation method that thoroughly resolves this issue. We also address the second limitation through a new approach, inspired by the classical page replacement algorithm, the Second Chance Algorithm. This method introduces a usage count mechanism, ensuring that tokens that have been strongly attended to recently are given a chance to be retained, even if their accumulated attention score is low. This approach combines both *Total Attention Level* and *Strong Attention Frequency* in the eviction strategy, achieving high hit rates. Additionally, to avoid the performance overhead caused by overly complex strategies, we implement a multi-step eviction strategy to further enhance efficiency. To improve the accuracy of the model, evicted key-value pairs are not entirely discarded. Instead, some information is retained using the reconstructed K(V) method. Moreover, during the process of storing new KV cache and performing KV cache eviction, improper memory management can result in excessive fragmentation, leading to inefficient memory utilization and significantly degrading system performance. To address this issue, we propose a memory management method based on scatter filling and positioning pointers, which effectively prevents memory fragmentation.

Experimental results demonstrate that AUV can maintain or even surpass the performance of full cache under extremely limited cache budgets, exhibiting excellent stability and adaptability. Our main contributions of this work can be summarized as follows:

- We introduce an innovative attention score aggregation mechanism in our AUV to address the issue of uneven eviction.
- We design a dual-metric eviction strategy in our AUV based on usage count, and further optimize efficiency, accuracy, and memory management.
- We validate the effectiveness of the proposed method across multiple models and datasets, demonstrating that it outperforms H2O and NACL in terms of accuracy, while achieving a significant throughput improvement compared to H2O.

## 2 Related works

### 2.1 Static Sparse Attention

LM-Infinite [4] and StreamingLLM [17] cache the initial token and the last L tokens' keys and values. MInference [6] and FlexPrefill [8] combine predefined structures with dynamic computation to balance stability and efficiency, but still fundamentally rely on static sparsification patterns. Although these static sparsification approaches are computationally efficient, they lack flexibility and do not perform optimally across varying input sequences.

### 2.2 Computation Efficient Sparse Attention

SpargeAttn [21] uses rapid attention map prediction and perceptual Softmax in a two-stage process to filter out unnecessary matrix multiplications. Twilight [10] leverages a hierarchical Top-p (nucleus sampling) strategy to adaptively prune attention. XAttention [18] incorporates block-level importance evaluation based on anti-diagonal sums. NSA [19] uses coarse-grained token compression and fine-grained token selection to implement its dynamic hierarchical sparsity strategy. These methods significantly reduce computational load during the prefill phase but do not alleviate memory consumption during the memory-bound decoding phase.

### 2.3 Adaptive KV Cache Eviction

The Attention-Gate method [20] introduces lightweight gating modules within LLMs. However, this method requires training or fine-tuning, making its application to existing models costly. A class of methods dynamically selects KV pairs based on heuristics or attention signals, eliminating the need for model training or fine-tuning. H2O [24] dynamically retains the latest tokens and "heavy hitter" tokens. SnapKV [9] clusters and identifies important features for each head based on observation window. Scissorhands [11], similar to H2O, retains "key tokens" with high probability during inference. SparQ Attention [13] eliminates unimportant KV cache using approximate attention scores. InfLLM [16] divides the KV cache into blocks and selects representative tokens for each block. NACL [2] combines attention-based "proxy token" eviction with random eviction. However, most of these methods rely on flawed attention score aggregation strategies for eviction and consider only a single evaluation metric.

## 3 AUV

### 3.1 KV Cache

As illustrated in Fig. 1, in the decoding stage of large language models, the token at timestep $t$ undergoes a series of transformations to become the query vector $Q_t$ of each head for the current timestep. This query vector is then dot-multiplied with the key vectors $K_{1:t}$ from all previous timesteps to obtain the attention scores. These scores are scaled and passed through a softmax function to normalize them. Subsequently, the value vectors $V_{1:t}$ from all previous timesteps are weighted according to the normalized scores and summed to produce the attention vector for the current timestep. This vector is then processed through additional transformations to generate the token for the next timestep, $\text{token}_{t+1}$. The Attention calculation formula is as follows:

$$Attention(Q_t, K_{1:t}, V_{1:t}) = softmax\left(\frac{Q_t K_{1:t}^T}{\sqrt{d_k}}\right) V_{1:t} \tag{1}$$

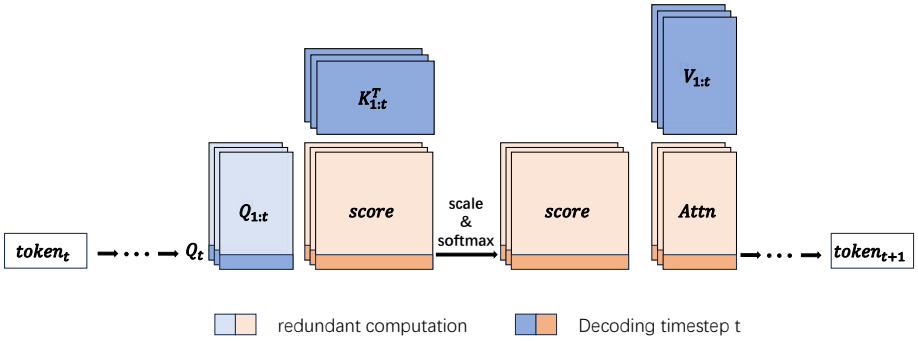



Fig. 1: In the KV Cache illustration, the light-colored regions represent data that are redundant in comparison to traditional attention computation and can be omitted when using the KV Cache.

By caching the previously computed key and value vectors, redundant computations during each decoding step are avoided, thereby significantly reducing both computational and memory access overhead.

### 3.2 Attention Score Aggregation in the Prefill Stage

In the prefill stage, existing methods for selecting important keys, represented by $H_2O$, typically perform aggregation by accumulating attention scores column-wise. However, this approach has two main drawbacks. First, due to the presence of masks, the number of valid elements increases with the row index, causing earlier keys to attend to more queries and thus accumulate higher aggregation scores. Second, under the softmax mechanism, where each row sums to one, a decreasing number of valid elements leads to score concentration in the upper-left submatrix, further amplifying the aggregation scores of earlier keys.

In Fig. 2, consider an example with four tokens, where the pairwise query-key inner product scores are identical. The resulting attention score matrix has each row consisting of equal valid elements summing to 1. $H_2O$ aggregation method accumulates the attention scores column-wise for each key. It is evident that the aggregation scores vary significantly between keys, and using these scores as eviction importance criteria would prioritize retaining earlier tokens. Although the score differences between keys are reduced in the average method, eviction would still prioritize earlier tokens. We propose a new aggregation method which first scales each row using the following formula:

$$x_{ij} = x_{ij} \cdot \frac{N_i}{N_{max}} \tag{2}$$

where $x_{ij}$ represents the value at row $i$, column $j$ of the attention score matrix, $N_i$ is the number of valid values in the current row (excluding masked values), and $N_{max}$ is the maximum number of valid values in any row.

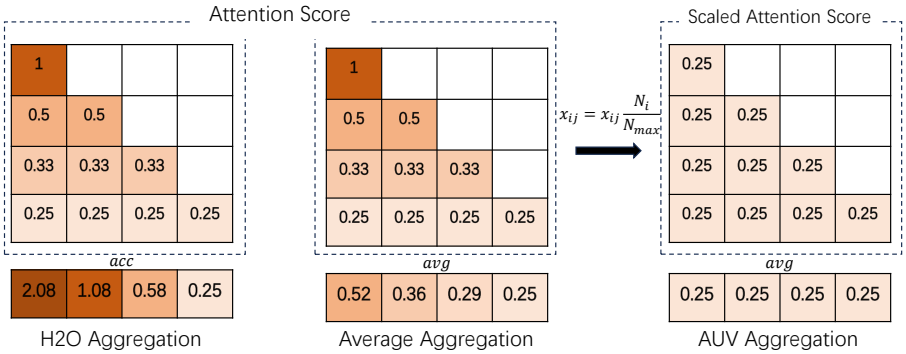



Fig. 2: Comparison of attention score aggregation methods: H2O, Average, and AUV.

This scaling function normalizes each row to the same scale while preserving the relative magnitudes of the elements. Averaging each column then yields more equitable aggregation scores.

It should be noted that directly averaging the attention score matrix without applying Softmax yields similar results. This is because, in this example, the pairwise scores are identical. In cases with varying scores, Softmax amplifies differences through exponentiation, leading to significantly different outcomes.

### 3.3 KV Cache Eviction in the Decoding Stage

Before performing KV cache eviction in the decoding stage, the usage count of the keys needs to be updated. After computing the new attention score vector by performing a query-key interaction across all timesteps, the top-k highest scores are selected, indicating that the corresponding keys were used at the current timestep. The usage count for these keys is incremented by 1. The aggregation of the attention scores involves combining the current attention score vector with the aggregated attention scores in the prefill stage according to the rules introduced in Section 3.2.

At the same time, the aggregated attention score needs to be updated using the following formula:

$$AAS' = \frac{\left(attnScore \cdot \frac{N_i}{N_{max}} + AAS \cdot scoreCount\right)}{scoreCount'} \tag{3}$$

where

$$scoreCount' = scoreCount + 1 \tag{4}$$

In this formula, $attnScore$ represents the attention score, $AAS$ refers to the aggregated attention score, and $scoreCount$ indicates the count of scores being

aggregated. The attention score is scaled using the method described in Section 3.2. Then, the sum of the past $AAS$ is added. Since the aggregation is performed by averaging, and the number of elements aggregated at each position is not the same, the score count is also a vector. After obtaining the sum, the average is computed by dividing by the total count, which is the sum of the previous score count and the count of the current step.

**Naive Version of Single-Step Eviction** The implementation of eviction based on usage count follows the steps of the second chance algorithm, as shown in the naive version in Fig. 3. First, the keys are sorted by $AAS$ in ascending order. Then, they are traversed from top to bottom. If the usage count of the current key is 0, the corresponding key-value pair will be evicted. Otherwise, the usage count of the current key is decremented by 1, and then the next key will be checked. Since the usage count for all keys in Fig. 3 is greater than 1, no eviction is found during the first traversal. The process then restarts, and since the usage count keeps decreasing, the evicted key-value pair will eventually be found. After traversing three times, the usage count of one key reaches 0, and it is evicted in the fourth round.

However, this approach requires one sorting step and multiple traversals, which results in a high time overhead. Thus, we consider ways to optimize this process.

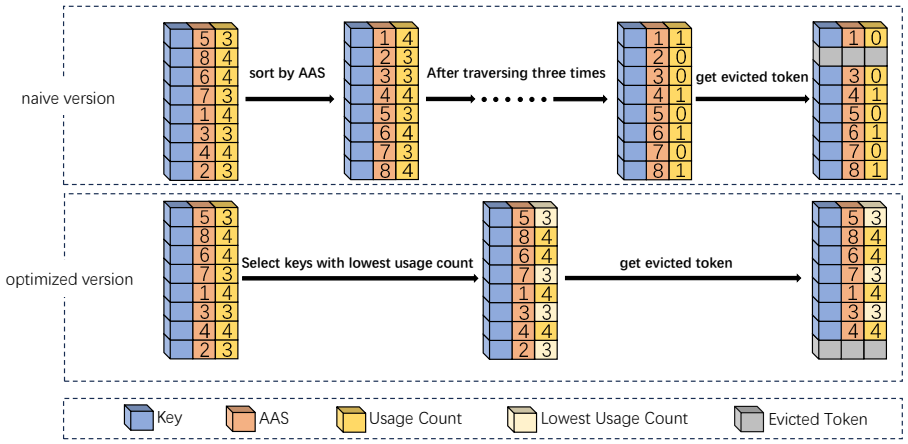



Fig. 3: Eviction flow of two single-step eviction versions.

**Optimized Version of Single-Step Eviction** Upon observation, it is evident that the key to be evicted always has the lowest usage count. Therefore, we propose an improvement to this process, as shown in the optimized version in

Fig. 3. First, all keys are traversed once to identify those with the lowest usage count. Note that there may be more than one such key. Then, among the selected keys, the one with the lowest value of $AAS$ is selected for eviction.

This method only requires two traversals to identify the evicted key-value pair. However, each traversal still incurs time overhead, so further optimization is considered.

**Multi-Step Eviction** We use a multi-step eviction approach to avoid the time overhead incurred by single-step evictions. This also allows past keys to interact more thoroughly with new queries and to aggregate more effectively, improving eviction accuracy. If we continue with the single-step eviction process, first checking usage count and then checking $AAS$, it becomes difficult to determine how many indices to select based solely on usage count. As observed in the previous section, the primary factor for evicting a key-value pair is the value of the usage count. In cases where the usage count is equal, $AAS$ value is used as the secondary criterion. To address this, we construct a composite vector for multi-step eviction, the composite vector is constructed as follows:

$$compositeVector = usageCount \cdot \max(AAS) + AAS \tag{5}$$

Here, the usage count of each key is multiplied by the maximum value in $AAS$, and then the values of $AAS$ are added to form the composite vector. To evict the top-k key-value pairs, we select the indices of the lowest values in the composite vector.

Additionally, to prevent the composite vector from becoming too large and causing overflow, the usage count is updated after each eviction. The formula is as follows:

$$usageCount' = (usageCount - \min(usageCount)) \cdot \alpha \tag{6}$$

In this step, the minimum value in the usage count vector is subtracted. This can reduce the computational burden for subsequent calculations without affecting the results. To further refine this, after each multi-step eviction, the past usage counts are multiplied by the decay factor $\alpha$ to counteract some of the errors introduced by the time-step difference.

This method also requires only two traversals but only performs the eviction at the final step of Multi-Step, significantly reducing the time overhead.

**Reconstructed K(V)** To improve accuracy, we aggregate the information of evicted key-value pairs into the reconstructed K(V).The formula for this is as follows:

$$rcsKV' = \frac{\left(rcsKV \cdot SES\right) + \sum_{i=1}^{n} KV \cdot AES_i}{SES'} \tag{7}$$

Where:

$$SES' = SES + \sum_{i=1}^{n} AES_i \tag{8}$$

In this formula, $n$ is the number of evicted pairs, $rcsKV$ is the reconstructed K(V), $AES_i$ is the aggregated attention score for each evicted key-value pair, and SES refers to the sum of all $AES$. Each evicted key (value) is fused into reconstructed K(V) by weighted averaging according to their aggregated attention scores.

### 3.4 KV Cache Memory Management

**Memory Management in Single-Step Eviction** In single-step eviction, the memory is typically managed using circular pointers and scatter filling. By maintaining a specific region, all KV cache entries can be efficiently stored. Fig. 4 illustrates the memory management during single-step eviction. The size of K(V) is $(sequence\ length, batch \times attention\ heads, hidden\ size)$ . Each head has one evicted element. The latter half of the sequence is treated as the local area. KV cache in this area will not be evicted, as these new keys have not accumulated enough usage count, and the number of elements aggregated by their corresponding $AAS$ values is insufficient. Therefore, eviction occurs only in the older KV cache entries within the candidate area.

Within the local area, there is a circular pointer that points to the oldest of the local tokens. The oldest element is scatter-inserted into the evicted area as shown in the figure below. Then, the newly generated element is inserted at the position of the circular pointer, and the pointer moves to the next position. This process achieves eviction replacement without occupying additional memory or creating fragmentation.

**Memory Management in Multi-Step Eviction** As discussed earlier, the time overhead of single-step eviction is relatively large. Therefore, a multi-step eviction version of memory management is introduced. As shown in Fig. 4, during multi-step eviction, the sequence length is expanded to three times the length of the local area to accommodate more new elements. Multi-step eviction starts when the number of incoming new elements equals the length of the local area, and the number of evicted elements also equals the local area length. The reason for this is to ensure that each key has been exposed to a number of queries equal to the length of the local area before eviction, achieving a dynamic equilibrium effect.

The pointer managing the positions of new elements is called the positioning pointer, which in the figure points to the end of the matrix, indicating that the current local K(V) cache is full. After obtaining the eviction index, the elements in the current local area are scattered to the evicted positions. The positioning pointer is then reset to receive the next new element.

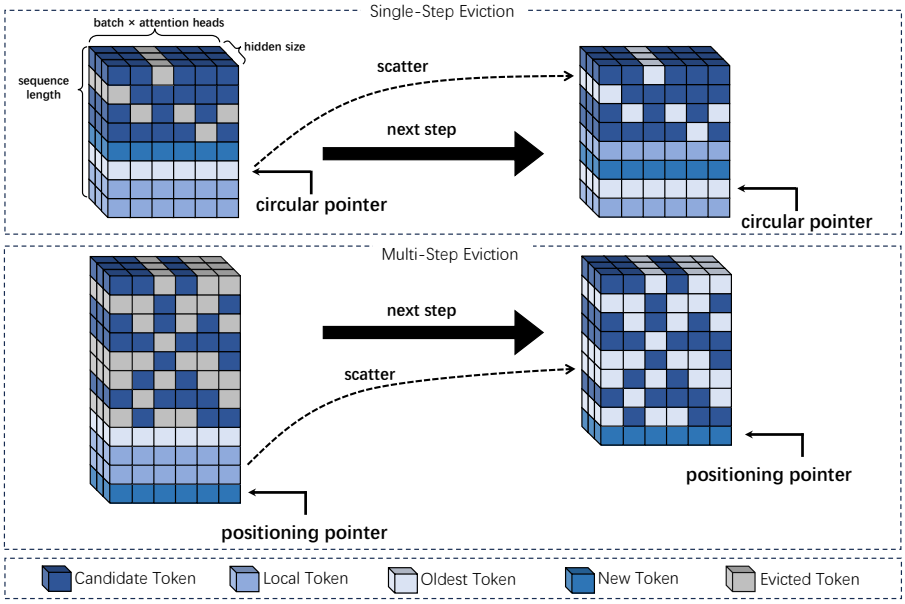



Fig. 4: KV cache memory management with single-step and multi-step (4-step) eviction.

## 4 Experiments

In this section, we evaluate the performance of our method (AUV) through a series of experiments, comparing it with existing approaches across different datasets and cache budget scenarios. Additionally, we conduct ablation studies to assess AUV from both accuracy and throughput perspectives, analyzing the contributions of the various components of the method.

### 4.1 Experimental Setup

For our experiments, we selected the OPT (Open Pre-trained Transformer) model [22], which has demonstrated strong language generation capabilities and has been pre-trained on large-scale corpora. We conducted experiments with two versions of the model: OPT-13B and OPT-30B. All experiments were run on a single GPU. We selected three datasets for evaluation: XSUM [12], CNN/DailyMail [14, 5], and Newsroom [3]. From each dataset, we randomly selected 1,000 samples.

We evaluated the performance using two categories of metrics. The first category is BERTScore [23], a metric based on the BERT model, which measures the semantic similarity between the generated text and reference text. The second category is ROUGE (Recall-Oriented Understudy for Gisting Evaluation), which measures the lexical overlap between the generated and reference texts. We

use three specific ROUGE metrics: ROUGE-1 (overlap of unigrams), ROUGE-2 (overlap of bigrams), and ROUGE-L (longest common subsequence). For both metrics, we report the F1 score.

### 4.2 Comparison with Existing Methods

In this section, we compare the performance of our proposed method with three existing KV cache eviction methods: H2O, NACL, and full cache (i.e., no eviction).

- **H2O**: The first method to propose retaining "heavy-hitter" tokens based on aggregated attention scores, significantly reducing memory usage while maintaining performance, but it may overlook important factors like token usage frequency.

- **NACL**: The latest advancement in KV cache eviction, combining attention scores with token usage history for more dynamic eviction decisions, but its performance may be unstable due to randomness in token selection.

Since all three methods performed well at higher cache budgets, we chose to focus on more challenging lower-budget scenarios. We reduced the KV cache budget to 2%, 4%, 6%, 8%, and 10% of the prefill size, representing a reduction of the cache by factors of 50x to 10x, and observed the performance trend as the budget was compressed.

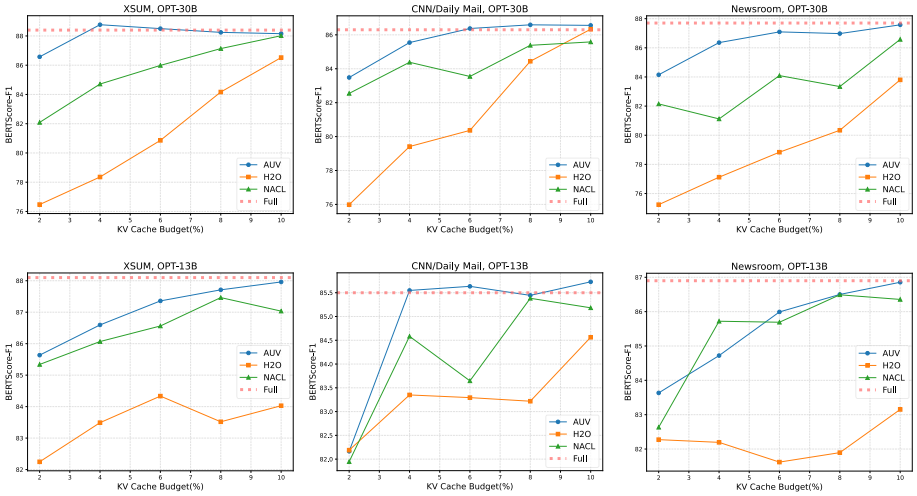



Fig. 5: Comparsion results between the baseline model with our AUV, H2O, NACL and Full cache.

The results in Fig. 5 indicate that, with only 2% of the budget, our method (AUV) outperforms H2O by 10 percentage points when using the OPT-30B

model. Moreover, in all low-budget scenarios, H2O shows poor performance, suggesting that it fails to retain the most important tokens. However, H2O is stable, and its performance gradually improves as the budget increases. The NACL method exhibits significant fluctuations, indicating that it may evict important tokens due to its random eviction strategy, leading to unstable performance. In contrast, our method either steadily increases in performance from a lower starting point or performs similarly to full cache, sometimes even exceeding full cache. This demonstrates our methods ability to filter out unimportant information effectively. Overall, AUV provides stable performance and consistently outperforms other methods.

Table 1: ROUGE scores on XSUM, CNN/DailyMail, and Newsroom datasets.

| Models | Methods | XSUM | | | CNN/DailyMail | | | Newsroom | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | ROUGE-1 | ROUGE-2 | ROUGE-L | ROUGE-1 | ROUGE-2 | ROUGE-L | ROUGE-1 | ROUGE-2 | ROUGE-L |
| OPT-30B | Full | **35.5** | **13.9** | 24.8 | **24.4** | **10.3** | **19.3** | **26.3** | 7.5 | 21.7 |
| | AUV | 31.3 | 10.3 | **28.5** | 23.2 | 5.5 | 18.3 | 24.0 | **8.2** | **22.3** |
| | H2O | 27.6 | 7.9 | 21.5 | 21.3 | 6.4 | 14.8 | 17.8 | 3.6 | 14.4 |
| | NACL | 28.2 | 8.4 | 21.9 | 19.6 | 5.0 | 14.9 | 22.0 | 5.7 | 19.4 |
| OPT-13B | Full | 34.3 | 12.3 | **27.9** | **24.7** | 7.8 | 18.0 | **22.7** | 7.5 | **19.3** |
| | AUV | **34.7** | **12.9** | 27.2 | 21.3 | **8.4** | 15.5 | 21.6 | **7.6** | 17.2 |
| | H2O | 21.4 | 6.1 | 17.2 | 12.8 | 2.5 | 10.9 | 15.1 | 5.6 | 13.4 |
| | NACL | 33.2 | 12.8 | 26.9 | 18.1 | 5.6 | 13.2 | 16.9 | 4.7 | 13.4 |

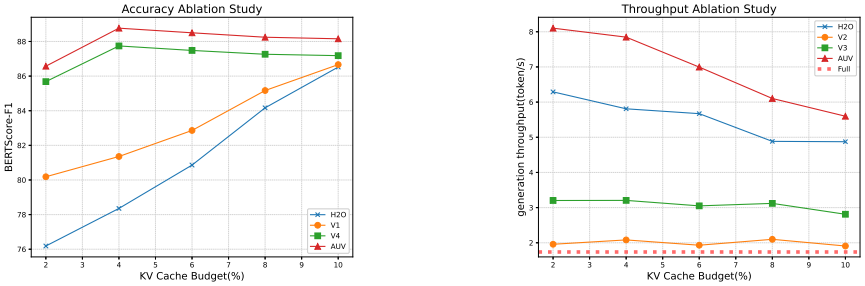
Table 1 shows the highest scores achieved by each model for each dataset at a 10% compression rate. The highest scores are highlighted in bold. It is evident that AUV and full cache dominate the top scores, with AUV performing similarly to full cache, outperforming H2O and NACL.

### 4.3 Ablation Study

**Accuracy Ablation Study** In this experiment, we evaluate the accuracy of different versions of the method using the XSUM dataset and BERTScore F1 scores.

As shown in Fig. 6 (a), modifying the attention score aggregation rule (V1) results in a performance trend similar to that of H2Oshowing steady improvement but without a substantial breakthrough. In contrast, the introduction of the multi-step eviction method based on usage count (V4) leads to a significant performance improvement, achieving a new level. Finally, the AUV, which incorporates the maintenance vector and decay factor, further enhances performance.

**Throughput Ablation Study** In this experiment, we evaluate the throughput of different versions of the method using the XSUM dataset with the OPT-13B model. The evaluation is conducted with a batch size of 1, after warming up




(a) Accuracy Ablation Study (b) Throughput Ablation Study

Fig. 6: Ablation study. In addition to H2O, Full cache, and AUV, we also compared several versions of AUV: only modifying attention score aggregation rule (V1), naive single-step eviction (V2), optimized single-step eviction (V3), multi-step eviction based on usage count (V4).

with 100 samples, and measuring the number of tokens generated per second over 1,000 samples.

As shown in Fig. 6 (b), the naive single-step eviction (V2) requires multiple rounds of traversal to identify eviction candidates, making it the slowest among the four methods. However, it still performs slightly better than the full cache, which incurs higher communication and computation costs due to caching all KV pairs. This highlights the importance of KV cache eviction. After optimizing the single-step eviction (V3), throughput shows a slight improvement, but the eviction computation cost remains relatively high, and the performance trend does not show significant improvements with varying budgets, rendering it inferior to H2O. With the multi-step eviction optimization, throughput improves substantially, reaching 5-8 times that of the full cache, and continues to increase as the cache budget is reduced.

## 5 Conclusion

In this paper, we present a novel AUV framework for efficiently performing KV cache eviction in large language models (LLMs). Our approach introduces a new attention score aggregation technique that avoids primarily retaining early tokens. It combines two complementary metrics based on a usage count mechanism, ensuring more efficient eviction with high hit rates. By integrating multi-step eviction strategy, compensation mechanism, and KV cache memory management, the method maintains high accuracy and throughput. Through extensive experiments on the OPT models, the robustness of AUV is validated, demonstrating consistent advantages in both accuracy and memory efficiency. These results confirm that AUV provides a significant improvement in both memory efficiency and model performance under constrained conditions.

Future work will explore extending AUV's applicability to other Transformer-based models and tasks. Additionally, we plan to further compress KV cache by incorporating quantization and low-rank mapping techniques, and provide more refined budget management for different attention heads and model layers. The ability of AUV to maintain high performance under strict memory constraints makes it a promising solution for optimizing LLMs in applications.

**Acknowledgments.** This study was funded by the National Key Research and Development Program of China(Nos. 2023YFA1011704 2021YFB0300101).

## References

1. Cai, Z., Zhang, Y., Gao, B., Liu, Y., Liu, T., Lu, K., Xiong, W., Dong, Y., Chang, B., Hu, J., et al.: Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. arXiv preprint arXiv:2406.02069 (2024)
2. Chen, Y., Wang, G., Shang, J., Cui, S., Zhang, Z., Liu, T., Wang, S., Sun, Y., Yu, D., Wu, H.: Nacl: A general and effective kv cache eviction framework for llms at inference time. arXiv preprint arXiv:2408.03675 (2024)
3. Grusky, M., Naaman, M., Artzi, Y.: Newsroom: A dataset of 1.3 million summaries with diverse extractive strategies. arXiv preprint arXiv:1804.11283 (2018)
4. Han, C., Wang, Q., Xiong, W., Chen, Y., Ji, H., Wang, S.: Lm-infinite: Simple on-the-fly length generalization for large language models (2023)
5. Hermann, K.M., Kocisky, T., Grefenstette, E., Espeholt, L., Kay, W., Suleyman, M., Blunsom, P.: Teaching machines to read and comprehend. Advances in neural information processing systems **28** (2015)
6. Jiang, H., Li, Y., Zhang, C., Wu, Q., Luo, X., Ahn, S., Han, Z., Abdi, A., Li, D., Lin, C.Y., et al.: Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. Advances in Neural Information Processing Systems **37**, 52481–52515 (2024)
7. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C.H., Gonzalez, J., Zhang, H., Stoica, I.: Efficient memory management for large language model serving with pagedattention. In: Proceedings of the 29th Symposium on Operating Systems Principles. pp. 611–626 (2023)
8. Lai, X., Lu, J., Luo, Y., Ma, Y., Zhou, X.: Flexprefill: A context-aware sparse attention mechanism for efficient long-sequence inference. arXiv preprint arXiv:2502.20766 (2025)
9. Li, Y., Huang, Y., Yang, B., Venkitesh, B., Locatelli, A., Ye, H., Cai, T., Lewis, P., Chen, D.: Snapkv: Llm knows what you are looking for before generation. Advances in Neural Information Processing Systems **37**, 22947–22970 (2024)
10. Lin, C., Tang, J., Yang, S., Wang, H., Tang, T., Tian, B., Stoica, I., Han, S., Gao, M.: Twilight: Adaptive attention sparsity with hierarchical top-$p$ pruning. arXiv preprint arXiv:2502.02770 (2025)
11. Liu, Z., Desai, A., Liao, F., Wang, W., Xie, V., Xu, Z., Kyrillidis, A., Shrivastava, A.: Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. Advances in Neural Information Processing Systems **36**, 52342–52364 (2023)
12. Narayan, S., Cohen, S.B., Lapata, M.: Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. ArXiv **abs/1808.08745** (2018)

13. Ribar, L., Chelombiev, I., Hudlass-Galley, L., Blake, C., Luschi, C., Orr, D.: Sparq attention: Bandwidth-efficient llm inference. arXiv preprint arXiv:2312.04985 (2023)
14. See, A., Liu, P.J., Manning, C.D.: Get to the point: Summarization with pointer-generator networks. arXiv preprint arXiv:1704.04368 (2017)
15. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)
16. Xiao, C., Zhang, P., Han, X., Xiao, G., Lin, Y., Zhang, Z., Liu, Z., Sun, M.: Infllm: Training-free long-context extrapolation for llms with an efficient context memory. arXiv preprint arXiv:2402.04617 (2024)
17. Xiao, G., Tian, Y., Chen, B., Han, S., Lewis, M.: Efficient streaming language models with attention sinks. arXiv preprint arXiv:2309.17453 (2023)
18. Xu, R., Xiao, G., Huang, H., Guo, J., Han, S.: Xattention: Block sparse attention with antidiagonal scoring. arXiv preprint arXiv:2503.16428 (2025)
19. Yuan, J., Gao, H., Dai, D., Luo, J., Zhao, L., Zhang, Z., Xie, Z., Wei, Y., Wang, L., Xiao, Z., et al.: Native sparse attention: Hardware-aligned and natively trainable sparse attention. arXiv preprint arXiv:2502.11089 (2025)
20. Zeng, Z., Lin, B., Hou, T., Zhang, H., Deng, Z.: In-context kv-cache eviction for llms via attention-gate. arXiv preprint arXiv:2410.12876 (2024)
21. Zhang, J., Xiang, C., Huang, H., Wei, J., Xi, H., Zhu, J., Chen, J.: Spargeattn: Accurate sparse attention accelerating any model inference. arXiv preprint arXiv:2502.18137 (2025)
22. Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X.V., et al.: Opt: Open pre-trained transformer language models. arXiv preprint arXiv:2205.01068 (2022)
23. Zhang, T., Kishore, V., Wu, F., Weinberger, K.Q., Artzi, Y.: Bertscore: Evaluating text generation with bert (2020), https://arxiv.org/abs/1904.09675
24. Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., et al.: H2o: Heavy-hitter oracle for efficient generative inference of large language models. Advances in Neural Information Processing Systems **36**, 34661–34710 (2023)