Proceedings of Machine Learning Research 304, 2025 ACML 2025

KVCrush: Key Value Cache size-reduction using similarity in head-behaviour

Gopi Krishna Jha GOPI.KRISHNA.JHA@INTEL.COM
Sameh Gobriel SAMEH.GOBRIEL@INTEL.COM
Liubov Talamanova LIUBOV.TALAMANOVA@INTEL.COM
Nilesh Jain NILESH.JAIN@INTEL.COM
Intel Corporation

Editors: Hung-yi Lee and Tongliang Liu

Abstract

Key-value (KV) caching has emerged as a crucial optimization technique for accelerating inference in large language models (LLMs). By allowing the attention operation to scale linearly rather than quadratically with the total sequence length, KV caching significantly enhances generation throughput (Zhang et al., 2024b). However, due to large context lengths in the modern LLMs, the memory footprint of the KV is a huge bottleneck for model deployment directly impacting the model's batch size, hindering its ability to deliver high-throughput (Liu et al., 2024b). Existing research addresses this challenge using several techniques, such as discarding low-attention tokens, quantization, and matrix approximation which typically lead to a negative impact on the model accuracy.

In this paper, we propose *KVCrush* technology which can be combined with many KV compression technologies to improve the model accuracy at a much smaller memory. *KVCrush* provides an alternate representation scheme for key-value states, along with a low-overhead token pruning algorithm that accounts for the token distribution in the KV cache, which in turn allows for a smaller footprint while maintaining the accuracy of the model. Based on our results, *KVCrush* reduces *LongBench* Bai et al. (2024) KV Cache size by 4× with less than 1% accuracy drop and achieves state-of-the-art average accuracy with minimal overhead, incurring less than 0.5% total inference latency. *KVCrush* not only outperforms state-of-the-art importance-based token retention schemes in accuracy, but also integrates seamlessly with quantization, paging, and head-sharing techniques, requiring no retraining or architectural changes.

1. Introduction

Generative AI models, such as large language models (LLMs), have revolutionized the computing industry. These models boast an extensive number of parameters and consistently achieve state-of-the-art performance across various downstream tasks Sun et al. (2019) Jha et al. (2024) Dosovitskiy (2020) Raffel et al. (2020). However, the current trend of model size growth toward multi-trillion parameters Isaev et al. (2023) — with models growing by one estimate Gholami et al. (2024) at a staggering rate of 410× every 2 years— poses huge challenges for deploying them in practice. For instance, GPT-175B Brown et al. (2020) requires 325GB of memory just to load the model weights. Additionally, not only do inferences for these models strain the platform compute and memory resources (both in terms of bandwidth and capacity), but typically, this is also coupled with strict latency requirements (in the order of tens of milliseconds) which further complicates the problem and poses a significant engineering challenge for efficiently delivering high inference throughput while maintaining low latency for user requests.

LLMs typically consist of stacked transformer decoder layers, with the self-attention module being a critical building block. This module weighs the importance of different tokens—capturing their contextual relationships—by attending over key–value token pairs at different positions within the input sequence. Nevertheless, self-attention matrix multiplication is compute intensive and has a quadratic computational complexity with respect to sequence length Vaswani et al. (2017) which significantly impacts the inference throughput, especially for longer sequences Shi et al. (2024).

KV caching has emerged as the de facto solution to this issue, converting the time complexity of token generation from quadratic to linear, albeit with increased memory overhead proportional to the context length. This mechanism is crucial for autoregressive tasks, where the model generates one output token at a time. In the decode phase, each token depends on the key and value tensors of all previous tokens (including the input tokens' KV tensors computed at prefill and all new KV tensors computed until the current time step). To avoid recomputing these tensors for every token at each step, a KV cache stores them in memory, incrementally appending new entries to the cache as generation progresses.

$$\text{KV Memory} = 2 \cdot B \cdot N_L \cdot N_H \cdot L_{\text{seq}} \cdot D \cdot precision \tag{1}$$

The scalability of KV caching becomes a critical issue as models grow larger and more complex. The total memory used by a KV cache can be determined using Equation 1, where B is the batch size, N_L represents the number of layers in the model, N_H represents the number of attention heads used, D represents the dimensionality of the embeddings, L_{seq} is the length of context in tokens, *precision* is the number of bytes per value stored (e.g. 4B for FP32) and the factor 2 is because two matrices for K and V are needed. As a result, for a given model, the KV cache size grows linearly with the maximum sequence length in the input context and the batch size, which, in practice, can result in an enormous KV cache size.

For instance, consider the OPT-175B model with its impressive 175 billion parameters, which consumes approximately 325 GB of memory. However, when using a batch size of 128 and a sequence length of only 8K, the KV cache requires around 4608 GB of memory. This is an order of magnitude (12X) larger than the model weights themselves. The issue of KV cache size has become increasingly prominent and is a significant cost factor in deploying large language models (LLMs). This is true, especially with the recent trend of LLM models that have been meticulously developed to scale up to handle extremely long context (for example, Google's Gemini-pro-1.5 has shown to support a staggering 1M token context length Georgiev et al. (2024)

Several approaches have been proposed to mitigate the memory bottleneck associated with KV caching. Recent research have explored different optimizations of KV caching, including approaches such as low-rank decomposition of the KV cache(e.g., Dong et al. (2024)) or pruning non-essential KV cache Zhang et al. (2024b); Li et al. (2024); Liu et al. (2024b); Ge et al. (2023), however, most of these techniques struggle to maintain the accuracy of the model at a smaller KV cache footprint when compared to the full model.

To address this challenge, we introduce *KVCrush*, a novel KV cache optimization that provides an alternate representation scheme for key-value states, along with a low-overhead token pruning algorithm that accounts for the token distribution in the KV cache. *KVCrush*, in turn, allows for a smaller footprint while maintaining the accuracy of the model and can be easily combined with many KV cache compression technologies. It is a modular add-on that augments existing methods by preserving contextual diversity, and requires no model retraining or architectural changes. Furthermore, *KVCrush* also works seamlessly with KV cache paging schemes (such as vLLM Kwon et al. (2023)) and mixed precision quantization Li et al. (2023) typically used in practical deployments. The technical contributions of this work can be summarized as follows:

- **Hardware-Efficient Alternative Representation of Tokens:** We leverage attention score patterns across heads to generate a binary feature vector for each token. This binary alternative representation is much smaller than the original key and value vectors, yet it preserves enough semantic information, to convey token importance and similarities, which we use to prune or retain the tokens very efficiently.
- **Accuracy-aware Low-Overhead KV Optimization Algorithm:** Our algorithm leverages the alternative binary representation of tokens and an *anchor point* to bucketize tokens into representative groups, using very efficient low-overhead distance calculations that scale linearly with the number of tokens. This approach ensures that different token groups are represented in the reduced KV memory footprint, thereby maintaining the inference accuracy of the model.
- **Practical Efficiency and Integration:** We show that *KVCrush* can be **efficiently combined** with token eviction-based KV cache compression methods. It achieves state-of-the-art average accuracy on *LongBench*, with less than 0.5% inference latency overhead, and offers up to 4× cache reduction with under 1% accuracy drop compared to the full cache.

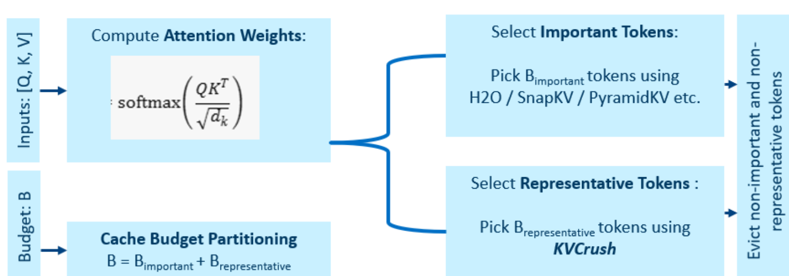


Figure 1: KVCrush flow: cache budget B is split into $B_{important}$ (pivotal tokens via baseline methods) and $B_{representative}$ (representative proxies selected using lightweight grouping).

2. Related Work

Techniques to reduce the size of KV Caches have received substantial research attention in recent years. Broadly speaking, we can divide these into three categories:

Algorithm 1: Generate Binary Token Representation Using Per-Head Thresholding

Input: Query matrix $Q \in \mathbb{R}^{S \times D \times H}$, Key matrix $K \in \mathbb{R}^{S \times D \times H}$, thresholds $\theta_1, \dots, \theta_H$

Output: Binary vector $b_t \in \{0,1\}^H$ for each token $t \in \{1, \dots, S\}$

```
Initialize b_1, ..., b_S as zero vectors in {0,1}^H
foreach head h ∈ {1, ..., H} do
    Compute attention weights: A_h = softmax(Q_h K_h^T / sqrt(d_k)) ∈ R^{S×S}
    foreach token t ∈ {1, ..., S} do
        Compute normalized attention score: w_h(t) = (1/S) Σ_{j=1}^{S} A_h(t, j)
        if w_h(t) ≥ θ_h then
            b_t[h] ← 1
        else
            b_t[h] ← 0
        end
    end
end
return b_t for all tokens t
```

Compute attention weights: $A_h = \text{softmax}\left(\frac{Q_h K_h^\top}{\sqrt{d_k}}\right) \in \mathbb{R}^{S \times S}$

Compute normalized attention score: $w_h(t) = \frac{1}{S}\sum_{j=1}^{S} A_h(t,j)$

2.1. Quantizing KV Cache

Quantization is the process of reducing the precision of a model's parameters and activations to lower bit-widths to save memory and computational resources. Quantization can be categorized into post-training quantization (PTQ) and quantization-aware training (QAT), with PTQ often preferred for large language models due to its lower resource requirements. Xiao et al. (2023a), Liu et al. (2024a), and Sheng et al. (2023) demonstrated that quantizing queries, keys, and values to INT8 enables efficient attention operations. However, these methods apply quantization uniformly across all KV cache tokens, and thus will typically negatively impact the model generation accuracy.

On the other hand, research work such as Wang et al. (2019), Yao et al. (2021), Chauhan et al. (2023), Yang et al. (2024) employ mixed precision quantization to allocate different number of bits to various model components or tensors, and thus enable a more efficient compression. This methods leverage the insight that different parts of a model exhibit varying levels of sensitivity to quantization.

As we discuss later in detail, *KVCrush* operates directly on the KV cache to determine which tokens to retain or evict. Since it does not alter the cache format itself, it is naturally complementary to quantization techniques. In fact, KVCrush can work seamlessly with mixed-precision KV caches by assigning different bitwidths to retained tokens based on their importance.

2.2. Sharing KV Cache

Multi-Query Attention (MQA) and Grouped Query Attention (GQA) are techniques developed to address the memory footprint issues LLMs allowing to share KV caches across heads. MQA, introduced by (Shazeer, 2019), reduces memory usage by sharing key and value representations across all attention heads, which enhances memory efficiency and inference speed, at the expense of generation quality. GQA, proposed by (Ainslie et al., 2023),

Algorithm 2: Token Grouping and Representative Selection via Hamming Clustering
Input: Binary vectors $b_1, b_2, \ldots, b_S \in \{0,1\}^H$ for S tokens, Number of buckets $B_{\text{representative}}$
Output: Selected representative tokens R
Generate an anchor vector $a \in \{0,1\}^H$ using a chosen strategy (e.g., random, mean of input vectors, or alternating 0-1)
Initialize $B_{\text{representative}}$ empty buckets
foreach *token* b_t **do**
  Compute Hamming distance from a: $d_t = \text{Hamming}(b_t, a)$
  Assign b_t to one of $B_{\text{representative}}$ buckets based on d_t
end
foreach *bucket* **do**
  Compute the centroid of all binary vectors in the bucket
  Select the token whose binary vector is closest to the centroid as representative
  Add representative to R
end
return R

extends this concept by grouping query heads to share KVs, balancing memory reduction with better performance retention. However, GQA involves higher training costs and complexity. Both methods offer trade-offs between memory efficiency and model performance, with MQA favoring memory savings and GQA providing a more balanced approach.

Intuitively, *KVCrush* is completely orthogonal to these KV cache sharing schemes. It operates directly on the deployed cache, regardless of whether key and value states are shared or grouped across attention heads.

2.3. Evicting inconsequential Keys and Values

This category is the closest related work to *KVCrush*. In this research category different methods aim to prune key-value (KV) pairs from cache after input processing, aiming to enhance decoding efficiency. By evicting tokens out of KV cache, memory consumption is reduced, facilitating support for larger batch sizes and longer context windows.

Different strategies for KV pruning and selectively dropping tokens from the KV cache have been proposed in recent research work, For example, in StreamLLM Xiao et al. (2023b), only the most recent tokens and attention sinks (first few tokens) are retained. H2O Zhang et al. (2024b) and Scissorhands Liu et al. (2024b) utilize attention-based metrics to determine eviction, with H2O summing attention weights and Scissorhands considering the frequency of attention surpassing a threshold. FastGen Ge et al. (2023) combines these methods with heuristics for special tokens. SnapKV Li et al. (2024) similarly depends on attention weights to prune the KV cache, but addresses the efficiency during prefill with long context, and due to the complexity of computing attention score for the whole context, it limits the observation window to the final tokens of the input prompt, thus reducing the complexity from $O(L^2)$ to $O(L)$, where L is the context length, while using max pooling to retain neighboring KVs. PyramidKV Zhang et al. (2024a) builds on SnapKV by configuring variable eviction rates across layers, with more aggressive pruning in later layers where attention is less evenly distributed. DynamicKV Zhou et al. (2025) adaptively resizes KV caches across layers based on task type. LaCache Shi et al. (2025) introduces a ladder-shaped caching pattern that prioritizes temporal coverage across layers.

The common drawback across these existing methods is that model accuracy drops as the KV compression ratio increases. These works focus solely on selecting a set of pivotal tokens (using varying algorithms) and evict the rest, which reduces KV cache size but at the cost of generation quality. *KVCrush* addresses this by explicitly representing groups of evicted tokens via a small set of representative tokens, preserving context diversity and mitigating accuracy loss under high compression. It ensures that different token groups are reflected in the reduced KV memory footprint, thereby sustaining accuracy. As a result, *KVCrush* can be combined with many existing methods to remedy accuracy drop at the same KV cache budget. We present results in Section 4, combining *KVCrush* with state-of-the-art techniques and demonstrating improved performance at equal compression ratios.

Benchmarks	**LongBench**	**lm-eval-harness**
Experiments	Section 4.2, Section 4.3.2	Section 4.3.1, Section 4.3.3, Section 4.3.4, Section 4.3.5
Datasets	narrativeqa, qasper, multifieldqa_en, hotpotqa, 2wikimqa, musique, gov_report, qmsum, multi_news, trec, triviaqa, samsum, passage_count, passage_retrieval_en, repobench-p	GSM8K and XSUM
Models	Mistral-7B-Instruct-v0.2, Meta-Llama-3-8B-Instruct	Phi-3-mini-4k-instruct, Meta-Llama-3-8B-Instruct, Llama-2-7b-chat-hf
Baselines	FullKV, H2O, SnapKV, PyramidKV	H2O
Paging Mode	Token Level, Chunk Level	Page Level (Page Size = 32)
Total Cache Budget (tokens)	2048	672 = 32(initial)+512(middle)+128(recent)
Cache Budget Partitioning	25% KVCrush + 75% Baseline	25% KVCrush + 75% Baseline
Cuda Version	12.2	12.2
Pytorch Version	2.4.1+cu121	2.4.1+cu121

Table 1: Experimental settings used for evaluation

	Cache Budget: 512	**Phi-3-mini-4k-instruct**		**Meta-Llama-3-8B-Instruct**		**Llama-2-7b-chat-hf**	
	hh-cl	**Strict**	**Flexible**	**Strict**	**Flexible**	**Strict**	**Flexible**
H2O	512-0	70.7	79.3	74.9	74.7	0.209	0.225
kvcrush.random	128-384	75.4	80.9	76.2	76.2	0.211	0.226
kvcrush.mean	128-384	75.2	80.6	76.5	76.4	0.21	0.229
kvcrush.alternate	128-384	74.6	80.9	75.7	75.6	0.212	0.227

Table 2: GSM-8K Accuracy using different anchor points in KVCrush. KVCrush outperforms the baseline H2O even using generic anchor points like random, mean and alternate 0s and 1s. Here hh and cl represents the cache budget used by H2O and KVCrush respectively.

3. KVCrush

The basic flow of KVCrush is shown in Figure 1. Given a certain total budget B for the KV Cache, this is split into two smaller portions, $B_{important}$ represents the cache budget available to store the set of pivotal tokens. This set is determined based on the specifics of KV cache compression algorithm used (e.g. H20, SnapKV, PyramidKV, etc.). While $B_{representative}$ represents the cache budget available to store along a set of representative tokens, these act as proxies of the evicted tokens, and are selected based on low-overhead grouping algorithm (discussed in Section 3.2) to ensure better representation of the whole context.

3.1. KVCrush Alternative Token Representation

In the KV cache tokens are represented as floating-point vectors of size D each. D is the size of embedding length and is typically not small. For example, for GPT-3 Brown et al. (2020) D is 2048 dimensions while for LLaMA-65B AI (2023) it is set at 4096 dimensions. As previously mentioned, KVCrush will try to group tokens in order to select representative tokens from each group. As a result, it is essential to minimize the overhead of the grouping algorithm. The running time of any clustering algorithm in D dimensions will be proportional to the value of D.

To minimize this overhead, KVCrush tries to have an alternative representation of each token with a vector of length $<< D$. The main challenge is how to find this smaller representation and yet still preserve sufficient semantic information of the tokens to be able to differentiate and group them based on their contextual relationship.

KVCrush draws some insights of alternative token representation from the FastGen research Ge et al. (2023). The authors in this work demonstrated that distinct attention heads exhibit unique structural patterns and attend to different tokens differently. For example, some attention heads will attend more to special tokens, others will attend more to locality among tokens, others will attend more to punctuation, etc.

Building on these findings we can deduce that the attention score of a certain token across H heads is a vector of length H and will represent good semantic information about the contextual properties of that token. Keeping in mind that $H << D$, where for example, H is 96, 128 for GPT-3 Brown et al. (2020) and LLaMA-65B AI (2023), respectively.

To reduce the grouping overhead even more, KVCrush takes the alternative token representation one step further. Instead of using a floating point vector of size H each, it converts that into a binary vector of size H replacing the floating point with a single bit. The main insight is that, for a given head and attention score based token eviction algorithm, the binary decision of whether to retain or evict a token encodes important contextual signals (e.g., attention score, recency), which we use to build token-level semantic fingerprints.

Algorithm 1 depicts how *KVCrush* generates a binary representation of size H for each input token. The process can be summarized in the following steps:

- **Compute the attention weight matrix:** For each head h, compute the attention weight matrix $A_h \in \mathbb{R}^{S\times S}$ using the query and key projections. Note that this attention computation is already performed during inference, and *KVCrush* leverages these values without recomputation.

- **Apply a per-head threshold:** For each token t and head h, compute the row-wise normalized attention score $w_h(t) = \frac{1}{S}\sum_{j=1}^{S} A_h(t, j)$. A token is marked with a bit value of 1 if $w_h(t) \geq \theta_h$, and 0 otherwise. The threshold θ_h is chosen such that it retains a target fraction of tokens for each head, based on desired compression ratio.

- **Form the binary representation:** For each token t, collate the H bit values across all heads into a binary vector $b_t \in \{0, 1\}^H$, which captures its cross-head importance signature.

3.2. KVCrush Token Grouping and Cache Eviction

As previously mentioned, KVCrush will select $B_{representative}$ tokens that act as proxies of the evicted tokens based to keep in the KV Cache. The selection is based on a low-overhead grouping algorithm built on top of the alternative binary representation of tokens. Algorithm 2 depicts the main steps of this weak clustering algorithm to form token groups and can be summarized as follows:

- The clustering algorithm takes a set of S tokens each represented with a binary vector of length H as input, it selects $B_{representative}$ tokens where $B_{representative} < S$ vectors as output
- An arbitrary anchor point is first selected (ideally, the anchor point should be centrally placed to all S vectors in an H dimension space). In our experiments we present results for 3 different (low-overhead) selection of anchor points (random, alternate and mean).
- For each of the S input vectors, hamming distance is computed with respect to the anchor point. The binary representation enables computing Hamming distances on-the-fly using lightweight bitwise operations, making the grouping process extremely efficient compared to standard clustering.
- Each token is then assigned to one of the $B_{representative}$ buckets. The selected bucket is the one with the lowest hamming distance.
- After the bucketization of all S tokens into their corresponding buckets, one representative vector *nearest to the centroid* of each bucket is selected, and the remaining $(S - B_{representative})$ tokens are dropped. The retained tokens, along with the $B_{important}$ tokens preserved by the KV compression algorithm, form the final B tokens of the KV cache.
- After assigning all S tokens to their respective buckets, one representative vector is selected from each by finding the token whose binary representation is closest to that bucket's centroid. All the other $(S - B_{\text{representative}})$ tokens are dropped. Those retained tokens, in addition to the $B_{\text{important}}$ tokens retained by the KV compression algorithm, represent the final B tokens of the KV cache.

It should be mentioned however, that we described KVCrush grouping algorithm using Hamming distance computations with respect to ONE anchor point, and thus, token pruning only requires S Hamming distance comparisons instead of using $O(S^2)$ distance computations required by standard clustering algorithms. In Section 4.3.3 we present results for using a higher overhead clustering algorithm on accuracy and latency of *KVCrush*.

4. Experiments

We organize our experiments to evaluate *KVCrush* along four dimensions: accuracy and latency trade-offs under constrained cache budgets; ablation studies on anchor selection, budget partitioning, and clustering methods (including comparisons with k-means); integration with existing KV eviction methods such as H2O, SnapKV, and PyramidKV; and performance in paged KV cache settings.

	Single Document QA			Multi Document QA			Summarization			Few Shot Learning			Synthetic		Code	
	narrativeqa	qasper	multifieldqa_en	hotpotqa	2wikimqa	musique	gov_report	qmsum	multi_news	trec	triviaqa	samsum	psg_count	psg_ret_en	repobench-p	Average
	18409	3619	4559	9151	4887	11214	8734	10614	2113	5177	8209	6258	11141	9289	4206	7839
Compression (x)	9.0	1.8	2.2	4.5	2.4	5.5	4.3	5.2	1.0	2.5	4.0	3.1	5.4	4.5	2.1	4
	Mistral-7B-Instruct-v0.2 (Cache Budget: 2048)															
FullKV	26.85	33.06	49.44	43.02	27.33	18.78	32.87	24.21	27.05	71.00	86.23	42.90	2.75	86.98	54.41	41.79
H2O	25.10	30.67	48.29	40.89	25.98	15.57	28.06	23.48	26.78	**60.50**	**86.33**	42.48	2.57	82.73	52.92	39.49
H2O+KVCrush	**25.77**	**30.78**	**48.47**	**41.29**	**26.33**	**16.42**	**28.87**	**23.62**	**26.83**	65.42	86.26	**42.93**	**2.67**	**83.44**	**53.21**	**40.15**
SnapKV	26.15	**32.38**	49.54	41.66	27.54	**19.43**	29.58	23.83	26.70	71.00	86.31	**43.07**	**2.89**	85.89	53.87	41.32
SnapKV+KVCrush	**26.20**	**32.09**	**49.86**	**41.86**	**28.33**	18.85	**29.62**	**23.92**	**26.95**	**71.00**	**86.20**	42.99	2.88	**86.06**	**54.13**	**41.40**
PyramidKV	25.82	31.67	49.20	41.19	27.01	**19.37**	29.15	23.89	26.72	71.00	86.28	**43.24**	2.73	85.06	53.57	41.06
PyramidKV+KVCrush	**26.05**	**31.99**	**49.30**	**41.37**	**27.12**	18.79	**29.15**	**24.22**	**27.05**	**71.00**	**86.25**	43.01	**2.77**	**85.23**	**53.87**	**41.14**
KVCrush*	**26.20**	32.09	**49.86**	**41.86**	**28.33**	18.85	**29.62**	**24.22**	**27.05**	**71.00**	86.26	43.01	2.88	**86.06**	**54.13**	**41.43**
	LLaMa-3-8B-Instruct (Cache Budget: 2048)															
FullKV	25.56	31.95	39.71	43.56	35.63	21.18	28.58	23.27	26.75	74.00	90.48	42.30	4.80	69.25	53.92	40.73
H2O	25.42	26.43	38.87	42.82	32.91	20.02	25.09	**23.26**	26.11	58.50	90.56	41.57	5.20	69.50	54.10	38.69
H2O+KVCrush	**25.48**	**27.01**	**39.02**	**43.20**	**33.23**	**20.78**	**25.67**	23.17	**26.33**	**62.00**	**90.48**	**41.87**	**5.23**	**69.50**	**54.32**	**39.15**
SnapKV	**25.70**	**29.96**	38.93	43.90	35.05	20.44	**26.89**	**23.43**	26.17	74.00	**90.56**	41.96	5.54	69.25	**56.16**	40.53
SnapKV+KVCrush	25.62	29.48	**39.56**	**44.05**	**36.20**	**20.93**	26.24	23.16	**26.32**	**74.00**	90.54	**42.17**	**5.83**	**69.25**	55.48	**40.59**
PyramidKV	25.53	**29.89**	38.67	43.90	35.04	21.60	**26.80**	**23.51**	26.37	73.50	90.56	42.21	5.08	69.25	55.36	40.48
PyramidKV+KVCrush	**25.57**	29.48	**38.97**	**44.03**	**35.75**	**21.62**	26.21	23.28	**26.52**	**74.00**	**90.56**	**42.24**	**5.17**	**69.50**	**55.49**	**40.56**
KVCrush*	25.62	29.48	**39.56**	**44.05**	**36.20**	**21.62**	26.24	23.28	**26.52**	**74.00**	**90.56**	**42.24**	**5.83**	**69.50**	55.49	**40.68**

Table 3: Performance comparison of KVCrush with PyramidKV, SnapKV and H2O on LongBench for LlaMa-3-8B-Instruct, Mistral-7B-Instruct-v0.2. Green bold indicates highest accuracy; similar background colors group each baseline method with its corresponding KVCrush result for easy comparison. KVCrush* (when paired with the top-performing KV compression method) achieves the highest accuracy across most datasets and the best average accuracy on both the *LLaMa-3-8B-Instruct* and *Mistral-7B-Instruct-v0.2* models.

4.1. Experimental Setup

We summarize experimental setup used for Evaluation in table 1.

4.1.1. Baselines

For baseline comparison, we evaluate *KVCrush* alongside four methods: **FullKV** uses the complete KV cache with zero eviction. **H2O** (Zhang et al., 2024b) determines eviction based on the cumulative attention weights, retaining the top-n tokens per layer. **SnapKV** (Li et al., 2024) restricts the observation window to the final tokens of the prompt while employing max pooling to preserve neighboring KVs. **PyramidKV** (Zhang et al., 2024a) extends SnapKV by applying variable eviction rates across layers, pruning more aggressively in later layers where attention distributions are less uniform.

4.1.2. Datasets and Models

We evaluate *KVCrush* on 16 diverse *LongBench* datasets and the GSM-8K and XSUM tasks from *lm-eval-harness*, using *LLaMa-3-8B-Instruct*, *Mistral-7B-Instruct-v0.2*, and *Phi-3-mini-4k-instruct* models. Details of datasets, models, and input lengths are summarized in Table 1.

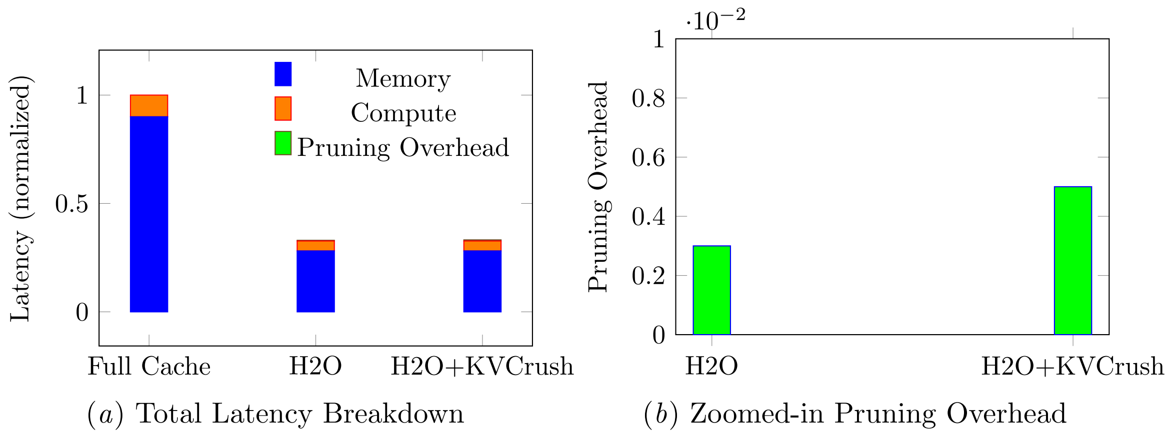


Figure 2: Latency breakdown on LongBench microbenchmark running on an Intel® Xeon® Platinum 8470 processor. H2O and H2O+KVCrush reduce KV cache size by 4×, which leads to a 3.2× reduction in memory access latency. KVCrush adds only ∼0.2% overhead while improving accuracy.

4.2. End-to-End Evaluation

4.2.1. Accuracy

The accuracy of *KVCrush*, in comparison with baseline methods, is detailed in Table 3. For this evaluation, a cache budget of 2048 was used. *KVCrush* employs a static cache budget partitioning scheme, wherein 25% of the total budget (i.e., 512) is allocated to *KVCrush* for selecting representative tokens. The remaining 75% is distributed to the accompanying method (H2O, SnapKV, or PyramidKV) for selecting high-attention tokens. The key insights are as follows:

- *KVCrush* when coupled with the best performing KV compression method (between H2O, SnapKV and PyramidKV) achieves the best accuracy for most of the datasets on both *LLaMa-3-8B-Instruct* and *Mistral-7B-Instruct-v0.2* models.
- Using *Mistral-7B-Instruct-v0.2*, it is the **only** method to achieve iso-accuracy w.r.t. FullKV baseline on *qmsum* and *multi-news* datasets.
- *KVCrush* achieves the best average accuracy on both *LLaMa-3-8B-Instruct* and *Mistral-7B-Instruct-v0.2* models.

4.2.2. Inference Latency Overhead

We evaluate the runtime latency of different KV cache strategies using a LongBench microbenchmark. The experiments were conducted on a system equipped with an Intel® Xeon® Platinum 8470 processor, simulating realistic CPU-bound server inference settings. The cache budget and associated compression parameters are detailed in Table 1.

Figure 2 shows the normalized latency breakdown across three components: memory access, compute, and pruning overhead. We observe that while the full cache baseline incurs the highest memory access latency, both H2O and H2O+KVCrush significantly reduce this cost by reducing the KV cache size by 4×. Importantly, the additional pruning overhead introduced by KVCrush remains minimal—only 0.2% higher than H2O—while achieving higher accuracy. This demonstrates that KVCrush can be deployed with negligible runtime cost while improving performance.

4.3. Ablation Study

4.3.1. Selecting Anchor Points

KVCrush employs an anchor point within the same binary space (a binary vector with a length equal to the number of heads) to form clusters of tokens based on their hamming distance to this anchor. As shown in Table 2, *KVCrush* outperforms the H2O baseline across all anchor-point strategies, including simple choices such as random, mean, and alternating 0–1 vectors. The 512-token budget shown is illustrative; similar trends are observed for other cache budgets.

4.3.2. Cache budget partitioning

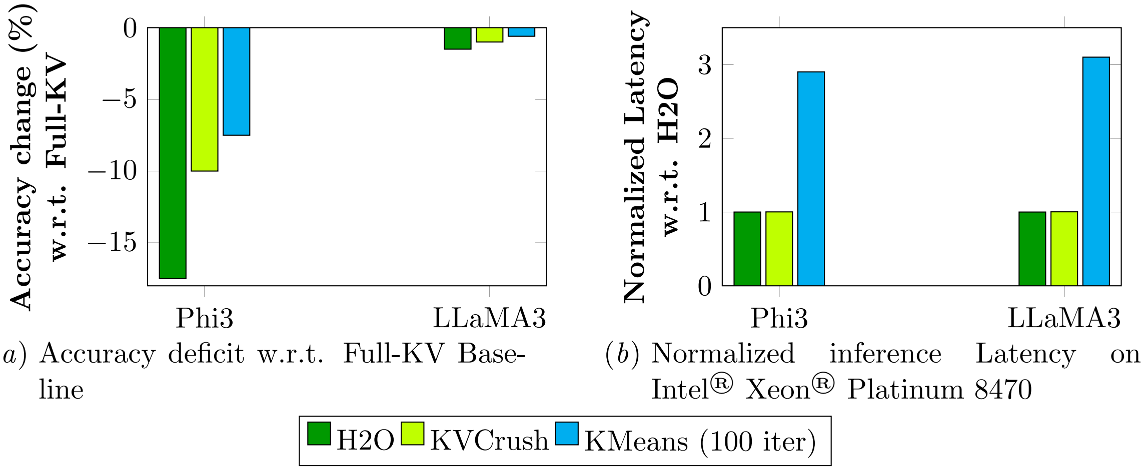
As discussed in Section 3, the cache budget B allocated for the KV Cache is divided into two distinct segments: $B_{important}$, utilized by the baseline method to identify tokens with high attention weights, and $B_{representative}$, used by *KVCrush* to select representative tokens.

Figure 4 illustrates the impact of varying the percentage of the budget allocated to *KVCrush* on overall accuracy. For certain workloads, such as *narrativeqa*, the optimal cache budget can reach up to 90% of the total budget. In contrast, for other workloads, the optimal budget typically ranges between 20% and 50% of the available budget. In Section 4.2.1, we compared our method with baseline approaches using a static cache budget partitioning scheme, allocating a fixed 25% of the total budget to *KVCrush*. This approach could potentially be enhanced by dynamically partitioning the budget based on attention weights distribution, which we plan to explore in future work.

4.3.3. KMeans vs KVCrush

Traditional clustering algorithms exhibit inefficiencies in KV Cache compression due to two primary factors. Firstly, the large size of input vectors poses a challenge. For instance, in Llama3-8B model, each input key (or value) vector comprises 128 FP16 values. By employing *kvcrush.representation*, a binary vector of length 32 (corresponding to the number of heads) is generated to represent each token (key-value pair). This approach not only reduces the data size by a factor of 64 but also enables faster distance computations using Hamming distances. Secondly, the time complexity for clustering algorithms is substantial. Utilizing *kvcrush.grouping*, the selection of values to be retained in the cache is achieved in $O(2SD)$ time, in contrast to the $O(tS^2D)$ time complexity of k-means clustering.

In Figure 3, we compare *KVCrush* and KMeans in terms of accuracy and inference latency using the GSM8K dataset. While KMeans achieves slightly higher accuracy than *KVCrush* (Figure 3(a)), it incurs approximately 200% additional overhead in the inference latency pipeline. In contrast, *KVCrush* provides a reasonable accuracy improvement over the pure H2O baseline, with an insignificant overhead of less than 0.5%.



(*a*) Accuracy deficit w.r.t. Full-KV Baseline

(*b*) Normalized inference Latency on Intel® Xeon® Platinum 8470

Figure 3: Accuracy-latency trade-off on GSM8K. KMeans offers slightly higher accuracy but with 200% more latency, while KVCrush improves H2O with negligible cost. **Note:** `Phi3` and `LLaMA3` denote `Phi-3-mini-4k-instruct` and `Meta-Llama-3-8B-Instruct`.

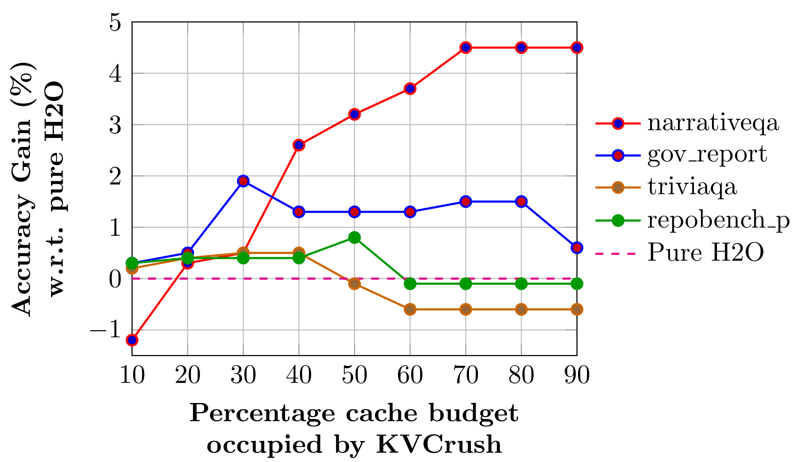
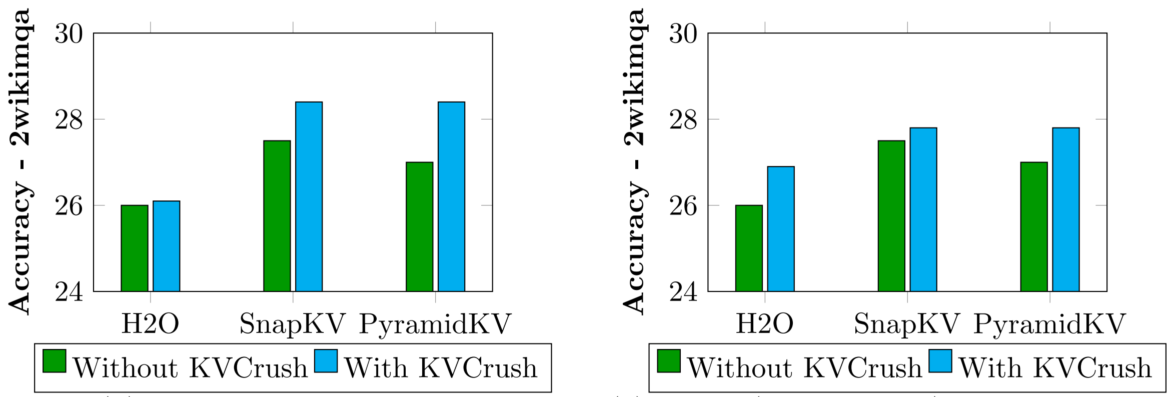


Figure 4: Accuracy gain versus percentage of a fixed total cache budget allocated to KVCrush showing the trade-off between KVCrush and baseline allocation led to an empirical sweet spot of 20–50% for most workloads, while a few (e.g., narrativeqa) benefit from higher allocations.

4.3.4. Accuracy implications of integrating KVCrush with other methods

We assess the impact on accuracy when integrating *KVCrush* with other methods, utilizing cumulative attention-weights as the importance metric. Figure 5 demonstrates the accuracy improvement provided by *KVCrush* to the baseline methods for the *2wikimqa* dataset. These baseline methods will prioritize selecting important tokens. As illustrated in Figure 1, *KVCrush* not only seamlessly integrates with these, but also improves upon their accuracies by selecting representative tokens (or pages) based on the head behavior. As shown in figure,



(a) Token Level Eviction (b) Chunk (size: 8 tokens) Level Eviction

Figure 5: Accuracy impact of integrating KVCrush with H2O, SnapKV, and PyramidKV on 2wikimqa. KVCrush improves both token and chunk-level pruning.

KVCrush also enhances the accuracy of the baseline methods even when they operate at chunks instead of individual tokens.

4.3.5. Evaluation of KVCrush in Paged KV settings

To evaluate *KVCrush* in paged KV settings, we utilized an H2O baseline that aggregates the row-wise sum of attention weights at the page level to evict low-importance pages. For *KVCrush*, the binary vector of a page is formed by concatenating the binary head vectors of all tokens within that page. Figure 6 shows that Paged-*KVCrush* outperforms paged-H2O on both GSM8K and XSUM datasets.

5. Conclusion and Future Work

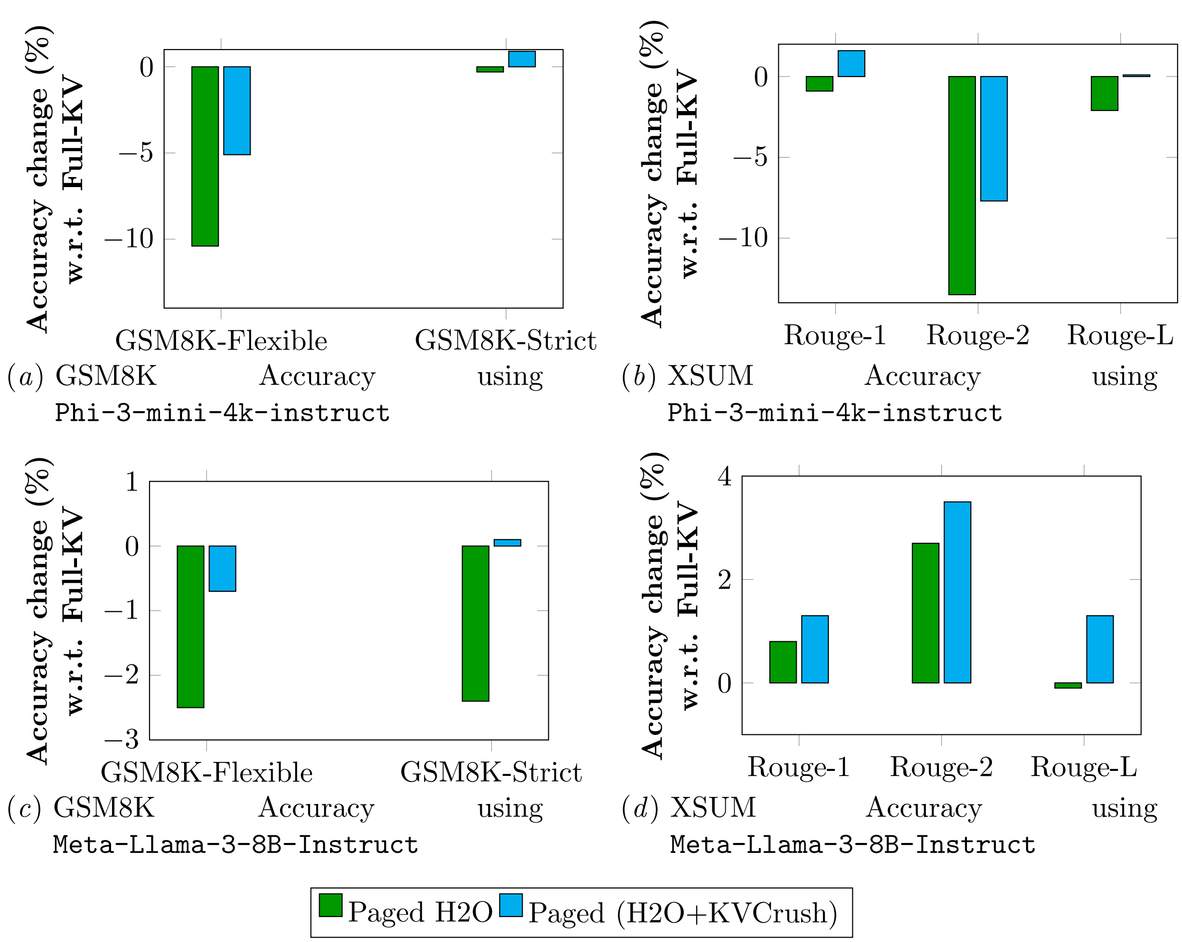
In this work, we presented an alternative approach to represent LLM tokens during inference. We showed that a compact representation paired with KVCrush compression algorithm leads to a substantial reduction in KV cache memory footprint. We demonstrated how we can use this method to achieve a memory-efficient LLM inference pipeline without compromising the quality of the generated tokens. In future work, we intend to investigate dynamic cache budget allocation and develop a more refined multi-anchoring approach.

References

Meta AI. LLaMA-65B: A 65-billion-parameter large language model, 2023. URL https://github.com/facebookresearch/llama.

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

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. Long-



(*a*) GSM8K Accuracy using `Phi-3-mini-4k-instruct`

(*b*) XSUM Accuracy using `Phi-3-mini-4k-instruct`

(*c*) GSM8K Accuracy using `Meta-Llama-3-8B-Instruct`

(*d*) XSUM Accuracy using `Meta-Llama-3-8B-Instruct`

Figure 6: Accuracy difference w.r.t. full cache for Paged H2O and Paged-KVCrush on GSM8K and XSUM. Paged-KVCrush outperforms Paged-H2O across models.

bench: A bilingual, multitask benchmark for long context understanding. *Xiv preprint arXiv:2308.14508*, 2024.

Tom Brown et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Arun Chauhan, Utsav Tiwari, and N R Vikram. Post training mixed precision quantization of neural networks using first-order information. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, pages 1343–1352, 2023.

Harry Dong, Xinyu Yang, Zhenyu Zhang, Zhangyang Wang, Yuejie Chi, and Beidi Chen. Get more with less: Synthesizing recurrence with kv cache compression for efficient llm inference. *arXiv preprint arXiv:2402.09398*, 2024.

Alexey Dosovitskiy. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

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

Petko Georgiev et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 44(2), 2024.

Mikhail Isaev, Nic McDonald, and Richard Vuduc. Scaling infrastructure to support multi-trillion parameter llm training. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2023.

Gopi Krishna Jha, Anthony Thomas, Nilesh Jain, Sameh Gobriel, Tajana Rosing, and Ravi Iyer. Mem-Rec: Memory efficient recommendation system using alternative representation. In *Proceedings of the 15th Asian Conference on Machine Learning*, pages 518–533, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *arXiv preprint arXiv:2309.06180*, 2023.

Shiyao Li, Xuefei Ning, Ke Hong, Tengxuan Liu, Luning Wang, Xiuhong Li, et al. Mixed-precision quantization for efficient llm deployment. *International Conference on Neural Information Processing Systems, NeurIPS*, 2023.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *arXiv preprint arXiv:2404.14469*, 2024.

Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. Llm-qat: Data-free quantization aware training for large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 467–484, Bangkok, Thailand and virtual meeting, 2024a. Association for Computational Linguistics.

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

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 2020.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Ying Sheng et al. Flexgen: High-throughput generative inference of large language models with a single gpu. In *Proceedings of the 40th International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

Dachuan Shi, Yonggan Fu, Xiangchi Yuan, Zhongzhi Yu, Haoran You, Sixu Li, Xin Dong, Jan Kautz, Pavlo Molchanov, Yingyan, and Lin. Lacache: Ladder-shaped kv caching for efficient long-context modeling of large language models. *arXiv preprint arXiv:2507.14204*, 2025.

Luohe Shi, Hongyi Zhang, Yao Yao, Zuchao Li, and Hai Zhao. Keep the cost down: A review on methods to optimize LLM' s KV-Cache consumption. *arXiv preprint arXiv:2407.18003*, 2024.

Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. Bert4rec: Sequential recommendation with bidirectional encoder representations from transformer. In *Proceedings of the 28th ACM international conference on information and knowledge management*, pages 1441–1450, 2019.

Ashish Vaswani et al. Attention is all you need. *International Conference on Neural Information Processing Systems, NeurIPS*, 2017.

Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023a.

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

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

Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael W. Mahoney, and Kurt Keutzer. Hawq-v3: Dyadic neural network quantization. In *38th International Conference on Machine Learning (ICML)*, 2021.

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

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2O: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2024b.

Xiabin Zhou, Wenbin Wang, Minyan Zeng, Jiaxian Guo, Xuebo Liu, Li Shen, Min Zhang, and Liang Ding. Dynamickv: Task-aware adaptive kv cache compression for long context llms. *arXiv preprint arXiv:2412.14838*, 2025.