TailorKV: A Hybrid Framework for Long-Context Inference via Tailored KV Cache Optimization

Anonymous ACL submission

Abstract

The Key-Value (KV) cache in generative large language models (LLMs) introduces substantial memory overhead. Existing works mitigate this burden by offloading or compressing the KV cache. However, loading the entire cache incurs significant latency due to PCIe bandwidth bottlenecks in CPU-GPU communication, while aggressive compression causes notable performance degradation. We identify that certain layers in the LLM need to maintain global information and are unsuitable for selective loading. In contrast, other layers primarily focus on a few tokens with dominant activations that potentially incur substantial quantization error. This observation leads to a key insight that loading dominant tokens and quantizing all tokens can complement each other. Building on this insight, we propose a hybrid compression method, TailorKV, which seamlessly integrates quantization and offloading. TailorKV develops an inference framework along with a hardware-friendly implementation that leverages these complementary characteristics. Extensive long-context evaluations exhibit that TailorKV achieves nearly lossless performance under aggressive compression settings, outperforming the state-of-the-art. Particularly, the Llama-3.1-8B with 128k context can be served within a single RTX 3090 GPU, reaching 82 ms per token during decoding.

1 Introduction

Large language models (LLMs) have demonstrated exceptional performance in tasks such as multi-turn dialogues (Chiang et al., 2023) and multi-document understanding (Bai et al., 2024). In response to the growing complexity of tasks, recent LLMs have expanded their context windows to over 128k tokens, e.g., GPT-4 (Achiam et al., 2023) and Gemini-1.5 (Team et al., 2024). Typically, the inference of LLMs is auto-regressive, with the Key-Value (KV) cache stored in memory to avoid recomputation. However, the size of KV cache grows linearly with sequence length, leading to much higher GPU memory consumption and inference latency.

Recent studies have proposed sparse attention mechanisms to reduce KV cache usage. These methods fall into two categories: irreversible eviction and recallable selection. Irreversible eviction methods (Li et al., 2024; Zhang et al., 2023; Xiao et al., 2023) suffer from **accuracy degradation** due to permanently discarding tokens that may later become crucial, particularly in multi-turn dialogues. Recallable selection methods adopt a different approach by maintaining the entire KV cache while selecting only a subset of tokens for processing. However, methods like Quest (Tang et al., 2024) and SparQ (Ribar et al., 2023) encounter **memory limitations** when attempting to store all tokens on the GPU. Although CPU offloading mitigates GPU memory limitations, existing approaches (Xiao et al., 2024; Zhang et al., 2024a) still require retrieving a substantial portion of tokens (around 20%), introducing significant **decoding latency overheads** due to slow data transfer between CPU RAM and GPU RAM.

To optimize accuracy, memory, and latency simultaneously, we first analyze the compression preferences for the KV cache based on layer characteristics. Prior researches (Feng et al., 2024; Cai et al., 2024) applied different sparsity rates to different layers under the same compression strategy. However, our analyses demonstrate that performance degradation primarily stems from the application of unsuitable compression at the layer level (Section 3). Therefore, we suggest that shallow layers, which exhibit dense attention patterns and emphasize global information (Wan et al., 2024), are better suited for uniform compression like quantization. Conversely, layers with a few dominant tokens and largely redundant information are well-suited for sparsity, as performance can be maintained by retrieving only the dominant tokens.

Building upon these insights, we propose a novel

framework, **TailorKV**, which employs hybrid compression techniques to reduce GPU memory usage. We introduce an identification metric to classify Transformer layers into two distinct types: ***(i)*** *quantization-friendly* layers, which preserve global information from a macro perspective, and ***(ii)*** *sparsity-friendly* layers, which capture crucial information from a micro perspective. This design enables *quantization-friendly* layers to employ static quantization, achieving a high compression ratio (1-bit per floating-point number) while maintaining model quality. Meanwhile, for *sparsity-friendly* layers, the system offloads the KV cache to CPU memory during prefilling and dynamically retrieves the Top-K tokens during decoding. By aligning compression strategies with the characteristics of each layer, this tailored approach significantly reduces overall memory consumption.

The accuracy and efficiency of TailorKV are evaluated on various backbone LLMs using long-context benchmarks. The results demonstrate that TailorKV drastically reduces memory usage by quantizing 1 to 2 layers to 1-bit precision and loading only 1% to 3% of the tokens for the remaining layers while maintaining nearly lossless performance. Specifically, TailorKV achieves a decoding latency of 82 ms for Llama-3.1-8B with a 128k-context on a single RTX 3090 (PCIe 1.0)[1], yielding a 53.7% reduction in peak GPU memory usage. The key contributions are summarized as follows.

- We identify layer-specific compression preferences and develop an identification metric to determine optimal compression strategies for different layers in the model.
- We present TailorKV, a hybrid KV cache compression framework that combines quantization and offloading techniques through an algorithm-system co-design, preserving both model accuracy and execution efficiency.
- Extensive experiments on long-context benchmarks demonstrate the nearly lossless performance of TailorKV with minimal GPU memory consumption and acceptable latency.

2 Preliminaries

2.1 Attention and KV Cache

LLM inference consists of two stages: ***prefill*** and ***decode***. During prefilling, the entire prompt is used to generate the first token. Consider the prompt embedding $\mathbf{X} \in \mathbb{R}^{n\times d}$ along with the weight matrices $\mathbf{W}_i^q, \mathbf{W}_i^k, \mathbf{W}_i^v \in \mathbb{R}^{d\times d_h}$ for head $i \in [1, h]$, where n is the sequence length, d is the hidden dimension and d_h is the head dimension. The keys and values for head i are computed and cached, as follows:

$$\mathbf{K}_i = \mathbf{X}\mathbf{W}_i^k,\ \mathbf{V}_i = \mathbf{X}\mathbf{W}_i^v. \tag{1}$$

During decoding, the new token embedding $\mathbf{x} \in \mathbb{R}^{1\times d}$ is computed iteratively to produce the query, key, and value vectors. The cache is updated and the output $\mathbf{o}$ of each attention head is computed as:

$$\mathbf{K}_i \leftarrow \mathrm{Cat}[\mathbf{K}_i, \mathbf{x}\mathbf{W}_i^k], \mathbf{V}_i \leftarrow \mathrm{Cat}[\mathbf{V}_i, \mathbf{x}\mathbf{W}_i^v], \tag{2}$$

$$\mathbf{a}_i = \mathrm{Softmax}\big(\mathbf{q}_i\mathbf{K}_i^\top/\sqrt{d_h}\big), \mathbf{o}_i = \mathbf{a}_i\mathbf{V}_i, \tag{3}$$

where $\mathbf{q}_i = \mathbf{x}\mathbf{W}_i^q$, and the attention outputs from all heads are concatenated and sent to the FFN.

2.2 Quantization of KV Cache

Quantization converts continuous or high-precision values into lower-precision discrete representations. Given a tensor $\mathbf{X}$ in high precision, the typical uniform quantization process can be expressed as:

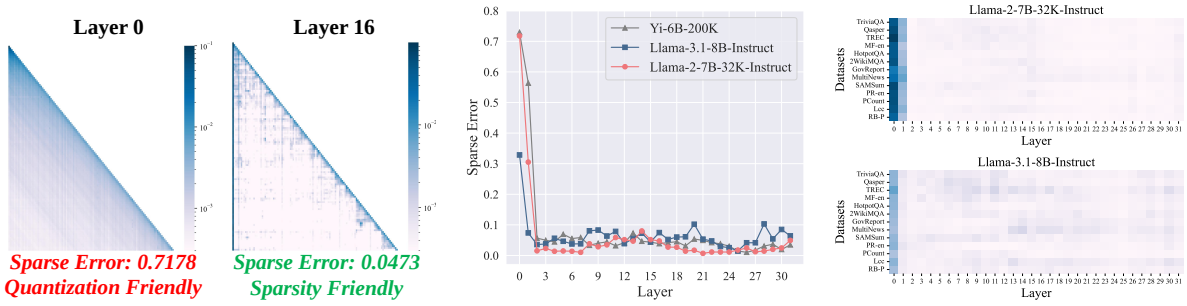
$$\begin{aligned}\mathbf{X}_Q &= \mathrm{Quant}_b(\mathbf{X}, s, z)\\ &= \mathrm{clamp}(\lfloor\frac{\mathbf{X}-z}{s}\rceil, 0, 2^b-1),\end{aligned} \tag{4}$$

where $\mathbf{X}_Q$ represents the quantized tensor in b-bit precision, with $z = \min \mathbf{X}$ as the zero point and $s = \frac{\max \mathbf{X} - \min \mathbf{X}}{2^b-1}$ as the scaler. The clamp function restricts values to the b-bit integer range and $\lfloor\cdot\rceil$ denotes the rounding function.

2.3 GPU-CPU Co-execution

As the sequence length increases, the size of the KV cache grows, significantly raising the demand for GPU resources. For example, with a sequence length of 512k, Llama-2-7B (Touvron et al., 2023) requires up to 256GB of memory for the KV cache. Current LLM serving systems (Kwon et al., 2023; Qin et al., 2024) employ an offloading strategy that stores the KV cache in cost-effective CPU memory and loads it onto the GPU during inference. However, I/O transfer latency becomes the bottleneck in inference due to the low-bandwidth PCIe interface. For instance, transferring the KV cache of a single layer ($\approx$ 8GB) from the CPU memory to the RTX 3090 GPU via PCIe 1.0 link (4GB/s) takes around 2s, while the attention computation for a single layer on the RTX 3090 GPU only takes around 10ms. Thus, on-demand fetching is currently the most common approach to reduce GPU idle time.

[1]We combine TailorKV with 4-bit weight-only quantization (Lin et al., 2024) for prefill phase memory allocation.



(a) Dense (left) and sparse (right) attention. (b) Sparse error on different LLMs. (c) Sparse error on different datasets.

Figure 1: **Observations on attention.** (a) Attention weights on Llama-2-7B-32K-Instruct. Detailed visualizations are in Appendix J. (b) Sparse error of different models on the 2WikiMQA dataset, with only the top 5% of attention scores retained. (c) Sparse error on different datasets, with only the top 5% of attention scores retained.

Strategy	RB-P	LCC	GovReport	TriviaQA
16-bit	56.7	63.4	34.9	91.6
1-bit (*KIVI*)	24.4	26.2	8.3	18.6
1-bit ($\mathbb{L} = \{0\}$)	**57.1**	**62.6**	**34.9**	**92.1**
1-bit ($\mathbb{L} = \{2\}$)	53.0	59.3	32.6	91.9
1-bit ($\mathbb{L} = \{10\}$)	52.3	59.3	31.2	90.7
1-bit ($\mathbb{L} = \{18\}$)	53.7	60.0	34.9	89.4

Table 1: Results of 1-bit quantization on different layers, using Llama-3.1-8B. $\mathbb{L}$ denotes the quantized layer.

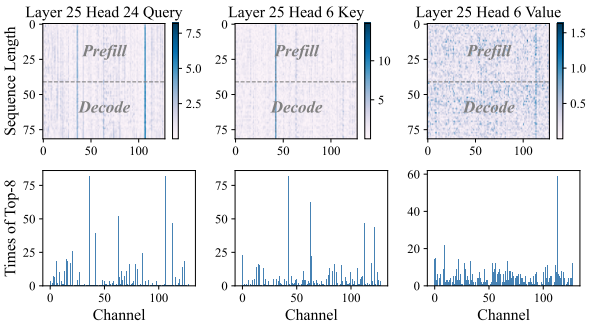


Figure 2: (Top) Query and key in Llama-3.1-8B-Instruct show outlier patterns in some channels, while the value shows no outliers. (Bottom) The number of times reaching the Top-8. Outliers may appear in any position.

3 Motivations and Observations

Layers have compression preferences. In contrast to previous belief (Li et al., 2024; Zhang et al., 2023), we propose that not all layers are suitable for sparsity. To quantify the sparsity challenges during decoding, we define the sparse error $\mathcal{E}$. Let $\mathbf{a} \in \mathbb{R}^{1\times n}$ represents the attention weight as defined in Equation 3, and let $\mathcal{M} \in \{0,1\}^n$ denote a binary mask that selects the top k elements of $\mathbf{a}$. The sparse error $\mathcal{E}$ for each head is defined as:

$$\hat{\mathbf{a}} = \mathbf{a} \odot \mathcal{M}, \quad \mathcal{E} = 1 - \textstyle\sum_{i=1}^{n} \hat{\mathbf{a}}_i. \tag{5}$$

As shown in Figure 1a, layers with dense attention distributions exhibit higher sparse errors compared to those with sparse distributions. Additionally, we observe sparse error patterns across models and datasets. Figure 1b shows that sparse errors are similar across models, with higher sparse errors in shallower layers (e.g., 0, 1). Figure 1c shows that the distribution of sparse errors remains consistent across various datasets for the same model.

Similarly, not all layers are suitable for quantization. As shown in Table 1, quantizing the KV cache to 1-bit leads to significant performance degradation. This degradation is primarily caused by layers with sparse distributions, which are more sensitive to quantization. In contrast, quantizing the dense layer (e.g., 0th) incurs no performance loss.

These findings highlight the need for a tailored KV cache compression strategy. We regard layers with dense distributions as ***quantization-friendly***, which focus on global information, and layers with sparse distributions as ***sparsity-friendly***, which prioritize crucial information.

Attention scores correlate with outliers. Each channel in the key and query contributes to the attention scores through their dot product, as expressed by the formula $\mathbf{q}\mathbf{K}^\top$. Figure 2 (Top) illustrates that some channels have large magnitudes in the query and key. It follows from the dot product formula that attention scores correlate with these outliers. A recent method (Yang et al., 2024b) focuses on static channel sparsity, utilizing offline calibration technique to identify high-magnitude channels. However, we find that the sparsity of query and key channels is dynamic rather than static. As shown in Figure 2 (Bottom), outliers in the query and key do not consistently appear in fixed posi-

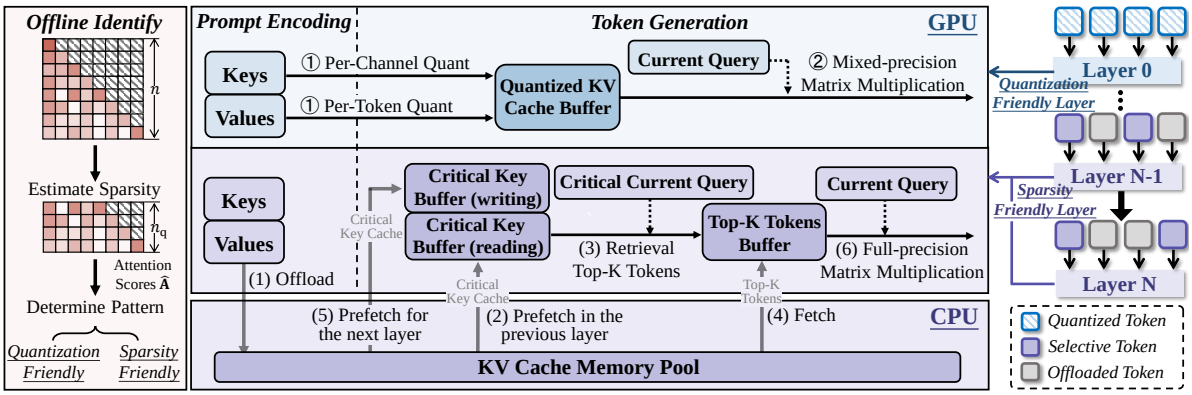


Figure 3: System overview of **TailorKV**. Offline identification categorizes the layers into quantization-friendly and sparsity-friendly. For quantization-friendly layers, we employ aggressive static quantization. For sparsity-friendly layers, we dynamically retrieve Top-K tokens. Critical current query and critical key cache represent the outliers in the query and key cache, respectively.

tions; instead, they may appear in any position. Furthermore, dynamically selecting high-magnitude channels improves the recall of dominant tokens compared to using a static offline strategy. This claim is empirically validated in Section 5.4.

4 Methodology

4.1 Offline Identification

Empirical observation in Section 3 suggests that some layers benefit more from quantization, while others are better suited for sparsity. To avoid disrupting the standard inference, we apply an offline strategy to identify the compression preference of each layer. In this phase, we introduce a metric—dense preference score $\mathcal{P}$—to assess whether each attention layer favors quantization or sparsity. Given a prompt length n, we first use the most recent n_q query vectors $\mathbf{Q}_{\text{last\_q}} \in \mathbb{R}^{n_q \times d_h}$ and the key vectors $\mathbf{K} \in \mathbb{R}^{n \times d_h}$ to compute the attention score matrix $\hat{\mathbf{A}}$ for each head during prefilling:

$$\hat{\mathbf{A}} = \text{Softmax}\big(\mathbf{Q}_{\text{last\_q}}\mathbf{K}^\top / \sqrt{d_h}\big). \quad (6)$$

Next, we select the top k indices from $\hat{\mathbf{A}}$ and sum the top k elements in order to compute the dense preference score $\mathcal{P}$:

$$\hat{\mathcal{I}} = \Big\{(i,j) \mid \text{Top}_k(\hat{\mathbf{A}}_{i,:}, k)\Big\}_{i=1}^{n_q}, \quad (7)$$

$$\mathcal{P} = n_q - \textstyle\sum_{(i,j)\in\hat{\mathcal{I}}} \hat{\mathbf{A}}_{i,j}. \quad (8)$$

If the dense preference score $\mathcal{P}_l$ of layer l exceeds the predefined threshold τ, the layer is regarded as quantization-friendly; otherwise, it is deemed sparsity-friendly. This can be formalized as:

$$C(l) = \begin{cases} \texttt{Quantization-Friendly}, & \text{if } \mathcal{P}_l > \tau, \\ \texttt{Sparsity-Friendly}, & \text{otherwise.} \end{cases} \quad (9)$$

The metric $\mathcal{P}$ consistently assesses the same model across various datasets (for details, see Appendix C). After layer-level identification, we apply ***dynamic retrieval*** for sparsity-friendly layers and ***static quantization*** for quantization-friendly layers. The overall workflow is shown in Figure 3.

4.2 Dynamic Retrieval

For sparsity-friendly layers, we propose a dynamic retrieval algorithm with an asynchronous system design. Figure 3 shows the management framework of the CPU memory pool and GPU memory buffer. To facilitate LLM inference on memory-limited devices, we offload the KV cache to lower-cost CPU memory layer by layer during prefilling. Subsequently, we retrieve the Top-K tokens on demand during decoding, thus minimizing communication overhead. The core design is illustrated in Figure 4.

As explained in Section 3, attention scores correlate with outliers in the query and key. To more accurately assess token importance, we approximate attention scores prior to original operation based on this insight. We first **estimate the critical channels** to identify outliers in the query and key cache, referred to as the critical current query and critical key cache. Since the critical key cache resides in the CPU, we employ prefetching to load it in advance. We leverage inter-layer similarity to predict the critical channels ahead of time (for a detailed explanation, see Appendix B). The similarity between adjacent layers arises from the residual connection, as validated in prior research (Lee et al., 2024). At layer $l-1$, we estimate the query $\hat{\mathbf{q}}$ for layer l, using the weight matrix from layer l and the hidden state from layer $l-1$. The contribution of

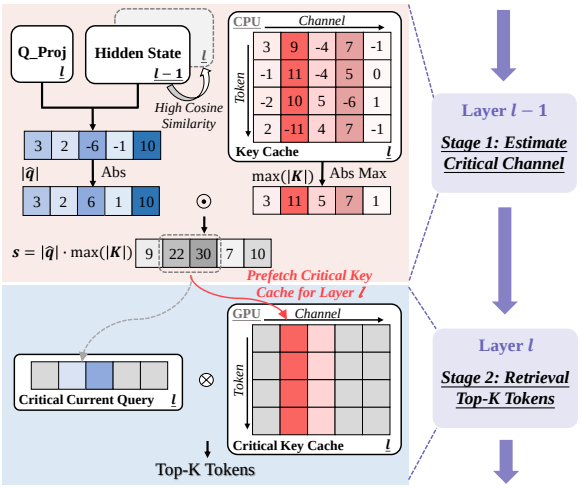


Figure 4: Two-stage dynamic retrieval process: **Stage 1** estimates critical channels at layer $l-1$ and prefetches critical key cache for layer l. **Stage 2** approximates attention scores and selects Top-K tokens at layer l.

the i-th channel to the attention scores is computed via element-wise multiplication of $\hat{\mathbf{q}}$ and $\mathbf{K}$:

$$\mathbf{s}_i = |\hat{\mathbf{q}}_i| \cdot \max(|\mathbf{K}_i|), i = 1, 2, ..., d_h. \quad (10)$$

Next, we prefetch the l-th layer's critical key cache based on $\mathbf{s}$, using double buffering—one buffer for writing and the other for reading—to enable concurrent execution. Then, we **retrieve the Top-K tokens** by approximating the attention scores at l-th layer based on the critical current query and the critical key cache, followed by fetching the Top-K tokens. Figure 5 outlines the computation and communication during decoding. The only non-overlappable operation is fetching Top-K tokens, as it depends on the current layer's query. TailorKV demonstrates how a heterogeneous design overcomes resource constraints by leveraging CPU-GPU co-execution.

4.3 Static Quantization

Unlike traditional quantization methods (Liu et al., 2024c; Yang et al., 2024a; He et al., 2024), TailorKV focuses on ensuring that each layer "plays its role," thus enabling more aggressive compression scheme, such as 1-bit quantization. As illustrated in Figure 2, outliers are present in the key cache along the channel dimension, while the value cache contains no outliers. For quantization-friendly layers, we apply static per-token quantization to the value cache and per-channel quantization to the key cache (Liu et al., 2024c). As shown in Equation 4, we introduce a 1-bit quantization kernel and also implement FP16×INT1 GEMV.

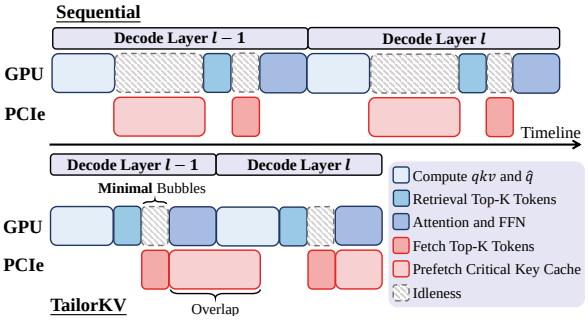


Figure 5: Timeline of dynamic retrieval. Blue signifies computation and pink signifies communication.

4.4 Memory Footprint Analysis

Let the number of layers be L, the number of heads be h, the sequence length be n, and the head dimensions be d_h. All input tokens are represented in FP16. We demonstrate a comparison of memory usage on the GPU for different methods in Table 2. TailorKV mainly manages a **quantized KV cache buffer** in quantization-friendly layers and a **critical key buffer** in sparsity-friendly layers.

Method	Memory	Parameters
Original	$2Lnhd_h$	-
SnapKV	$2\alpha Lnhd_h$	budget: α
Quest	$2Lnhd_h(1+\frac{1}{\beta})$	page size: β
Ours ($\mathbb{Q}$)	$2l_q nhd_h(\frac{1}{16}+\frac{2}{g})$	bit size: 1, group size: g, num layers of $\mathbb{Q}$: l_q
Ours ($\mathbb{S}$)	$2nhd_s$	num critical channel: d_s

Table 2: Comparison of memory usage among different methods. The symbols $\mathbb{Q}$ and $\mathbb{S}$ denote the quantization-friendly layer and sparsity-friendly layer, respectively.

5 Experiments

5.1 Experimental Setup

Baselines and Benchmarks. We evaluate three widely used models with their respective context lengths: Llama-3.1-8B-Instruct (Dubey et al., 2024), Yi-6B-200K (01-ai, 2024a), and Yi-9B-200K (01-ai, 2024b). To demonstrate the superior performance of our method, we compare TailorKV with competitive baselines, including StreamingLLM (Xiao et al., 2023), SnapKV (Li et al., 2024), Quest (Tang et al., 2024), and PQ-Cache (Zhang et al., 2024a). To evaluate the performance in long-context scenarios, we employ three well-designed benchmarks, including LongBench (Bai et al., 2024), InfiniteBench (Zhang et al., 2024b), and RULER (Hsieh et al., 2024). Refer to Appendix G for further details.

Methods	LongBench								InfiniteBench						
	Tokens	SD.QA	MD.QA	Summ	FS.L	Code	Synth	Avg.	Tokens	Retr	Dia	Novel	Math	Code	Avg.
Llama-3.1-8B	128k	49.6	50.9	31.2	69.4	60.0	53.5	53.8	128k	99.6	19.0	30.2	34.0	22.8	44.0
StreamLLM	192	26.3	42.7	17.9	50.0	48.2	53.5	40.6	1024	3.2	7.0	23.7	34.0	22.8	18.3
SnapKV	192	35.2	48.1	20.2	56.5	52.8	52.5	45.2	1024	**96.6**	9.5	27.4	34.0	22.8	41.0
Quest	192	40.1	46.9	20.7	61.6	48.0	52.4	46.2	1024	64.4	14.0	25.7	34.0	**25.1**	33.8
PQCache	192	48.4	49.5	27.0	67.3	56.3	**53.6**	51.7	1024	5.5	15.0	27.5	34.0	23.3	21.5
TailorKV-1	64(+128)	48.2	**50.9**	29.2	68.1	**58.3**	53.4	52.6	128(+896)	86.5	18.0	28.9	**34.0**	22.8	40.4
TailorKV-2	64(+128)	**49.3**	50.5	**29.4**	**68.7**	58.1	53.3	**52.9**	128(+896)	94.8	**18.5**	**30.0**	**34.0**	22.8	**42.6**
Yi-9B	200k	36.6	44.7	28.8	60.6	69.6	35.0	47.0	200k	100.0	2.5	25.2	23.4	26.3	39.2
StreamLLM	192	21.3	33.6	11.0	44.1	51.8	14.7	30.6	1024	1.5	2.5	24.2	23.7	21.3	16.4
SnapKV	192	25.0	38.8	11.9	49.0	59.7	18.8	35.0	1024	59.0	3.0	24.9	22.5	**26.6**	30.0
Quest	192	29.2	37.9	15.4	57.5	59.6	25.7	39.1	1024	98.4	4.0	21.8	18.2	18.7	36.1
PQCache	192	32.4	41.6	19.2	58.6	64.4	**27.8**	42.0	1024	7.8	2.0	25.3	22.2	25.6	18.5
TailorKV-1	64(+128)	**38.0**	**44.3**	**27.3**	**60.2**	**66.3**	24.3	**44.7**	128(+896)	**98.7**	2.5	**26.6**	**24.0**	21.3	39.2
TailorKV-2	64(+128)	35.6	43.5	**27.3**	60.1	66.0	23.5	44.0	128(+896)	98.5	**4.5**	25.3	**24.0**	24.9	**39.4**
Yi-6B	200k	32.4	15.3	1.3	49.9	69.8	9.5	29.7	200k	99.2	0.0	25.2	6.8	26.9	37.1
StreamLLM	192	20.0	11.6	**1.6**	34.0	44.6	4.0	20.4	1024	2.0	0.0	21.8	4.8	25.8	13.5
SnapKV	192	24.2	13.0	**1.6**	38.5	51.2	3.7	23.3	1024	55.7	2.5	23.2	4.5	**26.9**	26.4
Quest	192	26.5	12.5	0.3	46.9	51.9	**8.5**	26.2	1024	**99.5**	3.0	22.0	5.1	26.6	35.8
PQCache	192	30.4	14.5	0.6	48.0	55.8	4.0	27.3	1024	6.9	1.5	24.6	5.7	**26.9**	16.3
TailorKV-1	64(+128)	**32.5**	**15.4**	1.4	**49.7**	55.9	4.0	**28.3**	128(+896)	98.7	2.5	**25.3**	7.7	26.4	37.2
TailorKV-2	64(+128)	**32.5**	15.3	1.5	49.1	**56.4**	4.0	28.2	128(+896)	98.5	**3.0**	**25.3**	**8.0**	26.7	**37.3**

Table 3: Task performance (%) on **LongBench** and **InfiniteBench**. 13 sub-tasks of LongBench are aggregated into 6 classes, and 9 sub-tasks of InfiniteBench are aggregate into 5 classes. The aggregation of sub-tasks is discussed in Table 10 and Table 11, while the detailed results for all sub-tasks can be found in Table 14 and Table 15.

Implementation. We set τ to 0.2 for all models. TailorKV-1 and TailorKV-2 represent KV cache stored with 1-bit and 2-bit precision in quantization-friendly layers, respectively. The group size is 64, with the zero point and scaler stored in 16-bit. For sparsity-friendly layers, the number of tokens involved in attention computation is $n_{\text{local}} + (n_{\text{topk}})$, where n_{local} refers to the GPU budget and n_{topk} represents the additional communication overhead. The number of critical channels is 8 for LongBench and 12 for both InfiniteBench and RULER. The symbol $\mathbb{Q}$ represents quantization-friendly layers. Llama-3.1-8B is configured with $\mathbb{Q} = \{0\}$, while Llama-2-7B, Yi-6B, and Yi-9B are configured with $\mathbb{Q} = \{0, 1\}$. Additional details are in Appendix D.

Hardware. The experiments are conducted under two different settings: the first equipped with an NVIDIA RTX 3090 GPU (24GB) and Intel Xeon Gold 6240 CPU, interconnected via PCIe 1.0 ×16 (4GB/s); the second equipped with an NVIDIA A100 GPU (80GB) and Intel Xeon Platinum 8369B CPU, interconnected via PCIe 4.0 ×16 (32GB/s).

5.2 Accuracy on Long Context Tasks

LongBench. As shown in Table 3, SnapKV and StreamingLLM degrade in performance due to the loss of crucial information. Although Quest and PQCache improve performance, their individual strategies face limitations under restricted budgets. TailorKV outperforms the best method by 2.32%, 5.42%, and 3.66% on Llama-3.1-8B, Yi-9B, and Yi-6B, respectively, by preserving the 1-bit KV cache for quantization-friendly layers and selecting 192 tokens for sparsity-friendly layers. Appendix F provides a discussion on retrieval accuracy of our sparsity-friendly layers compared to other methods.

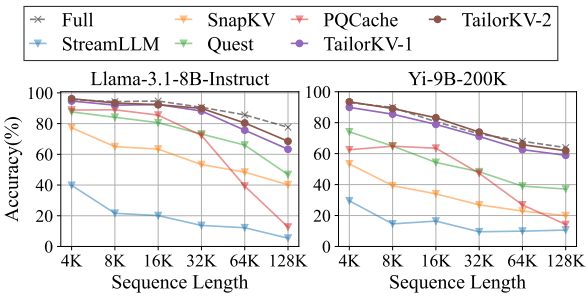


Figure 6: The average accuracy of different methods on **RULER**. The sparsity-friendly layer in TailorKV uses 128+(896) tokens, while other methods use 1024 tokens. See Table 16 for details.

InfiniteBench. Table 3 presents evaluations on the challenging benchmark InfiniteBench. As the context length increases, the clustering overhead of PQCache on the CPU grows. We restrict K-Means to one iteration for real-time inference, which compromises accuracy and exposes PQCache's limi-

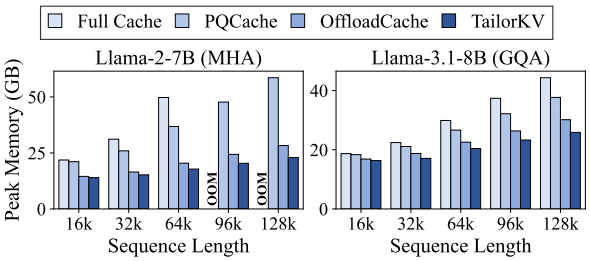


Figure 7: Peak memory usage on A100 (80GB).

Methods	Llama-2-7B				Llama-3.1-8B		
	16k	32k	64k	96k	16k	32k	64k
NVIDIA RTX 3090 (24GB, PCIe 1.0 link)							
Full Cache	OOM	OOM	OOM	OOM	0.033	0.042	OOM
OffloadCache	0.893	1.776	OOM	OOM	0.242	0.460	OOM
PQCache	OOM	OOM	OOM	OOM	0.126	OOM	OOM
TailorKV	0.067	0.087	0.135	0.176	0.062	0.067	0.103
NVIDIA A100 (80GB, PCIe 4.0 link)							
Full Cache	0.045	0.077	0.140	OOM	0.024	0.033	0.050
OffloadCache	0.433	0.838	1.767	3.253	0.124	0.227	0.435
PQCache	0.108	0.111	0.114	0.115	0.104	0.105	0.108
TailorKV	0.041	0.062	0.098	0.132	0.045	0.047	0.054

Table 4: Decoding latency(s) on different devices. Additional results are provided in Table 13.

tations. Notably, our hybrid strategy outperforms individual strategies, with an average performance loss under 1.5% compared to the full cache, especially excelling in dialogue, novel, and math tasks.

RULER. Figure 6 summarizes the accuracy on RULER, with the sequence length ranging from 4K to 128K. TailorKV captures crucial information from redundant contexts, leading to superior performance on most tasks, such as Needle-in-a-haystack, Question Answering, and Variable Tracking (detailed results provided in Table 16).

5.3 Efficiency Results

We evaluate peak memory usage and decoding latency in comparison with the full cache, OffloadCache, and PQCache. Specifically, the full cache is implemented by FlashAttention-2 (Dao, 2023) and OffloadCache is a script[2] from the official library that prefetches next layer's KV cache from the CPU memory to the GPU.

Peak Memory Usage. As shown in Figure 7, our method achieves superior memory efficiency compared to alternative methods, enabling deployment on lower-end GPUs such as the RTX 3090. Specifically, compared to full cache, TailorKV reduces

[2] https://github.com/huggingface/transformers

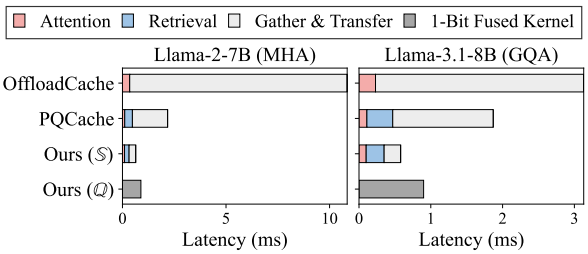


Figure 8: Latency breakdown (ms) under different methods. $\mathbb{Q}$ and $\mathbb{S}$ denote the quantization-friendly layer and sparsity-friendly layer, respectively.

GPU memory usage by approximately $\mathbf{73.8}\%$ for Llama-2-7B with a sequence length of 128k.

End-to-End Latency. As shown in Table 4, the increasing sequence length causes out-of-memory errors in the full cache, PQCache, and OffloadCache on the RTX 3090. For 64k context on the A100, TailorKV achieves significant latency reductions compared to OffloadCache and PQCache: $\mathbf{18.0}\times$ and $\mathbf{1.2}\times$ faster than the MHA model, and $\mathbf{8.1}\times$ and $\mathbf{2.0}\times$ faster than the GQA model. TailorKV's latency is comparable to that of full attention, as a result of multi-threading used to execute asynchronous tasks, which enables the overlap of computation and CPU-GPU communication.

Latency Breakdown. As depicted in Figure 8, we evaluate the breakdown of latency for a single Transformer block with a sequence length of 16k on the A100 GPU. Compared to PQCache, TailorKV reduces retrieval latency by 27.8% for the GQA model and 40.5% for the MHA model, and data transfer latency by 83.5% and 82.2% for the same models. This reduction is primarily attributable to our use of DGL (Wang et al., 2019) to directly transfer rows from a CPU tensor to the GPU device, whereas PQCache first gathers rows on the CPU and then transfers them to the GPU.

5.4 Ablation Study

We conduct ablation studies on the LongBench benchmark using the Llama-3.1-8B-Instruct model.

Effect of Tailored Strategies. As depicted in Figure 9 (a), 1-bit quantization is applied to certain layers, while only 64(+128) tokens are computed for the remaining layers, with the quantization-friendly layer defined as $\mathbb{Q} = \{0\}$. The results indicate that quantizing only the 0th layer yields the best performance, while quantizing sparsity-friendly layers degrades performance, highlighting the need for tailored compression strategies.

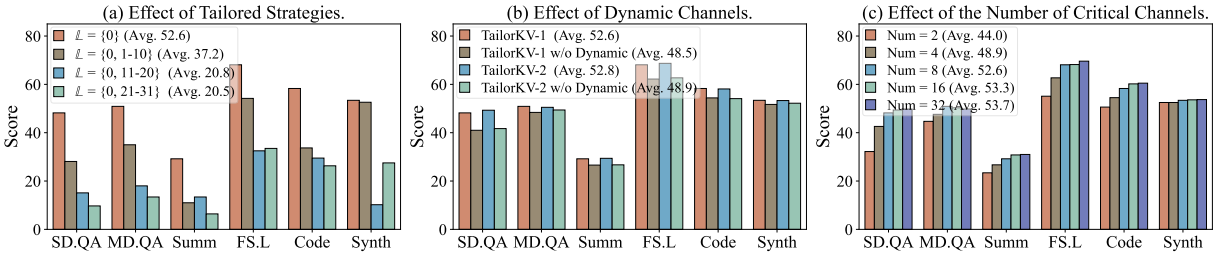


Figure 9: **Ablation studies.** (a) Performance comparison with different layers quantized to 1-bit. (b) Performance of TailorKV with dynamic or static channels. (c) Performance comparison with different numbers of critical channels.

Effect of Dynamic Channels. Prior study (Yang et al., 2024b) employed offline calibration to statically select high-magnitude channels. However, we find that outliers may appear at any position, not fixed to specific channels (Section 3). Figure 9 (b) compares the performance of dynamic and static channels. In general, our dynamic retrieval method demonstrates better performance.

Effect of the Number of Critical Channels. In Figure 9 (c), we maintain the 64(+128) configuration and adjust the number of critical channels. Reducing the number of critical channels decreases retrieval latency. However, performance significantly degrades when the number is set to 2 or 4. Overall, selecting 8 critical channels achieves a favorable balance between performance and latency.

6 Related Work

Existing KV cache compression methods include eviction, selection, and quantization, with detailed comparisons in Appendix A. Eviction methods reduce KV cache size by evicting most tokens during inference. StreamingLLM (Xiao et al., 2023) identifies 'Attention Sinks' by retaining the initial and the most recent tokens. H2O (Zhang et al., 2023), SnapKV (Li et al., 2024), and Scissorhands (Liu et al., 2023) estimate token importance based on historical attention scores. However, evicting dominant tokens may degrade accuracy in tasks like 'needle-in-the-haystack' and multi-turn dialogues.

Selection methods are more commonly used in sparse attention scenarios. Quest (Tang et al., 2024) retains the KV cache and utilizes paged keys for retrieving tokens, but it fails to reduce memory usage and suffers from lower recall. Instead, KV cache offloading methods like PQCache (Zhang et al., 2024a) and InfiniGen (Lee et al., 2024) approximate attention scores for identifying and loading critical tokens from CPU to GPU, though they face challenges in balancing computation and communication due to large KV cache loads. Some methods (Chen et al., 2024; Liu et al., 2024b) use LSH and KNN to retrieve critical tokens, which are processed on the CPU and subsequently merged with GPU outputs; however, imbalanced computation times may result in GPU idle time.

Quantization is a common compression technique that converts high-precision floats into low-precision integers. Existing methods employ various solutions to minimize quantization error. For example, KVQuant (Hooper et al., 2024) isolates outliers for mixed precision, GEAR (Kang et al., 2024) utilizes SVD to recover residuals, and KIVI (Liu et al., 2024c) quantizes keys per channel and values per token. FlexGen (Sheng et al., 2023) reduces I/O transfer latency by quantizing the KV cache to 4-bits. However, none of these methods reduce the KV cache to 1-bit. By contrast, we focus on exploring layer characteristics and selecting the most suitable compression strategy.

7 Conclusion

In this paper, we propose TailorKV, an effective framework for KV cache management in LLMs. We begin by observing that different layers exhibit distinct compression preferences and categorize them into quantization-friendly and sparsity-friendly, each employing a tailored strategy. Specifically, quantization-friendly layers aggressively quantize the KV cache to 1-bit. Sparsity-friendly layers, on the other hand, dynamically retrieve dominant tokens based on large magnitudes in the query and key channels, integrating CPU-GPU co-design. Experiments across long-context benchmarks show that TailorKV effectively minimizes the usage of the KV cache while maintaining model performance, with an acceptable latency cost. Our hybrid framework demonstrates the potential of deploying LLMs on resource-limited GPUs, extending the application of LLMs to more devices while maintaining efficiency.

Limitations

Although TailorKV has demonstrated superior memory optimization and latency reduction in long-context scenarios, it still exhibits some limitations, which are summarized as follows: (1) TailorKV primarily focuses on improving the efficiency of the decode phase by asynchronously transferring tokens from the CPU memory to the GPU. However, it is challenging to completely overlap the offloading latency during the prefill phase. Moreover, the efficiency of the prefill phase in long-context scenarios is also important. It is noteworthy that our work is compatible with and complementary to approaches for prefilling acceleration (Jiang et al., 2024). (2) We have designed tailored strategies for different layers to facilitate deployment, and we are confident that TailorKV can be adapted on a head-wise basis. These issues hold significant importance, and we intend to further explore them in our future research.

References

01-ai. 2024a. Yi-6b-200k. https://huggingface.co/01-ai/Yi-6B-200K. Accessed: 2024-07-01.

01-ai. 2024b. Yi-9b-200k. https://huggingface.co/01-ai/Yi-9B-200K. Accessed: 2024-07-01.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *ArXiv preprint*, abs/2303.08774.

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 *Proc. of ACL*, pages 3119–3137. 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*, abs/2406.02069.

Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, et al. 2024. Magicpig: Lsh sampling for efficient llm generation. *ArXiv preprint*, abs/2410.16179.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E Gonzalez, et al. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. *See https://vicuna. lmsys. org (accessed 14 April 2023)*, 2(3):6.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *ArXiv preprint*, abs/2307.08691.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *ArXiv preprint*, abs/2407.21783.

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*, abs/2407.11550.

Yefei He, Luoming Zhang, Weijia Wu, Jing Liu, Hong Zhou, and Bohan Zhuang. 2024. Zipcache: Accurate and efficient kv cache quantization with salient token identification. *ArXiv preprint*, abs/2405.14256.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *ArXiv preprint*, abs/2401.18079.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. Ruler: What's the real context size of your long-context language models? *ArXiv preprint*, abs/2404.06654.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, et al. 2024. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *ArXiv preprint*, abs/2407.02490.

Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. 2024. Gear: An efficient kv cache compression recipefor near-lossless generative inference of llm. *ArXiv preprint*, abs/2403.05527.

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*, pages 611–626.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. Infinigen: Efficient generative inference of large language models with dynamic kv cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 155–172.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. Snapkv:

Llm knows what you are looking for before generation. *ArXiv preprint*, abs/2404.14469.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. 2024a. Minicache: Kv cache compression in depth dimension for large language models. *ArXiv preprint*, abs/2405.14366.

Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, et al. 2024b. Retrievalattention: Accelerating long-context llm inference via vector retrieval. *ArXiv preprint*, abs/2409.10516.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. In *Proc. of NeurIPS*.

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

Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2024. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *ArXiv preprint*, abs/2407.00079.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. 2023. Sparq attention: Bandwidth-efficient llm inference. *ArXiv preprint*, abs/2312.04985.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single GPU. In *Proc. of ICML*, volume 202 of *Proceedings of Machine Learning Research*, pages 31094–31116. PMLR.

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

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *ArXiv preprint*, abs/2403.05530.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Zhongwei Wan, Xinjian Wu, Yu Zhang, Yi Xin, Chaofan Tao, Zhihong Zhu, Xin Wang, Siqi Luo, Jing Xiong, and Mi Zhang. 2024. D2o: Dynamic discriminative operations for efficient generative inference of large language models. *ArXiv preprint*, abs/2406.13035.

Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *ArXiv preprint*, abs/1909.01315.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. 2024. Infllm: Training-free long-context extrapolation for llms with an efficient context memory. In *Proc. of NeurIPS*.

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

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

Shuo Yang, Ying Sheng, Joseph E Gonzalez, Ion Stoica, and Lianmin Zheng. 2024b. Post-training sparse attention with double sparsity. *ArXiv preprint*, abs/2408.07092.

Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. 2024a. Pqcache: Product quantization-based kvcache for long context llm inference. *ArXiv preprint*, abs/2407.12820.

Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Hao, Xu Han, Zhen Thai, Shuo Wang, Zhiyuan Liu, et al. 2024b. bench: Extending long context evaluation beyond 100k tokens. In *Proc. of ACL*, pages 15262–15277.

Xuan Zhang, Cunxiao Du, Chao Du, Tianyu Pang, Wei Gao, and Min Lin. 2024c. Simlayerkv: A simple framework for layer-level kv cache reduction. *ArXiv preprint*, abs/2410.13846.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark W. Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2O: heavy-hitter oracle for efficient generative inference of large language models. In *Proc. of NeurIPS*.

A Comparison with Other Approaches

Figure 10 compares TailorKV with other methods: (a) Full cache retains the entire KV cache. (b) The eviction methods permanently evict specific tokens from each layer, leading to irreversible information loss since evicted tokens may be important later. (c) The selection methods offload the entire KV cache to the CPU, enabling tokens recall but incurring significant communication overhead because of the large volume of tokens involved. (d) Our method employs layer-specific compression strategies, facilitating more aggressive compression.

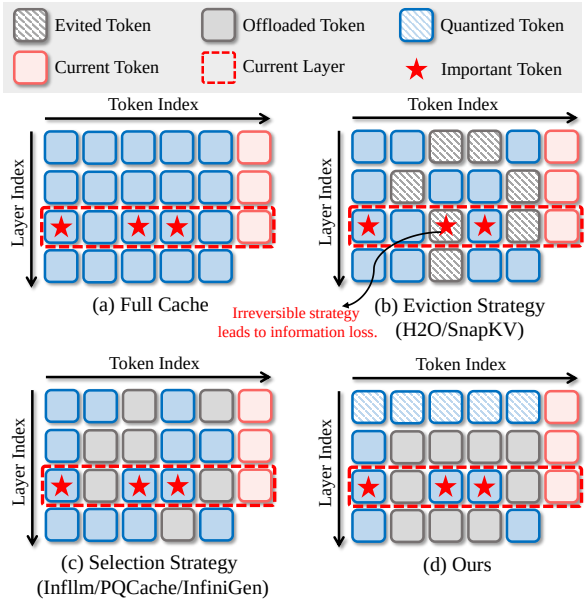


Figure 10: Comparison of TailorKV with other methods in managing KV cache budget across layers.

B Inter-Layer Similarity

Let $\mathbf{h}_l$ denote the hidden state at the l-th layer. To quantify the similarity between the hidden states of two adjacent layers, we employ cosine similarity, which is formally defined as:

$$\text{sim}(\mathbf{h}^{(l-1)}, \mathbf{h}^{(l)}) = \frac{\mathbf{h}^{(l-1)} \cdot \mathbf{h}^{(l)}}{\|\mathbf{h}^{(l-1)}\| \|\mathbf{h}^{(l)}\|}. \quad (11)$$

We define the query weight at the l-th layer as $W_q^{(l)}$. As shown in Figure 11, $\mathbf{h}^{(l)}$ and $\mathbf{h}^{(l-1)}$ closely resemble each other, allowing us to approximate the query at the l-th layer based on the hidden state from the $l-1$-th layer:

$$\hat{\mathbf{q}}^{(l)} = \mathbf{W}_q^{(l)}(\mathbf{h}^{(l-1)}). \quad (12)$$

Existing research (Liu et al., 2024a) has elucidated that the KV cache exhibits similarity across adjacent layers. However, as illustrated in Figure 11, the similarity between $\hat{\mathbf{q}}^{(l)}$ and $\mathbf{q}^{(l)}$ exceeds that between $\mathbf{q}^{(l-1)}$ and $\mathbf{q}^{(l)}$, suggesting that using hidden states from the preceding layer enhances prediction accuracy.

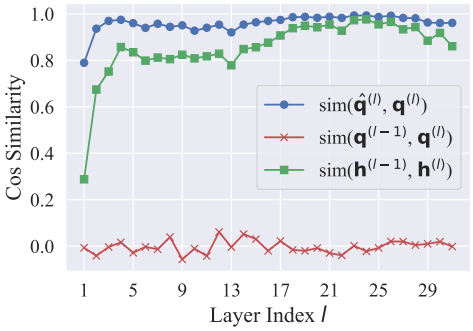


Figure 11: Cosine similarity between adjacent layers.

C Offline Identification on Different Datasets

As shown in Figure 12, the curves represent different datasets. The distribution of $\mathcal{P}$ is consistent across various datasets for the same model, indicating that the metric $\mathcal{P}$ effectively captures the characteristics of different layers, enabling appropriate compression strategy.

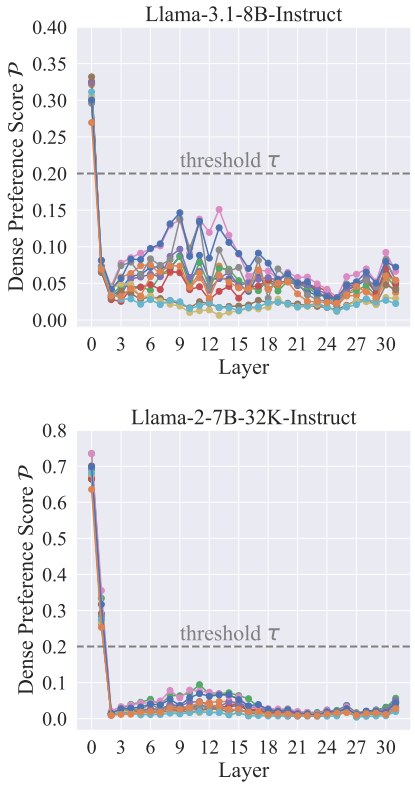


Figure 12: Dense preference score $\mathcal{P}$ for layers across different offline datasets.

Methos	Ratio	Qspr	MulFi	HQA	WMQA	GRpt	MulN	TREC	SMSM	TriQA	Repo	LCC	PsgC	PsgR	Avg.
Llama-3.1-8B	1×	45.5	53.8	54.7	47.1	34.9	27.5	73.0	43.8	91.6	56.7	63.4	7.5	99.5	53.8
SimLayerKV	1.53×	**45.6**	52.3	54.5	44.5	**32.2**	26.9	**71.5**	**43.8**	91.3	**54.9**	**62.8**	**7.9**	95.5	52.6
TailorKV-1	34.2×	43.4	53.0	**55.3**	**46.5**	31.3	**27.2**	70.0	42.5	91.6	54.8	61.8	**7.9**	**99.0**	52.6
TailorKV-2	32.7×	44.8	**53.9**	54.8	46.2	31.9	26.8	70.5	43.2	**92.3**	54.2	62.1	7.7	**99.0**	**52.9**

Table 5: Performance comparison between TailorKV and SimLayerKV. TailorKV computes only 64 (+128) tokens for sparsity-friendly layers. SimLayerKV retains the most recent 1024 tokens for the "lazy" layers, while the "non-lazy" layers preserve full precision. Additionally, the threshold for SimLayerKV on Llama-3.1-8B is 0.9, with more than half of the layers being "non-lazy."

Methods	Configurations
StreamingLLM	num local: 128, num initial: 64
SnapKV	window size: 64, max capacity prompt: 128, kernel size: 7, pooling: max pooling
Quest	page size: 16, token budget: 196
PQCache	partitions in PQ: 2, bits for PQ codes: 6, K-Means iterations: adaptive, n_{local}: 64, n_{topk}: 128
TailorKV-1	τ: 0.2, bit size: 1, group size: 64, n_{local}: 64, n_{topk}: 128, num critical channels: 8
TailorKV-2	τ: 0.2, bit size: 2, group size: 64, n_{local}: 64, n_{topk}: 128, num critical channels: 8

Table 6: Configurations of long-context methods on **LongBench**.

Methods	Configurations
StreamingLLM	num local: 896, num initial: 128
SnapKV	window size: 128, max capacity prompt: 896, kernel size: 7, pooling: max pooling
Quest	page size: 16, token budget: 1024
PQCache	partitions in PQ: 2, bits for PQ codes: 6, K-Means iterations: 1 (exceeding 64k), n_{local}: 128, n_{topk}: 896
TailorKV-1	τ: 0.2, bit size: 1, group size: 64, n_{local}: 128, n_{topk}: 896, num critical channels: 12
TailorKV-2	τ: 0.2, bit size: 2, group size: 64, n_{local}: 128, n_{topk}: 896, num critical channels: 12

Table 7: Configurations of long-context methods on **InfiniteBench** and **RULER**.

D Baselines Settings

In Table 6 and Table 7, we present the configuration for the long-context methods employed in our experiments.

E Comparison with Hybrid Method

To validate the effectiveness of our quantization-sparsity hybrid framework, we compare it to SimLayerKV (Zhang et al., 2024c), a similar hybrid method. SimLayerKV assumes that some layers in LLMs exhibit "lazy" behavior, retaining only the initial and most recent tokens, while "non-lazy" layers require full precision to retain all tokens. Table 5 presents the experimental results on LongBench. The results show that at an average compression rate of $34.2\times$, the performance of our method is comparable to that of SimLayerKV at a $1.53\times$ compression rate. Our approach achieves optimal performance with minimal memory overhead, providing strong evidence for the practicality of this quantization-sparsity hybrid architecture. In contrast, SimLayerKV requires real-time identification of layer types based on historical attention scores, making it incompatible with FlashAttention. This introduces additional computational and memory overhead, which increases latency and may cause out-of-memory issues.

F Effectiveness of Dynamic Retrieval

Table 8 presents a comparison of retrieval accuracy between our sparsity-friendly layers and alternative methods, using the Llama-3.1-8B-Instruct model on the LongBench benchmark. Specifically, we retain full precision for the KV cache in the 0th layer of StreamLLM, SnapKV, and Quest, thereby preserving the global information in the 0th layer. TailorKV-1 and TailorKV-2 represent the quantization of the 0th layer's KV cache to 1-bit and 2-bit precision, respectively.

The experimental results demonstrate that our retrieval method outperforms other sparse methods when the global information is preserved in the 0th layer. Specifically, TailorKV applies quantization to the 0th layer, whereas other methods use full precision (16-bit), and the attention calculation utilizes the same tokens from layer 1 to layer 31. This notable performance advantage highlights that our retrieval method effectively identifies the most important tokens, thereby minimizing the loss of crucial information.

Methos	Tokens	Qspr	MulFi	HQA	WMQA	GRpt	MulN	TREC	SMSM	TriQA	Repo	LCC	PsgC	PsgR	Avg.
Llama-3.1-8B	128k	45.5	53.8	54.7	47.1	34.9	27.5	73.0	43.8	91.6	56.7	63.4	7.5	99.5	53.8
StreamLLM‡	192	21.4	31.3	46.5	38.9	17.9	18.0	40.0	34.4	75.7	45.7	50.7	8.0	99.0	40.6
StreamLLM†	192	21.6	30.8	45.5	39.0	18.4	17.9	40.5	34.1	75.6	45.5	52.8	8.0	99.0	40.7
SnapKV‡	192	25.7	44.7	52.6	43.7	20.0	20.5	41.0	39.6	89.0	48.7	57.0	8.0	97.0	45.2
SnapKV†	192	32.4	47.0	54.6	44.0	21.9	22.8	48.0	40.0	90.3	51.9	59.9	8.0	98.0	47.6
Quest‡	192	35.9	44.2	52.8	41.0	17.7	23.8	63.0	36.0	86.0	43.6	52.3	**8.4**	96.5	46.2
Quest†	192	39.1	45.1	52.4	43.4	21.1	25.6	65.5	38.7	88.1	44.8	52.0	8.1	97.0	47.8
TailorKV-1	64(+128)	43.4	53.0	**55.3**	**46.5**	31.3	**27.2**	70.0	42.5	91.6	**54.8**	61.8	7.9	**99.0**	52.6
TailorKV-2	64(+128)	**44.8**	**53.9**	54.8	46.2	**31.9**	26.8	**70.5**	**43.2**	**92.3**	54.2	**62.1**	7.7	**99.0**	**52.9**

Table 8: Effectiveness of dynamic retrieval. Methods marked with † indicate that the 0th layer of the model retains the full-precision (16-bit) KV cache, while methods marked with ‡ indicate that all layers use the same compression strategy. TailorKV-1 and TailorKV-2 store the KV cache as 1-bit and 2-bit, respectively, in the 0th layer.

G More Information on Models and Benchmarks

G.1 Baselines

In all of our experiments, we use pre-trained model weights obtained from Huggingface. These models are based on two representative attention structures: (1) MHA: including Llama-2-7B-32K-Instruct[3]. (2) GQA: including Llama-3.1-8B-Instruct[4], Yi-6B-200K[5], and Yi-9B-200K[6]. Detailed information about the four models can be found in Table 9.

To showcase the state-of-the-art performance of our method, we compare TailorKV with the following baselines: (1) **StreamingLLM** (Xiao et al., 2023): An eviction strategy that retains only the initial and most recent tokens. (2) **SnapKV** (Li et al., 2024): An eviction strategy that chooses clustered important KV positions. (3) **Quest** (Tang et al., 2024): A selection strategy that determines page criticality through paged key. (4) **PQCache** (Zhang et al., 2024a): A selection strategy that retrieves Top-K tokens through vector quantization.

G.2 Benchmarks

LongBench (Bai et al., 2024). A benchmark is conducted across six categories: summarization, code completion, synthetic tasks, few-shot learning, and single/multi-document question answering. Table 10 presents detailed information on the 13 datasets in LongBench.

InfiniteBench (Zhang et al., 2024b). A benchmark designed to assess the ability of language models to process, understand, and reason with extremely long contexts (200k+ tokens). We test the Llama3 and Yi models with context lengths of 128K and 200K, truncating inputs beyond these limits. Table 11 provides details of the 9 datasets in InfiniteBench.

RULER (Hsieh et al., 2024). A benchmark intended to assess the long-context modeling capabilities of language models, covering question answering, retrieval, aggregation, and multi-hop tracing. This benchmark consists of 13 representative tasks, with sequence lengths ranging from 4K to 128K. For each task, we employed 25 samples. Detailed information is provided in Table 12.

H Detailed Results

H.1 Accuracy on Long Context Tasks

As shown in Figure 13, TailorKV consistently outperforms other methods on the RULER benchmark for Yi-6B-200K (01-ai, 2024a). Additionally, Table 16 shows accuracy results for sequence lengths of 64k and 128k on RULER. Table 14 and Table 15 present experimental results for LongBench and InfiniteBench.

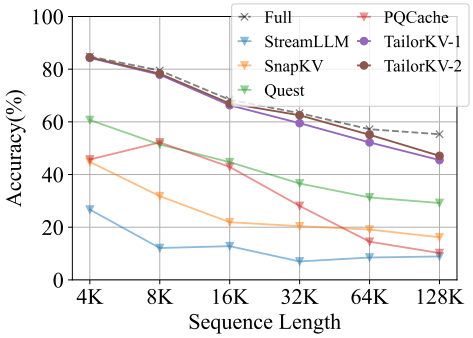


Figure 13: Performance comparison of various methods on RULER with **Yi-6B-200K**.

[3] https://huggingface.co/togethercomputer/Llama-2-7B-32K-Instruct

[4] https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct

[5] https://huggingface.co/01-ai/Yi-6B-200K

[6] https://huggingface.co/01-ai/Yi-9B-200K

Name	Claimed Length	Query Heads	KV Heads	Num Layers	$\mathbb{Q}$
Llama-3.1-8B-Instruct	128k	32	8	32	{0}
Llama-2-7B-32K-Instruct	32k	32	32	32	{0, 1}
Yi-6B-200K	200k	32	4	32	{0, 1}
Yi-9B-200K	200k	32	4	48	{0, 1}

Table 9: Details of models. $\mathbb{Q}$ denotes the quantization-friendly layer.

Label	Task	Capability	Metric	Avg len	#data
Qspr	Qasper	Single-Doc. QA (SD.QA)	F1	3,619	200
MulFi	MultiFieldQA-en	Single-Doc. QA (SD.QA)	F1	4,559	150
HQA	HotpotQA	Multi-Doc. QA (MD.QA)	F1	9,151	200
WMQA	2WikiMultihopQA	Multi-Doc. QA (MD.QA)	F1	4,887	200
GRpt	GovReport	Summarization (Summ)	Rouge-L	8,734	200
MulN	MultiNews	Summarization (Summ)	Rouge-L	2,113	200
TREC	TREC	Few-shot Learning (FS.L)	Accuracy (CLS)	5,177	200
SMSM	SAMSum	Few-shot Learning (FS.L)	Rouge-L	6,258	200
TriQA	TriviaQA	Few-shot Learning (FS.L)	F1	8,209	200
Lcc	LCC	Code Completion (Code)	Edit Sim	1,235	500
Repo	RepoBench-P	Code Completion (Code)	Edit Sim	4,206	500
PsgC	PassageCount	Synthetic (Synth)	Accuracy (EM)	11,141	200
PsgR	PassageRetrieval-en	Synthetic (Synth)	Accuracy (EM)	9,289	200

Table 10: Details of LongBench.

Label	Task	Context	Capability	Metric	Avg len	#Examples
R.PK	Retrieve.PassKey	Fake Book	Retrieve (Retr)	Accuracy	122.4k	590
R.Num	Retrieve.Number	Synthetic	Retrieve (Retr)	Accuracy	122.4k	590
En.Dia	En.Dia	Script	Dialogue (Dia)	Accuracy	103.6k	200
Sum	En.Sum	Fake Book	Novel	Rouge-L-Sum	171.5k	103
En.MC	En.MC	Fake Book	Novel	Accuracy	184.4k	229
En.QA	En.QA	Fake Book	Novel	QA F1 Score	192.6k	351
Zh.QA	Zh.QA	New Book	Novel	QA F1 Score	2068.6k	175
Math.F	Math.Find	Synthetic	Math	Accuracy	87.9k	350
Code.D	Code.Debug	Code Document	Code	Accuracy	114.7k	394

Table 11: Details of InfiniteBench.

Label	Task	Category
N-S1	Single NIAH	Retrieval
N-S2	Single NIAH	Retrieval
N-S3	Single NIAH	Retrieval
N-MK1	Multi-keys NIAH	Retrieval
N-MK2	Multi-keys NIAH	Retrieval
N-MK3	Multi-keys NIAH	Retrieval
N-MV	Multi-values NIAH	Retrieval
N-MQ	Multi-queries NIAH	Retrieval
VT	Variable Tracking	Multi-hop Tracing
CWE	Common Words	Aggregation
FWE	Frequent Words Extraction	Aggregation
QA-1	Question Answering	Question Answering
QA-2	Question Answering	Question Answering

Table 12: Details of RULER.

H.2 Efficiency Results

In Table 13, we present the end-to-end latency for Llama-3.1-8B-Instruct, Llama-2-7B-32K-Instruct, Yi-6B-200K, and Yi-9B-200K. The results indicate that our method achieves efficiency closest to that of the original model.

Method	16k	32k	64k	96k	128k
Llama-3.1-8B-Instruct					
Full Cache	0.024	0.033	0.050	0.062	0.082
OffloadCach	0.124	0.227	0.435	0.743	0.992
PQCache	0.104	0.105	0.108	0.108	0.110
TailorKV	0.045	0.047	0.054	0.054	0.056
Llama-2-7B-32K-Instruct					
Full Cache	0.045	0.077	0.140	OOM	OOM
OffloadCach	0.433	0.838	1.767	3.253	4.468
PQCache	0.108	0.111	0.112	0.115	0.120
TailorKV	0.041	0.062	0.098	0.132	0.170
Yi-6B-200K					
Full Cache	0.019	0.021	0.029	0.036	0.044
OffloadCach	0.066	0.118	0.221	0.325	0.430
PQCache	0.085	0.087	0.090	0.092	0.094
TailorKV	0.041	0.042	0.046	0.049	0.056
Yi-9B-200K					
Full Cache	0.029	0.032	0.043	0.055	0.070
OffloadCach	0.102	0.205	0.417	0.626	0.843
PQCache	0.130	0.138	0.139	0.144	0.150
TailorKV	0.066	0.067	0.072	0.076	0.079

Table 13: Decoding latency(s) on A100 (80G).

I Inference Algorithm Overview

Figure 14 provides the PyTorch-style pseudo-code of the TailorKV algorithm. Lines 2-7 illustrate the initialization process. To reduce memory allocation overhead, tensors are allocated on the specified device during the initialization phase. Simultaneously, a thread pool is used to manage the parallel execution of multiple threads.

During prefilling, for quantization-friendly layers, TailorKV quantizes the KV cache and stores the quantization parameters (Lines 19-21), then calculates attention (Line 68) using FlashAttention (Dao, 2023). For sparsity-friendly layers, TailorKV transfers the KV cache to CPU memory and calculates attention (Line 68) using FlashAttention.

During decoding, for quantization-friendly layers, TailorKV integrates the quantized KV cache into the attention calculation. For sparsity-friendly layers, Lines 24-37 illustrate the process of dynamically retrieving Top-K tokens. Specifically, at layer $l-1$, the algorithm first estimates the critical channels (Equation 10) and prefetches the critical key cache for the next layer (Lines 52-58). Next, it uses the critical current query and the prefetched key cache to approximate the attention scores (Lines 39-45), and then fetches the Top-K tokens (Lines 46-51) at layer l.

J Attention Visualization Across Models

As shown in Figure 15, the attention patterns of different models closely match the results predicted by our usage metric $\mathcal{P}$. Specifically, the quantization-friendly layers of Llama-2-7B-32K-Instruct and Yi-6B-200K are identified as the 0th and 1st layers. In these layers, attention patterns are dense, while the other layers are sparse. Similarly, the quantization-friendly layer of Llama-3.1-8B-Instruct is the 0th layer, where attention pattern is dense, with sparse features in the remaining layers.

K Observations on QKV

Figure 16 illustrates the distribution patterns of queries, keys, and values across different attention heads in Llama-3.1-8B-Instruct. Although outliers appear in both the keys and queries, the locations of the outlier channels are not consistently fixed.

L Information About Use Of Ai Assistants

We use Copilot and ChatGPT to assist us with debugging and writing.

Method	Tokens	SD.QA		MD.QA		Summ		FS.L			Code		Synth		Avg.
		Qspr	MulFi	HQA	WMQA	GRpt	MulN	TREC	SMSM	TriQA	Repo	LCC	PsgC	PsgR	
Llama-3.1-8B	128k	45.5	53.8	54.7	47.1	34.9	27.5	73.0	43.8	91.6	56.7	63.4	7.5	99.5	53.8
StreamLLM	192	21.4	31.3	46.5	38.9	17.9	18.0	40.0	34.4	75.7	45.7	50.7	8.0	99.0	40.6
SnapKV	192	25.7	44.7	52.6	43.7	20.0	20.5	41.0	39.6	89.0	48.7	57.0	8.0	97.0	45.2
Quest	192	35.9	44.2	52.8	41.0	17.7	23.8	63.0	36.0	86.0	43.6	52.3	**8.4**	96.5	46.2
PQCache	192	**45.6**	51.2	53.8	45.3	29.0	25.9	69.5	41.2	91.3	54.1	58.5	8.2	99.0	51.7
TailorKV-1	64(+128)	43.4	53.0	**55.3**	**46.5**	31.3	**27.2**	70.0	42.5	91.6	**54.8**	61.8	7.9	**99.0**	52.6
TailorKV-2	64(+128)	44.8	**53.9**	54.8	46.2	**31.9**	26.8	**70.5**	**43.2**	**92.3**	54.2	**62.1**	7.7	**99.0**	**52.9**
Yi-9B	200k	38.4	34.9	52.7	36.7	31.0	26.7	77.0	14.9	90.0	67.4	71.9	2.5	67.5	47.0
StreamLLM	192	22.3	20.4	36.7	30.6	11.8	10.3	45.5	9.2	77.7	49.6	54.1	3.5	26.0	30.6
SnapKV	192	26.7	23.3	44.2	33.4	11.5	12.3	44.5	13.4	89.1	56.6	62.8	1.6	36.0	35.0
Quest	192	32.4	26.0	44.2	31.5	14.3	16.5	73.0	13.4	86.2	55.6	63.7	3.9	47.5	39.1
PQCache	192	36.9	27.9	47.6	35.5	19.3	19.2	74.0	12.2	**89.6**	62.0	66.8	**4.6**	**51.0**	42.0
TailorKV-1	64(+128)	**37.7**	**38.2**	**52.8**	**35.8**	**29.8**	24.9	**76.0**	**15.2**	89.5	**64.3**	68.4	3.5	45.0	**44.7**
TailorKV-2	64(+128)	37.6	33.6	52.6	34.4	29.4	**25.3**	**76.0**	15.0	89.3	63.5	**68.5**	3.0	44.0	44.0
Yi-6B	200k	25.4	39.5	14.8	15.8	2.8	0.01	72.5	7.7	69.7	58.5	61.2	3.5	15.5	29.7
StreamLLM	192	11.0	29.0	11.1	12.1	3.1	0.1	41.5	5.1	55.5	43.1	46.2	3.0	5.0	20.4
SnapKV	192	14.9	33.5	13.1	13.0	**3.3**	0.01	45.0	6.6	63.8	48.8	53.7	3.0	4.5	23.3
Quest	192	19.9	33.2	11.8	13.2	0.5	0.1	67.0	**9.4**	64.5	50.2	53.7	**3.5**	**13.5**	26.2
PQCache	192	24.1	36.8	13.6	15.3	1.3	0.01	68.5	7.6	68.1	54.1	57.4	2.5	5.5	27.3
TailorKV-1	64(+128)	**24.5**	40.5	**15.6**	15.3	2.7	**0.1**	**72.0**	8.5	**68.3**	**55.2**	56.5	2.5	5.5	**28.3**
TailorKV-2	64(+128)	24.1	**40.8**	15.2	**15.5**	3.0	0.01	**72.0**	8.6	66.7	54.8	**57.9**	2.5	5.5	28.2

Table 14: Results on **LongBench** (Bai et al., 2024) of different methods.

Methods	**Tokens**	**R.PK**	**R.Num**	**En.Dia**	**Sum**	**En.MC**	**En.QA**	**Zh.QA**	**Math.F**	**Code.D**	**Avg.**
Llama-3.1-8B	128K	100.0	99.3	19.0	26.8	65.9	14.8	13.3	34.0	22.8	44.0
StreamLLM	1024	3.3	3.0	7.0	12.7	66.3	5.9	9.7	34.0	22.8	18.3
SnapKV	1024	100.0	**93.2**	9.5	22.4	65.5	10.4	11.3	34.0	22.8	41.0
Quest	1024	100.0	28.9	14.0	12.2	**69.8**	9.2	11.4	34.0	**25.1**	33.8
PQCache	1024	8.6	2.5	15.0	18.9	65.9	12.6	12.6	34.0	23.3	21.5
TailorKV-1	128+(896)	99.8	73.2	18.0	22.8	66.4	13.6	**13.0**	**34.0**	22.8	40.4
TailorKV-1	128+(896)	**100.0**	89.4	**18.5**	**24.1**	66.8	**14.4**	12.9	**34.0**	22.8	**42.6**
Yi-9B	200K	100.0	100.0	2.5	8.2	65.0	10.8	16.7	23.4	26.3	39.2
StreamLLM	1024	2.5	0.5	2.5	6.4	66.8	8.8	15.0	23.7	21.3	16.4
SnapKV	1024	99.8	18.3	3.0	8.6	**67.6**	8.4	14.9	22.5	**26.6**	30.0
Quest	1024	100.0	96.9	4.0	3.4	58.5	12.5	12.7	18.2	18.7	36.1
PQCache	1024	9.6	5.9	2.0	**8.7**	66.3	11.2	14.9	22.2	25.6	18.5
TailorKV-1	128+(896)	**100.0**	98.3	2.5	7.3	64.2	**15.1**	**19.7**	**24.0**	21.3	39.2
TailorKV-1	128+(896)	**100.0**	**99.7**	**4.5**	8.2	65.1	11.2	16.6	**24.0**	24.9	**39.4**
Yi-6B	200K	100.0	98.4	1.0	3.4	53.3	18.2	26.0	6.8	26.9	37.1
StreamLLM	1024	3.3	0.6	0.0	3.9	52.8	10.2	20.2	4.8	25.8	13.5
SnapKV	1024	100.0	11.5	2.5	3.1	53.2	14.1	22.2	4.5	**26.9**	26.4
Quest	1024	100.0	**99.1**	3.0	3.6	52.4	13.4	18.8	5.1	26.6	35.8
PQCache	1024	10.1	3.7	1.5	4.4	51.5	16.4	**26.2**	5.7	**26.9**	16.3
TailorKV-1	128+(896)	**100.0**	97.5	2.5	**4.4**	**53.3**	17.4	26.0	7.7	26.4	37.2
TailorKV-1	128+(896)	**100.0**	97.0	**3.0**	4.0	**53.3**	**18.0**	25.8	**8.0**	26.7	**37.3**

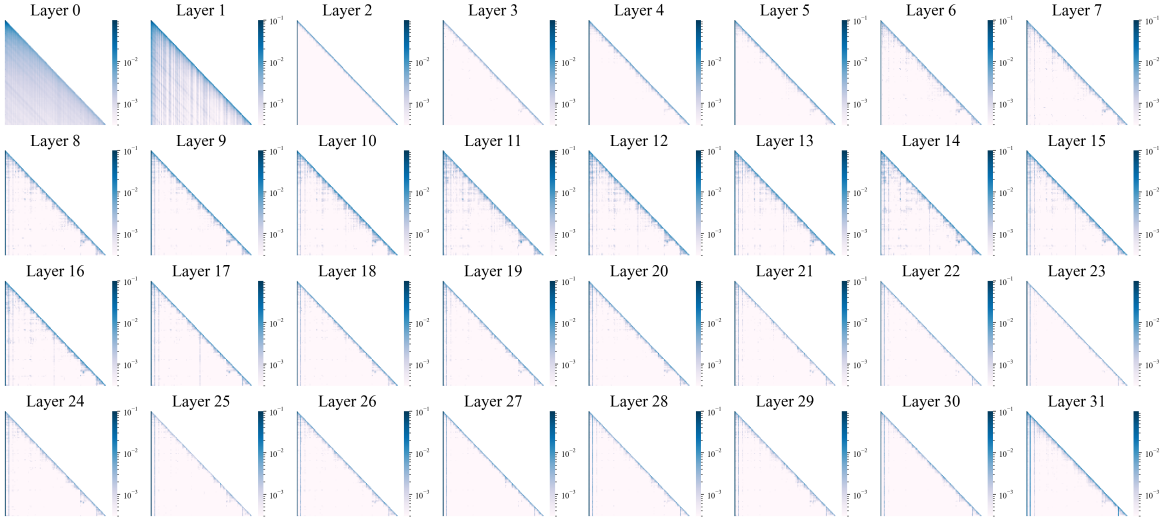
Table 15: Results on **InfiniteBench** (Zhang et al., 2024b) of different methods.

Methods	N-S1	N-S2	N-S3	N-MK1	N-MK2	N-MK3	N-MV	N-MQ	VT	CWE	FWE	QA-1	QA-2	Avg.
Sequence Length = 64k														
Llama-3.1-8B	100.0	100.0	100.0	100.0	100.0	96.0	99.0	100.0	100.0	14.8	92.0	60.0	52.0	85.6
StreamLLM	8.0	4.0	0.0	8.0	0.0	0.0	5.0	3.0	2.4	0.8	72.0	28.0	28.0	12.2
SnapKV	96.0	84.0	0.0	88.0	32.0	0.0	40.0	69.0	74.4	0.8	41.3	56.0	48.0	48.4
Quest	88.0	100.0	60.0	92.0	72.0	0.0	93.0	90.0	80.0	8.4	70.6	52.0	52.0	66.0
PQCache	36.0	60.0	12.0	68.0	48.0	4.0	16.0	37.0	52.8	0.0	**73.3**	56.0	48.0	39.2
TailorKV-1	**100.0**	**100.0**	96.0	**100.0**	**96.0**	28.0	**99.0**	**100.0**	85.6	18.4	57.3	56.0	48.0	75.7
TailorKV-2	**100.0**	**100.0**	**100.0**	**100.0**	**96.0**	**68.0**	97.0	98.0	**88.0**	**19.6**	62.7	**60.0**	**56.0**	**80.4**
Yi-9B	100.0	100.0	100.0	100.0	92.0	48.0	61.0	88.0	12.8	15.6	88.0	32.0	48.0	68.1
StreamLLM	0.0	4.0	0.0	4.0	0.0	0.0	1.0	0.0	0.0	1.2	**74.6**	16.0	28.0	9.9
SnapKV	80.0	28.0	0.0	20.0	4.0	0.0	11.0	11.0	22.4	5.6	48.0	24.0	44.0	22.9
Quest	68.0	92.0	20.0	68.0	40.0	0.0	24.0	42.0	16.0	10.8	62.6	28.0	36.0	39.0
PQCache	32.0	56.0	4.0	36.0	16.0	0.0	7.0	39.0	31.2	6.0	66.6	20.0	36.0	26.9
TailorKV-1	**100.0**	**100.0**	**92.0**	**100.0**	**84.0**	**28.0**	53.0	89.0	6.4	32.0	49.3	**32.0**	**48.0**	62.6
TailorKV-2	**100.0**	**100.0**	**92.0**	**100.0**	**84.0**	**28.0**	**62.0**	**90.0**	**42.4**	**33.6**	48.0	28.0	**48.0**	**65.8**
Yi-6B	100.0	100.0	100.0	96.0	56.0	24.0	39.0	76.0	24.8	0.8	73.3	32.0	24.0	57.3
StreamLLM	0.0	0.0	0.0	8.0	0.0	0.0	3.0	0.0	0.0	0.4	62.6	20.0	16.0	8.5
SnapKV	88.0	4.0	0.0	16.0	0.0	0.0	5.0	7.0	15.2	0.0	**65.3**	28.0	20.0	19.1
Quest	72.0	84.0	0.0	52.0	20.0	0.0	28.0	30.0	20.0	**1.6**	56.0	24.0	20.0	31.3
PQCache	16.0	20.0	0.0	28.0	8.0	0.0	5.0	3.0	10.4	0.0	50.6	24.0	24.0	14.5
TailorKV-1	**100.0**	**100.0**	**100.0**	96.0	12.0	24.0	**41.0**	65.0	28.8	1.2	58.7	28.0	**24.0**	52.2
TailorKV-2	**100.0**	**100.0**	**100.0**	**100.0**	**24.0**	**28.0**	40.0	**67.0**	**42.4**	0.8	57.3	**32.0**	**24.0**	**55.1**
Sequence Length = 128k														
Llama-3.1-8B	100.0	100.0	100.0	100.0	88.0	64.0	96.0	98.0	95.2	1.6	66.6	64.0	36.0	77.6
StreamLLM	0.0	4.0	0.0	0.0	4.0	0.0	5.0	4.0	0.0	0.4	9.3	24.0	20.0	5.4
SnapKV	100.0	84.0	0.0	84.0	24.0	0.0	19.0	38.0	65.6	0.0	28.0	48.0	32.0	40.2
Quest	80.0	68.0	0.0	88.0	48.0	0.0	66.0	71.0	59.2	0.4	52.0	48.0	28.0	46.8
PQCache	0.0	8.0	0.0	4.0	8.0	0.0	2.0	3.0	0.8	0.0	**66.6**	40.0	32.0	12.6
TailorKV-1	92.0	**92.0**	**100.0**	**100.0**	**64.0**	0.0	93.0	**98.0**	67.2	**0.4**	16.0	60.0	**40.0**	63.3
TailorKV-2	**100.0**	**92.0**	**100.0**	**100.0**	**64.0**	**16.0**	**96.0**	97.0	**85.6**	**0.4**	40.0	**64.0**	36.0	**68.5**
Yi-9B	100.0	100.0	100.0	96.0	80.0	28.0	69.0	84.0	10.4	3.6	89.3	36.0	36.0	64.0
StreamLLM	0.0	4.0	4.0	0.0	0.0	0.0	2.0	1.14	0.0	0.0	**86.6**	16.0	24.0	10.6
SnapKV	92.0	12.0	0.0	20.0	4.0	0.0	12.0	4.0	7.2	2.0	53.3	20.0	32.0	19.9
Quest	100.0	84.0	4.0	72.0	24.0	0.0	28.0	28.0	16.8	0.8	69.3	24.0	32.0	37.1
PQCache	8.0	16.0	0.0	24.0	4.0	0.0	2.0	5.0	4.0	0.4	77.3	16.0	28.0	14.2
TailorKV-1	**100.0**	**100.0**	**96.0**	**96.0**	72.0	**20.0**	44.0	79.6	19.2	23.2	44.0	**40.0**	**32.0**	58.9
TailorKV-2	**100.0**	**100.0**	**96.0**	**96.0**	**76.0**	**20.0**	**55.0**	**80.0**	**48.8**	**24.0**	41.3	36.0	**32.0**	**61.9**
Yi-6B	100.0	100.0	100.0	84.0	72.0	4.0	30.0	67.0	4.8	1.2	100.0	32.0	24.0	55.3
StreamLLM	0.0	4.0	4.0	0.0	0.0	0.0	2.0	1.0	0.0	0.8	68.0	20.0	16.0	8.9
SnapKV	76.0	0.0	0.0	16.0	0.0	0.0	1.0	4.0	8.0	0.8	**69.3**	20.0	16.0	16.2
Quest	96.0	72.0	0.0	72.0	20.0	0.0	17.0	23.0	6.4	0.8	49.3	16.0	8.0	29.2
PQCache	0.0	8.0	0.0	12.0	4.0	0.0	1.0	2.0	0.0	0.0	54.6	24.0	**28.0**	10.2
TailorKV-1	**100.0**	**100.0**	**100.0**	**84.0**	16.0	0.0	**25.0**	47.0	3.2	0.8	61.3	**32.0**	22.7	45.5
TailorKV-2	**100.0**	**100.0**	**100.0**	**84.0**	**28.0**	**4.0**	21.0	**48.0**	**14.4**	**1.2**	60.0	**32.0**	20.0	**47.1**

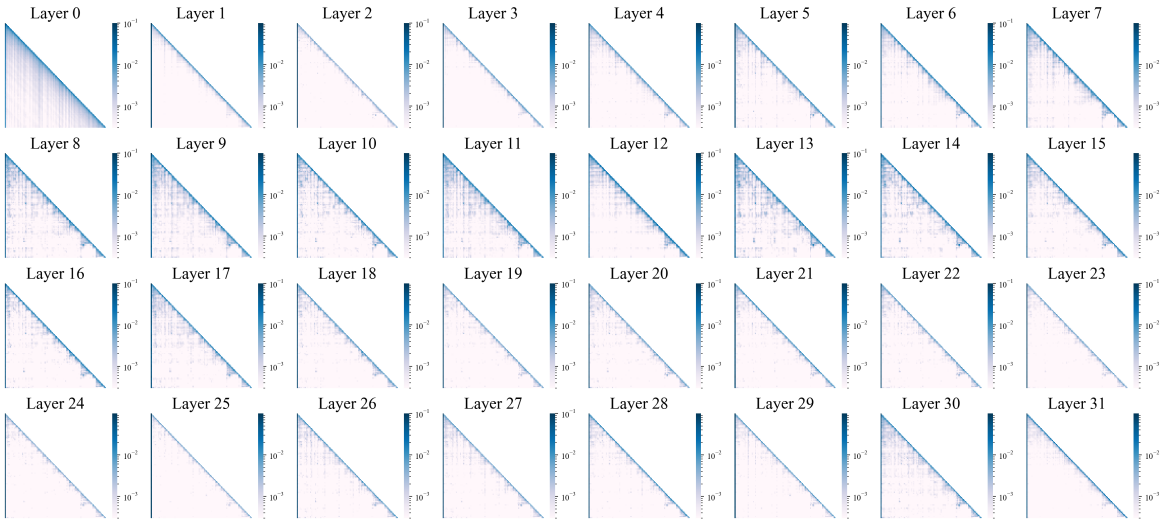
Table 16: Accuracy (%) of different methods and models on **RULER** (Hsieh et al., 2024) evaluated at length of 64k and 128k. The sparsity-friendly layer in TailorKV uses 128+(896) tokens, while other methods use 1024 tokens.

```python
class TailorKV(Cache):
    def __init__(self, layers):
        self.quant_layer = [], self.quant_unit = [] # quantization parameters
        self.cpu_k, self.cpu_v = [], [] # CPU
        self.static_k, self.static_v, self.critical_k, self.quant_unit = [], [], [], [] # GPU
        self.executor = ThreadPoolExecutor(max_workers = max_workers), self.future = [None for _ in range(self.num_layers)]
        ...
    def update(self, key_states, value_states, query_states, next_layer_q, layer_idx):
        if prefill: # prefilling
            if layer_idx not in self.quant_layer: # sparsity-friendly layer
                # Offload the KV Cache to CPU
                self.cpu_k[layer_idx].copy_(key_states), self.cpu_v[layer_idx].copy_(value_cache)
                self.trans_cpu_k[layer_idx].copy_(key_states.transpose(2, 3))
                # Retain the local and initial tokens on the GPU
                self.static_k[layer_idx][:,:,:self.initial,:] = key_states[:,:,:self.initial,:]
                self.static_k[layer_idx][:,:,-self.local:,:] = key_states[:,:,-self.local:,:]
                self.static_v[layer_idx][:,:,:self.initial,:] = value_states[:,:,:self.initial,:]
                self.static_v[layer_idx][:,:,-self.local:,:] = value_states[:,:,-self.local:,:]
            else: # quantization-friendly layer
                ...
                self.quant_unit[layer_idx] = (k_quant, k_scale, k_zp, v_quant, v_scale, v_zp)
            self.absmax_k[layer_idx] = torch.max(torch.abs(key_states), dim = 2, keepdim = True)[0]
        else: # decoding
            if layer_idx not in self.quant_layer: # sparsity-friendly layer
                # Save new token
                self.gen_k[layer_idx][:,:,self._gen_tokens-1:self._gen_tokens,:] = key_states
                self.gen_v[layer_idx][:,:,self._gen_tokens-1:self._gen_tokens,:] = value_states
                # Synchronize
                self.future[layer_idx].result()
                # Retrieval Top-K indices based on approximate attention score
                topk_indices = self.approximate_attn(query_states, key_states, layer_idx)
                flatten_index = self.index_prefix[:,None] + topk_indices.view(self.batch_size * self.num_kv_heads, self.budget)
                # Load a portion of tokens to the GPU
                select_k, select_v = self.load_gpu(layer_idx, flatten_index.view(-1))
                # Prefetch critical key cache for next layer
                self.future[(layer_idx+1) % self.num_layers] = self.executor.submit(self.prefecth_critical_k, next_layer_q,
                    layer_idx)
                return select_k, select_v
            else: ... # quantization-friendly layer
    def approximate_attn(self, query_states, key_states, layer_idx):
        ...
        critical_q = torch.gather(reduce_q(query_states, kv_groups), dim=-1, index=self.channel_indices[layer_idx])
        partial_att = torch.matmul(critical_q, self.critical_k[layer_idx & 1][:,:,:,:self._seen_tokens])
        _, topk_indices = torch.topk(partial_att, k=self.budget, dim=-1)
        topk_indices = topk_indices + self.initial
        return topk_indices
    def load_gpu(self, layer_idx, flatten_index):
        self.cuda_k[:,:,self.initial + self.local:,:] = gather_pinned_tensor_rows(self.cpu_k[layer_idx].view(-1, self.head_dim),
            flatten_index).view(self.batch_size, self.num_kv_heads, self.budget, self.head_dim)
        self.cuda_v[:,:,self.initial + self.local:,:] = gather_pinned_tensor_rows(self.cpu_v[layer_idx].view(-1, self.head_dim),
            flatten_index).view(self.batch_size, self.num_kv_heads, self.budget, self.head_dim)
        self.cuda_k[:,:,:self.initial + self.local,:] = self.static_k[layer_idx]
        self.cuda_v[:,:,:self.initial + self.local,:] = self.static_v[layer_idx]
        return torch.cat([self.cuda_k, self.gen_k[layer_idx][:, :, :self._gen_tokens, :]], dim=2), torch.cat([self.cuda_v, self.
            gen_v[layer_idx][:, :, :self._gen_tokens, :]], dim=2)
    def prefecth_critical_k(self, next_layer_query, layer_idx):
        if (layer_idx+1) not in self.quant_layer:
            result = torch.mul(torch.abs(reduce_q(next_layer_q,self.num_kv_groups)), self.absmax_k[layer_idx+1])
            _, top_channel = torch.topk(result, self.num_channel, dim=-1)
            self.channel_indices[layer_idx+1] = top_channel
            flatten_channel = self.channel_prefix[:,None] + top_channel.view(self.batch_size * self.num_kv_heads, self.num_channel)
            self.critical_k[(layer_idx+1) & 1] = gather_pinned_tensor_rows(self.trans_cpu_k[layer_idx+1].view(-1, self.max_len),
                flatten_channel.view(-1)).view(self.batch_size, self.num_kv_heads, self.num_channel, self.max_len)
def reduce_q(hidden_states, n_rep):
    batch, num_attention_heads, slen, head_dim = hidden_states.shape
    if n_rep == 1: return hidden_states # MHA
    hidden_states = hidden_states.view(batch, num_attention_heads // n_rep, n_rep, slen, head_dim) # GQA
    return hidden_states.mean(dim=2)
...
def attention_forward(self, hidden_states, ..., past_key_value = Optional[Cache] = None):
    if prefill: # prefilling
        past_key_value.update(key_states, value_states, query_states, self.layer_idx)
        attn_output = flash_attention(key_states, value_states, query_states)
    else: # decoding
        next_layer_q_proj = past_key_value.layers[(self.layer_idx+1) % self.num_layers].self_attn.q_proj
        next_layer_q = next_layer_q_proj(hidden_states).view(bsz, 1, self.num_heads, self.head_dim).transpose(1, 2)
        next_layer_q, _ = apply_rotary_pos_emb(next_layer_q, key_states, cos, sin, position_ids)
        if self.layer_idx in past_key_value.quant_layer: ... # quantization-friendly layer
        else: # sparsity-friendly layer
            select_k, select_v = past_key_value.update(key_states, value_states, query_states, next_layer_q, self.layer_idx)
            attn_output = flash_attention(select_k, select_v, query_states)
```

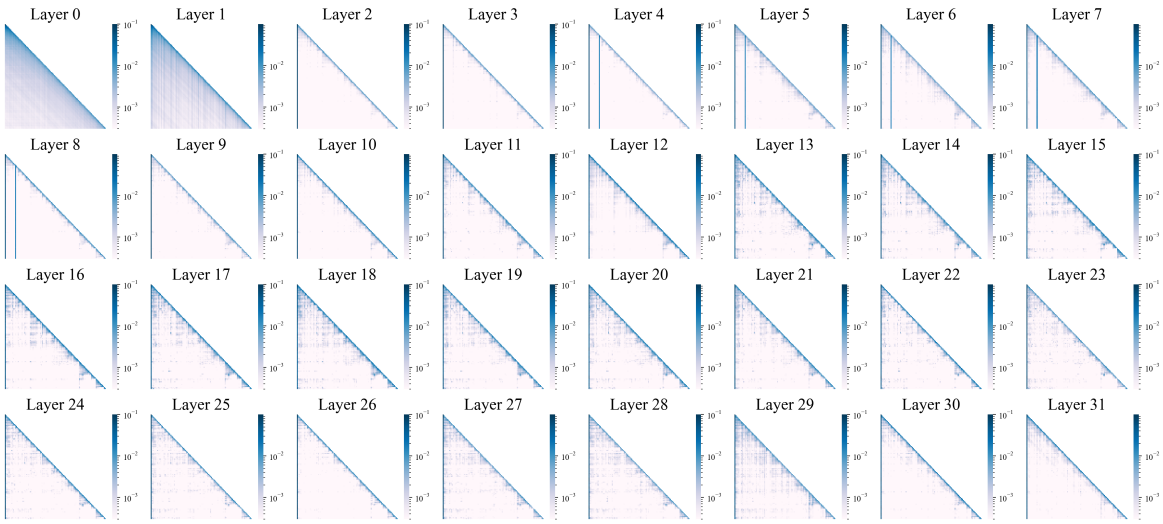
Figure 14: Implementation of TailorKV in pseudo PyTorch style.



(a) Visualization of attention weights on Llama-2-7B-32K-Instruct.



(b) Visualization of attention weights on Llama-3.1-8B-Instruct.



(c) Visualization of attention weights on Yi-6B-200K.

Figure 15: Visualization of attention weights across the 2WikiMQA dataset.

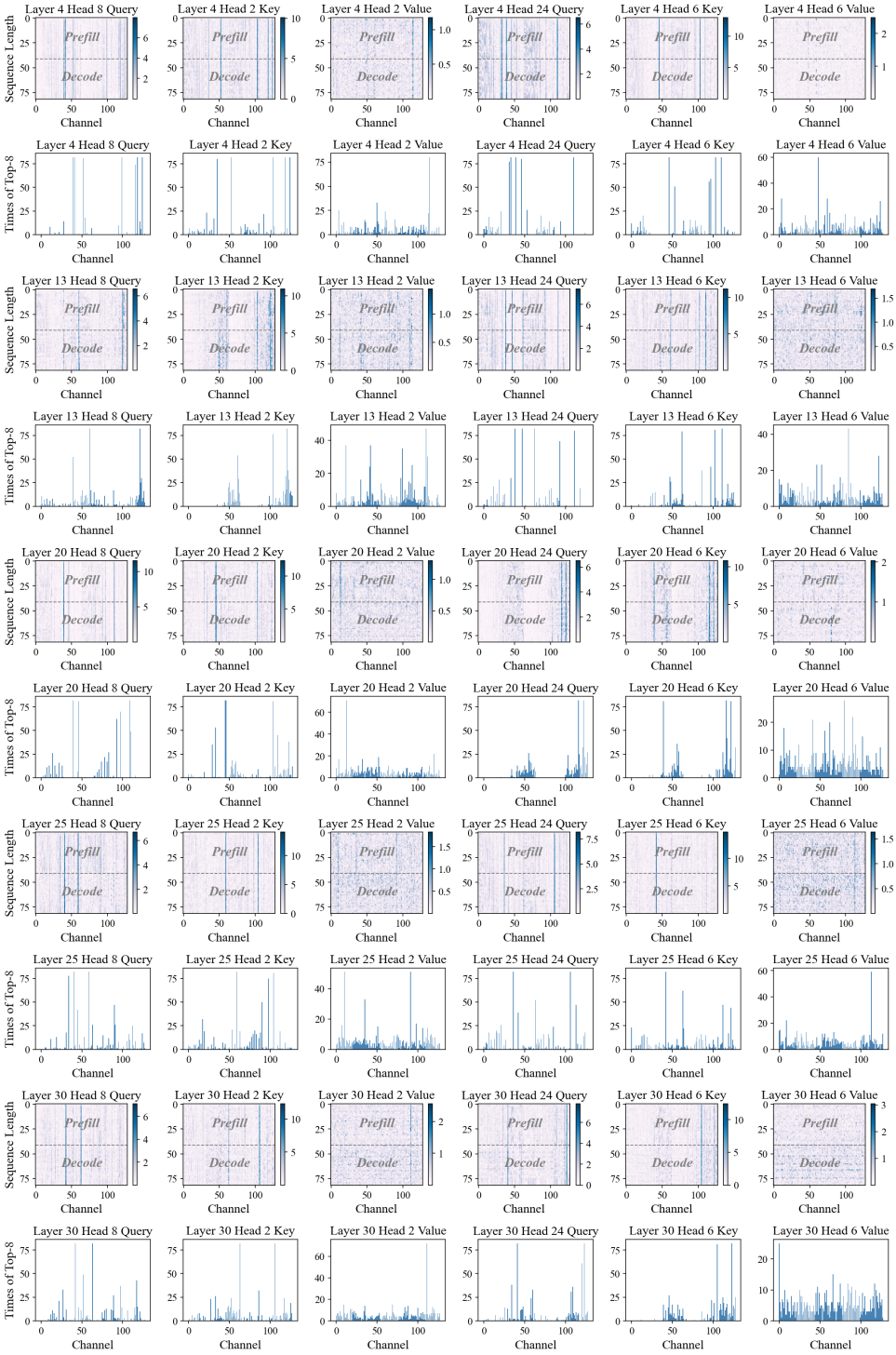


Figure 16: Magnitude of query, key and value for Llama-3.1-8B-Instruct.