

HachiWare: One-Shot Selective SVD in a Dual-Branch Framework for KV Cache Compression

Hsiao-Ting Shao^{1*} Xiaolin Lin^{2*} Yu-Chen Lu¹
Grace Li Zhang² Kai-Chiang Wu¹ Ning-Chi Huang³

¹National Yang Ming Chiao Tung University, Hsinchu, Taiwan

²Technical University of Darmstadt, Darmstadt, Germany

³National Cheng Kung University, Tainan, Taiwan

¹{ting0602.cs13, yuchen.cs11}@nycu.edu.tw, kcw@cs.nycu.edu.tw

²{xiaolin.lin, grace.zhang}@tu-darmstadt.de

³nchuang@gs.ncku.edu.tw

Abstract

Long-context inference is increasingly limited by the memory and access cost of the key-value (KV) cache. Existing KV cache compression methods face a fundamental trade-off between efficiency and information preservation. Eviction-based methods are efficient but may discard critical tokens, whereas low-rank-based methods preserve all tokens at the cost of substantial computational overhead. We propose HachiWare, a dual-branch framework that applies one-shot selective SVD to semantic-critical cache states while handling less critical states with lightweight eviction. It stores these states in a Semantic-Critical Branch (SCB) using low-rank factors, while assigning the remaining states to an Eviction Branch (EB) for cheaper eviction. During decoding, bounded active-cache refresh selectively reconstructs only active SCB entries, avoiding reconstruction of the full SCB cache. On the RULER benchmark with Llama-3.1-8B-Instruct, HachiWare incurs only a 3% accuracy drop compared with the full KV cache baseline at a KV cache compression ratio of up to 16 $\times$, while improving accuracy by 5% over existing compression methods. For decoding efficiency, HachiWare achieves up to 9.44 $\times$ and 16.50 $\times$ throughput speedups over Full KV at 64K and 128K context lengths. These results highlight selective allocation of reconstructability as an effective way to improve the accuracy–efficiency trade-off of KV cache compression.

1 Introduction

The key-value (KV) cache enables efficient autoregressive decoding by reusing past attention states. However, its memory footprint grows with context length, batch size, model depth, and the number of KV heads, making KV cache storage and access a major bottleneck in long-context inference (Kwon et al., 2023; Liu et al., 2024a; Yuan et al., 2024).

*These authors contributed equally.

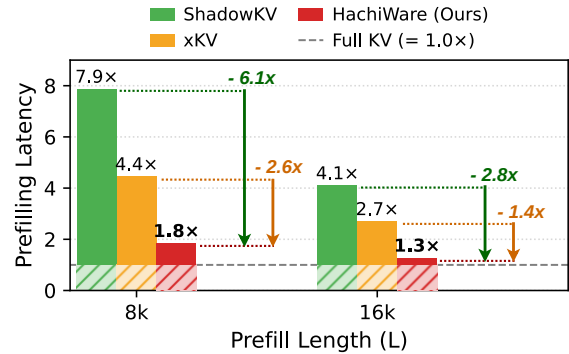


Figure 1: Prefilling latency normalized to Full KV on Llama-3.1-8B-Instruct using an NVIDIA A6000 GPU. At 8K context length, xKV and ShadowKV reach 4.4 $\times$ and 7.9 $\times$ Full KV latency, while HachiWare remains at 1.8 $\times$. The 16K result shows the same trend, with HachiWare staying closer to Full KV by applying SVD selectively rather than to the full KV cache. Full 8K–64K latency results are provided in Appendix C.10.

KV cache compression methods reduce different aspects of this bottleneck. Quantization reduces cache precision (Liu et al., 2024b; Hooper et al., 2024; Kang et al., 2024; Yang et al., 2024b), but its compression is bounded by the accuracy and outlier-handling constraints of low-bit storage. Query-aware selection methods such as Quest (Tang et al., 2024) reduce decoding-time access by selecting an active subset, but leave the stored cache representation largely intact. These directions are complementary to methods that reduce stored KV states. Among existing approaches, KV cache eviction (Zhang et al., 2023; Liu et al., 2023; Xiao et al., 2024; Li et al., 2024; Cai et al., 2025; Qin et al., 2025; Wan et al., 2025; Feng et al., 2025) is the most direct: it retains only selected KV states and discards the rest, reducing both memory usage and attention computation. However, eviction is irreversible: once a KV state is removed, later decoding steps cannot recover it. This issue becomes more pronounced when KV caches are

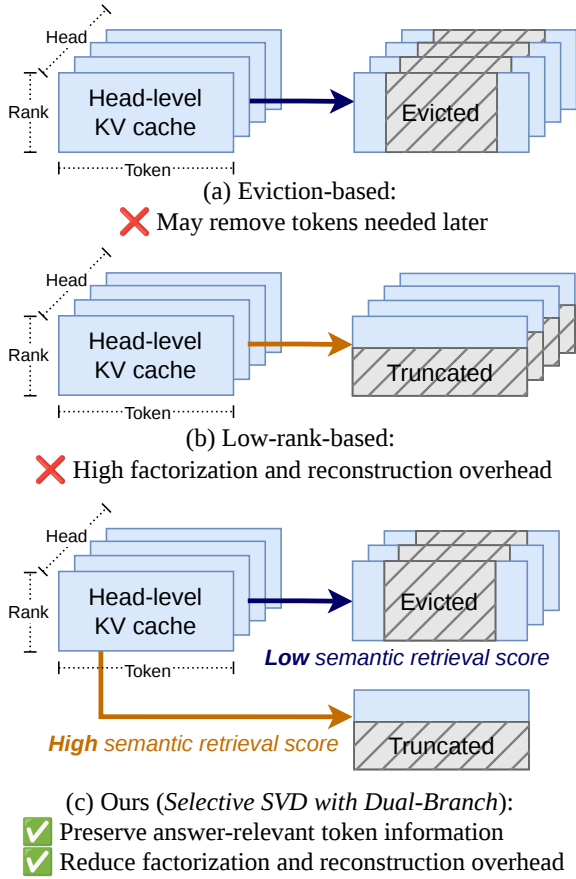


Figure 2: Conceptual comparison of KV cache reduction strategies. Eviction is efficient but irreversible, while full-cache SVD is reconstructable but costly. HachiWare uses dual-branch selective SVD to allocate reconstructable low-rank storage only to semantic-critical heads and applies cheaper eviction to the rest.

reused across turns or requests. SCBench (Li et al., 2025) evaluates such shared-context settings and shows that discard-based methods may lose effectiveness in multi-turn scenarios, where later turns can depend on information removed earlier.

Low-rank KV cache compression based on Singular Value Decomposition (SVD) offers the complementary property missing from eviction: reconstructability. Palu (Chang et al., 2025) uses low-rank projection to cache compressed intermediate states and reconstruct full keys and values on the fly, ShadowKV (Sun et al., 2025) maintains a low-rank key cache and reconstructs states for attention, and xKV (Chang et al., 2026) applies cross-layer SVD to grouped KV caches. Unlike eviction, low-rank compression preserves a compact approximation from which KV states can later be reconstructed, reducing the risk of permanently losing useful cache information.

However, reconstructable low-rank storage comes with non-negligible factorization and reconstruction overhead. SVD-based methods must decompose large cache matrices during the prefilling stage and reconstruct KV states during decoding. This observation motivates selective SVD, which applies low-rank compression only to selected portions of the KV cache. Figure 1 shows that selective SVD keeps SVD-related prefill overhead much lower than broader low-rank baselines.

The next question is how to decide which cache states should receive reconstructable storage. Prior head-level KV cache methods show that cache policies and budgets need not be uniform across heads (Xiao et al., 2025; Tang et al., 2025; Fu et al., 2025). Retrieval-head studies further identify attention heads that play a disproportionate role in long-context factual retrieval (Wu et al., 2025). Building on this direction, CompressKV (Lin et al., 2025b) introduces semantic retrieval scores to identify attention heads that are more effective at attending to semantically important information in the context. The selected heads are further used to guide KV cache eviction.

Figure 2 summarizes stored-cache reduction strategies. Eviction-based methods, as illustrated in Figure 2(a), are computationally efficient but may discard important information. Low-rank-based methods, as shown in Figure 2(b), can better preserve key information but incur substantial additional latency. In contrast, HachiWare performs one-shot selective SVD in a dual-branch design, as shown in Figure 2(c). HachiWare uses offline semantic retrieval scores to identify heads with stronger answer-relevant retrieval behavior. The KV cache of these high-score heads is assigned to the *Semantic-Critical Branch* (SCB) and jointly compressed using SVD-based low-rank compression to preserve critical token information. The remaining KV cache is assigned to the *Eviction Branch* (EB), where a lightweight eviction strategy is applied. By applying SVD only once to the SCB cache, HachiWare reduces the scope of prefilling-stage factorization and decoding-time reconstruction while preserving reconstructability for semantic-critical cache states.

During decoding, bounded active-cache refresh further reduces reconstruction overhead by reconstructing only the active SCB entries needed for attention, avoiding reconstruction of the full SCB cache.

We evaluate HachiWare on LLaMA (Grattafiori

et al., 2024) and Mistral (Jiang et al., 2023; Mistral AI, 2024) models with the RULER (Hsieh et al., 2024) benchmark. At a $16\times$ KV cache compression ratio, HachiWare outperforms prior methods by 5% in accuracy while incurring only a 3% accuracy drop compared with the full KV cache baseline. For efficiency, HachiWare achieves up to a $16.5\times$ decoding-throughput speedup over full KV cache inference. These results show that HachiWare effectively reduces memory usage and accelerates decoding while maintaining strong model performance.

Our key contributions are:

- We propose HachiWare, a one-shot selective SVD framework that stores the SCB cache with reconstructable low-rank factors while using cheaper eviction for the EB cache.
- We introduce bounded active-cache refresh for efficient decoding over heterogeneous SCB/EB storage, reducing runtime reconstruction and selection overhead.
- We validate HachiWare across retrieval, reasoning, and efficiency evaluations, showing strong high-compression accuracy with reduced prefill and decoding overhead.

We discuss additional related work on KV cache compression and head-aware cache policies in Appendix D.

2 Preliminaries

HachiWare uses the semantic retrieval score introduced by CompressKV (Lin et al., 2025b) as an offline head-level importance signal. Section 3.1 aggregates these scores into KV-head semantic retrieval scores: higher-scoring KV heads enter the SCB for reconstructable storage, while the remaining heads enter the EB and are initialized during prefilling using CompressKV’s token-eviction rule. Details are provided in Appendix A.1.

2.1 Semantic Retrieval Score

An answer span is the contiguous segment of text in the context that contains the ground-truth answer. CompressKV assigns each attention head a semantic retrieval score by accumulating its attention mass over the answer span when the model generates answer tokens. Unlike retrieval-head criteria based on exact top-attention hits (Wu et al., 2025), this score credits heads that attend to the full

answer-relevant span even when their maximum attention does not fall on a single answer token.

Let $\mathcal{A}_{\text{span}}$ denote the reference answer span used during offline identification. At decoding step t , y_t is the generated token, and $a_{t,j}^{(\ell,h)}$ is the attention weight assigned by head h in layer ℓ to token position j . Following CompressKV, the semantic retrieval score is

$$s_{\ell,h}^Q = \sum_{t=1}^D \mathbb{I}[y_t \in \mathcal{A}_{\text{span}}] \sum_{j \in \mathcal{A}_{\text{span}}} a_{t,j}^{(\ell,h)}, \quad (1)$$

where D is the number of decoding steps used for offline identification and $\mathbb{I}[\cdot]$ is the indicator function. A larger $s_{\ell,h}^Q$ indicates that head h more consistently allocates attention mass to answer-relevant span positions when generating answer tokens.

The reference answer span is used only for offline identification. Once computed, the semantic retrieval scores are fixed and reused during inference without answer spans, task-specific labels, or per-input scoring; profiling details are provided in Appendix A.1.

3 Method

To reduce the overhead of reconstructable KV cache compression, HachiWare applies one-shot selective SVD to semantic-critical cache states and cheaper eviction to the rest.

This framework consists of three components, summarized in Figure 3: (1) branch assignment, which forms the *Semantic-Critical Branch* (SCB) cache and the *Eviction Branch* (EB) cache; (2) SCB cache factorization, which applies SVD once to the SCB cache during the prefilling stage; and (3) bounded active-cache refresh, which uses anchor heads to construct one shared ordered list across layers and selectively reconstructs only active SCB entries during decoding.

3.1 Semantic-Critical Branch Construction

Branch assignment determines which cache states receive reconstructable low-rank storage, as shown in Figure 3. Because KV cache storage is organized by KV heads, HachiWare assigns cache states at the KV-head level: states associated with selected KV heads enter the SCB cache, while the remaining states enter the EB cache.

We use the offline semantic retrieval scores $s_{\ell,h}^Q$ (Eq. 1) as fixed measures of answer-relevant retrieval behavior. Since these scores are produced by query-head attention distributions, we aggregate

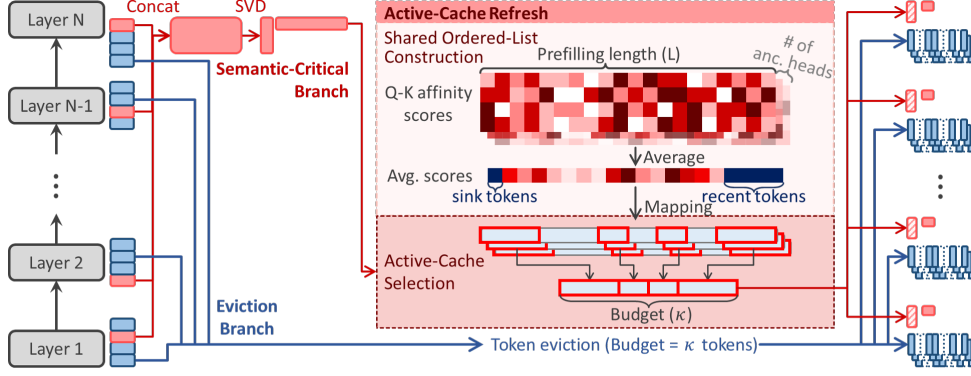


Figure 3: Overview of HachiWare. (1) Branch assignment uses KV-head semantic retrieval scores to form the SCB cache and EB cache. (2) During prefilling, the SCB cache is concatenated and factorized once by SVD, while the EB cache is initialized by the token-retention rule adapted from CompressKV. (3) Active-cache refresh constructs the bounded active cache for decoding: anchor heads build a shared ordered list from Q–K affinity scores, and κ positions are selected with sink/recent tokens always included. SCB entries are selectively reconstructed from low-rank factors, while EB entries are gathered from retained exact states. The initial refresh is applied at the end of the prefilling stage and then repeated every I decoding steps.

them to KV-head semantic retrieval scores for storage decisions. For KV head m in layer ℓ , let $\mathcal{Q}_{\ell,m}$ denote the query heads sharing this KV head. Under standard multi-head attention (MHA) (Vaswani et al., 2017), $|\mathcal{Q}_{\ell,m}| = 1$; under grouped-query attention (GQA) (Ainslie et al., 2023), multiple query heads may share one KV head. We define

$$s_{\ell,m}^{\text{KV}} = \frac{1}{|\mathcal{Q}_{\ell,m}|} \sum_{h \in \mathcal{Q}_{\ell,m}} s_{\ell,h}^{\text{Q}}. \quad (2)$$

This reduces to $s_{\ell,h}^{\text{Q}}$ under standard MHA and averages over shared query heads under GQA.

Since these scores are precomputed, branch assignment does not require a reference answer span or task-specific label at inference time. Given an SCB routing budget, KV heads with the highest KV-head semantic retrieval scores form the SCB cache, whose states receive one-shot selective SVD. The remaining KV heads form the EB cache, which is initialized using the token-retention rule adapted from CompressKV.

3.2 One-Shot Selective SVD

3.2.1 SCB Cache Factorization

This step corresponds to the prefilling-stage SCB factorization in Figure 3. After branch assignment, HachiWare applies SVD only to the SCB cache, while the EB cache is not factorized. The factorization is *one-shot*: it is performed once during the prefilling stage, and the resulting SCB factors are reused throughout decoding rather than recomputed at each step. The low-rank representation

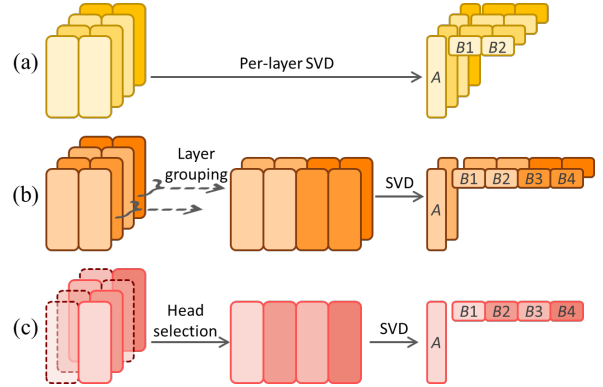


Figure 4: Granularity comparison of SVD in different frameworks. (a) Per-layer SVD factorizes each layer separately. (b) xKV applies cross-layer group-wise SVD (Chang et al., 2026). (c) HachiWare applies one-shot selective SVD only to the concatenated SCB cache, yielding a shared factor A and head-wise blocks B_i , while the EB cache is not factorized.

stores the SCB cache using a shared token basis A and head-wise reconstruction blocks B_i , enabling selective reconstruction during decoding.

Figure 4 compares this factorization granularity with prior SVD-based designs. Layer-wise SVD factorizes each layer separately, while cross-layer SVD factorizes fixed layer groups (Chang et al., 2026). In contrast, HachiWare applies SVD only to the SCB cache, avoiding full-cache factorization.

We next define the factorization shown in Figure 4(c) for a generic SCB cache matrix $\mathbf{X}$. Let M be the number of KV heads assigned to the SCB. For the i -th such KV head, let $\mathbf{X}_i \in \mathbb{R}^{L \times d_h}$ denote its prefilling-stage cache matrix, where L is

the prefilling sequence length and d_h is the head dimension. The SCB cache matrices are concatenated along the head dimension, yielding a width of Md_h , while keeping token positions aligned:

$$\mathbf{X} = [\mathbf{X}_1 \ \mathbf{X}_2 \ \cdots \ \mathbf{X}_M] \in \mathbb{R}^{L \times Md_h}. \quad (3)$$

A rank- r SVD approximation is stored as

$$\mathbf{X} \approx A[B_1 \ B_2 \ \cdots \ B_M], \quad (4)$$

where $A \in \mathbb{R}^{L \times r}$ is a shared token basis and each $B_i \in \mathbb{R}^{r \times d_h}$ is the reconstruction block for the i -th KV head assigned to the SCB. This factorization does not assume that KV heads from different layers share the same representation space: each assigned KV head keeps its own block B_i , while A captures token-side correlations in the concatenated SCB cache.

This storage form supports selective reconstruction during decoding: selected token rows are gathered from A first and multiplied only with the required B_i blocks, avoiding reconstruction of the full SCB cache. Section 3.3 describes how these selected entries are chosen during decoding.

3.2.2 Low-Rank Structure of the SCB Cache

When a few singular directions capture much of the matrix energy, low-rank approximation can effectively compress a matrix. Therefore, rather than factorizing caches at individual layers or nearby layer groups as in common SVD-based KV compression designs, HachiWare constructs the SCB cache by globally selecting KV heads according to their semantic retrieval scores, decoupled from layer proximity.

We analyze the rank sensitivity of different pre-RoPE K-cache grouping approaches with Llama-3.1-8B-Instruct on the RULER multi-query Needle-in-a-Haystack task (Hsieh et al., 2024). Figure 5 compares the SCB with Random, xKV-style, and ShadowKV-style grouping alternatives under matched compression rates. The SCB grouping retains more cumulative explained variance than the alternatives, indicating stronger singular-value energy concentration and better suitability for low-rank approximation.

These results indicate that SCB routing yields a cache matrix with stronger shared low-rank structure, supporting SVD-based compression. Because the selected heads are identified by shared answer-relevant attention patterns, their cache representations exhibit consistent token-side retrieval behavior, yielding a structurally coherent SCB cache

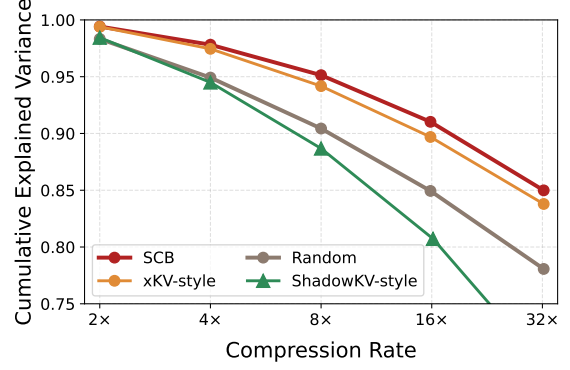


Figure 5: Rank sensitivity of different pre-RoPE K-cache groupings. Cumulative explained variance measures retained singular-value energy under matched compression rates. SCB uses the top-32 KV heads ranked by KV-head semantic retrieval scores; Random, xKV-style, and ShadowKV-style provide global-random, nearby-layer, and per-layer grouping baselines. The SCB grouping retains more variance than the alternatives.

matrix that is well-suited for low-rank SVD approximation. Appendix B.2 provides the CKA analysis and rank-sensitivity protocol.

3.3 Active-Cache Refresh for Decoding

The *Active-Cache Refresh* module in Figure 3 summarizes the decoding-time procedure. Although selective SVD already reduces the reconstruction scope, reconstructing the full SCB cache at every decoding step would still introduce substantial overhead. HachiWare therefore maintains a bounded active cache: only active SCB entries are reconstructed from low-rank factors, while active EB entries are gathered from retained exact states.

The active cache is refreshed every I decoding steps. At each refresh point, HachiWare updates the active set under budget κ ; between refresh points, newly generated KV states are kept active until the next refresh. Thus, token selection, SCB reconstruction, and EB gathering occur only at refresh points and are amortized across the I -step interval.

To keep refresh lightweight, HachiWare constructs one shared ordered list of token positions rather than ranking tokens separately for every layer. A small set of *anchor heads*, selected once from the SCB KV heads after branch assignment, produces this ordering from key-cache affinity scores. The ordered list is reused across layers for active-cache selection, with sink and recent tokens always included following StreamingLLM (Xiao et al., 2024). Appendix A.4 provides pseudocode

and attention-recovery analysis.

4 Experiments

4.1 Experimental Setup

Models and inference. The main evaluation uses Llama-3.1-8B-Instruct (Grattafiori et al., 2024) and Mistral-7B-Instruct-v0.3 (Jiang et al., 2023; Mistral AI, 2024). Llama-3.1-8B-Instruct is evaluated at 64K context length, and Mistral-7B-Instruct-v0.3 is evaluated at 32K context length. All experiments use bf16 precision and greedy decoding. Appendix C.7 provides supplementary results on Qwen2.5-7B-Instruct-1M (Yang et al., 2025) and Qwen2.5-14B-Instruct (Yang et al., 2024a), where the comparisons are restricted to stored-cache compression baselines.

Benchmarks. Our main long-context benchmark is RULER (Hsieh et al., 2024), which covers retrieval, multi-hop tracking, frequent word extraction, and question-answering tasks. The primary metric is the simple average over evaluated RULER subtasks. To assess behavior beyond single-turn long-context retrieval, we also include 5-shot GSM8K (Cobbe et al., 2021) for reasoning robustness and multi-turn Needle-in-a-Haystack (NIAH) (Kamradt, 2023) for retrieval robustness when future queries change.

Baselines. We compare HachiWare with representative KV cache reduction baselines: xKV (Chang et al., 2026), the closest low-rank KV cache compression baseline; PyramidKV (Cai et al., 2025) and SnapKV (Li et al., 2024), token-eviction baselines; KIVI (Liu et al., 2024b), a quantization baseline; and Quest (Tang et al., 2024) and ShadowKV (Sun et al., 2025), whose memory savings involve runtime selection or offloading rather than only stored-cache¹ compression.

Implementation details. We use MInference implementations (Jiang et al., 2024) for PyramidKV, SnapKV, KIVI, and Quest, with an observation window of 8 for PyramidKV and SnapKV to match HachiWare’s eviction window. For HachiWare, we route 32 KV heads to the SCB and include 4 sink tokens and 16 recent tokens within the active-token

¹Stored-cache compression measures the KV representation actually kept after prefilling. This differs from the GPU-side reduction of Quest and ShadowKV, whose savings come from runtime selection or offloading rather than an equivalently compressed stored KV representation.

budget κ . Additional hyperparameters are provided in Appendix A.3.

Compression metrics. **Comp.** denotes the compression ratio of the stored KV cache representation after prefilling and serves as the primary accuracy-table metric. Gray parentheses report effective GPU-side cache reduction for Quest and ShadowKV. Appendix A.2 details memory accounting, including HachiWare’s auxiliary-buffer-inclusive compression and measured peak GPU allocation.

4.2 Long-Context RULER Results

Tables 1 and 2 report the main RULER results on Llama-3.1-8B-Instruct and Mistral-7B-Instruct-v0.3, with full Mistral per-task results deferred to Appendix C.2. The main comparison emphasizes the speed-oriented $I = 32$ setting, which is also used in the efficiency study. Across both models, HachiWare preserves strong long-context accuracy under aggressive stored-cache compression.

On Llama-3.1-8B-Instruct, HachiWare gives the strongest accuracy–compression trade-off in both compression regimes. With $I = 32$, it achieves 90.12 average RULER accuracy at $8.55\times$ stored-cache compression, close to the FullKV average accuracy of 92.30 and above all compressed baselines. At $16.04\times$ stored-cache compression, the same speed-oriented setting preserves 88.99 average RULER accuracy, substantially higher than the $16\times$ eviction baselines. The $I = 2$ results are similar, indicating that the Llama gains do not rely on a more frequent refresh schedule.

On Mistral-7B-Instruct-v0.3, the $I = 32$ setting remains effective despite less frequent active-cache refresh. At $8.29\times$ stored-cache compression, it achieves 78.34 average RULER accuracy, within 0.5 points of xKV-4 and Quest and above the token-eviction, quantization, and ShadowKV baselines. At $16.58\times$ stored-cache compression, it obtains 71.01 average RULER accuracy, substantially outperforming the $16\times$ PyramidKV and SnapKV results. The $I = 2$ rows provide an accuracy-oriented reference, reaching 82.66 average RULER accuracy at $8.29\times$ compression and 81.41 at $16.58\times$ compression; this refresh-interval trade-off is analyzed further in Appendix C.9.

As a larger-model check, Appendix C.7 also reports RULER results on Qwen2.5-14B-Instruct at 64K context length. HachiWare achieves 80.70 average RULER accuracy at $7.12\times$ stored-cache compression, outperforming SnapKV and xKV-4

Method	Comp.	N-S1	N-S2	N-S3	N-MK1	N-MK2	N-MQ	N-MV	QA-1	QA-2	VT	FWE	Avg.
FullKV	1.00	100.00	100.00	100.00	100.00	91.66	100.00	98.69	82.29	60.41	97.91	84.37	92.30
KIVI-2	7.10	100.00	97.91	73.95	94.79	65.62	93.75	86.97	82.29	54.16	81.66	84.02	83.19
xKV-4	8.03	100.00	97.91	62.50	98.95	91.66	94.27	92.96	77.08	57.29	87.29	77.43	85.21
PyramidKV	8.00	100.00	100.00	78.12	100.00	90.62	99.21	98.69	83.33	57.29	95.41	63.19	87.81
SnapKV	8.00	100.00	98.95	82.29	100.00	86.45	99.73	97.91	83.33	58.33	95.00	65.62	87.96
Quest	1.00 (8.00)	100.00	100.00	85.41	100.00	89.58	100.00	97.39	83.33	58.33	87.70	84.02	89.61
ShadowKV	1.83 (9.08)	100.00	100.00	100.00	100.00	91.66	97.91	96.61	82.29	60.41	65.00	74.30	88.02
HachiWare _{I=2}	8.55	100.00	100.00	98.95	100.00	92.70	99.73	97.39	81.25	61.45	91.87	62.50	89.62
HachiWare _{I=32}	8.55	100.00	100.00	100.00	100.00	91.66	99.73	96.61	83.33	58.33	92.91	68.75	90.12
PyramidKV	16.00	100.00	98.95	38.54	100.00	77.08	99.21	97.91	83.33	58.33	94.58	60.76	82.61
SnapKV	16.00	100.00	98.95	53.12	100.00	76.04	98.95	98.17	84.37	58.33	94.37	60.76	83.91
HachiWare _{I=2}	16.04	100.00	100.00	98.95	100.00	88.54	99.47	96.35	82.29	58.33	90.83	67.01	89.25
HachiWare _{I=32}	16.04	100.00	100.00	93.75	100.00	88.54	99.47	97.91	81.25	58.33	87.08	72.56	88.99
HachiWare _{I=32}	34.20 [†]	100.00	97.91	83.33	95.83	86.45	97.65	95.57	74.99	67.70	82.29	58.33	85.46

Notes. N-S, N-MK, N-MQ, and N-MV denote Needle single-key, multi-key, multi-query, and multi-value tasks. Gray parentheses denote effective GPU-side cache reduction for Quest and ShadowKV. [†] denotes a stress-test setting beyond the main 8×/16× regime.

Table 1: Main RULER results on Llama-3.1-8B-Instruct at 64K context length. **Comp.** denotes representation-level compression of the stored KV cache; gray parentheses denote effective GPU-side cache reduction for selection/offloading baselines.

Method	8× regime		16× regime		Mean
	Comp.	Avg.	Comp.	Avg.	
FullKV	1.00	86.36	–	–	
KIVI-2	7.10	58.89	–	–	
xKV-4	8.27	78.82	–	–	
PyramidKV	8.00	69.58	16.00	58.21	
SnapKV	8.00	69.49	16.00	58.50	
Quest	1.00 (8.00)	78.63	–	–	
ShadowKV	1.82 (9.02)	76.25	–	–	
HachiWare _{I=2}	8.29	82.66	16.58	81.41	
HachiWare _{I=32}	8.29	78.34	16.58	71.01	

Notes. Gray parentheses denote effective GPU-side cache reduction for Quest and ShadowKV.

Table 2: Average RULER results on Mistral-7B-Instruct-v0.3 at 32K context length. Full per-task results are provided in Appendix C.2.

at comparable compression.

4.3 Cross-Task Robustness

Table 3 evaluates robustness beyond the single-turn long-context setting. It exposes complementary weaknesses of the two baseline compression styles: xKV preserves multi-turn retrieval well, reaching 98.43 average NIAH accuracy, but drops sharply to 29.57 GSM8K accuracy; PyramidKV and SnapKV maintain stronger GSM8K accuracy at 75.28, but degrade in multi-turn retrieval, reaching 56.45 and 14.37 average NIAH accuracy.

HachiWare gives the best balance across the two settings. It maintains 72.78 GSM8K accuracy at 3.97× stored-cache compression and reaches 98.12 average multi-turn NIAH accuracy at 8.28× stored-cache compression, yielding the best compressed-

Method	GSM8K		Multi-turn NIAH		Mean
	Comp.	Acc.	Comp.	Avg.	
FullKV	1.00	78.77	1.00	100.00	89.39
xKV-4	4.01	29.57	8.03	98.43	64.00
PyramidKV	4.01	75.28	8.00	56.45	65.87
SnapKV	4.01	75.28	8.00	14.37	44.83
HachiWare	3.97	72.78	8.28	98.12	85.45

Notes. GSM8K reports 5-shot accuracy; Multi-turn NIAH reports the five-turn average. For HachiWare, GSM8K uses $I = 4$ and Multi-turn NIAH uses $I = 32$. Mean averages only GSM8K Acc. and Multi-turn NIAH Avg.

Table 3: Reasoning and multi-turn retrieval robustness on Llama-3.1-8B-Instruct. GSM8K is evaluated at approximately 4× compression, and multi-turn NIAH at approximately 8× compression. **Comp.** denotes representation-level compression of the stored KV cache.

method mean accuracy of 85.45. This pattern is consistent with the SCB/EB design: the reconstructable SCB cache helps preserve information useful for future queries, while the EB cache avoids the severe reasoning degradation observed under broader low-rank compression.

Appendix C.3 further broadens the evaluation to LongBench tasks covering long-document QA, multi-hop QA, summarization, retrieval, and code. At 14.40× stored-cache compression, HachiWare achieves 52.19 average score, within 2.71 points of FullKV.

4.4 Robustness to Compression Scaling

To test whether the gains persist beyond the main 8×/16× regimes, we sweep the stored-cache compression ratio on representative RULER subtasks

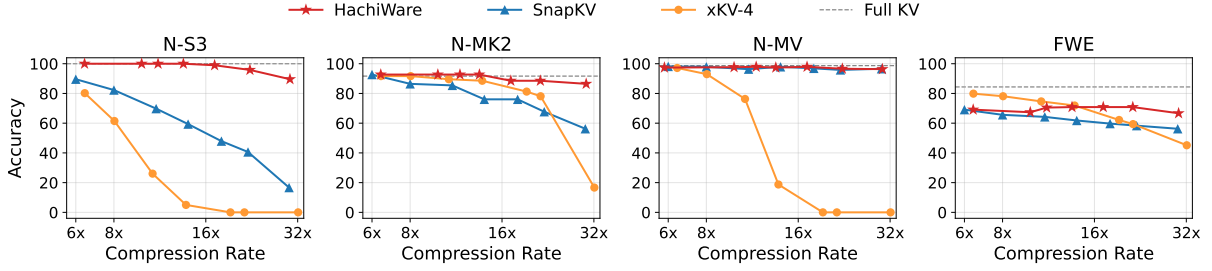


Figure 6: Compression scaling on representative RULER subtasks with Llama-3.1-8B-Instruct. HachiWare remains more stable under aggressive stored-cache compression, especially on retrieval-heavy subtasks. The dashed line denotes Full KV performance.

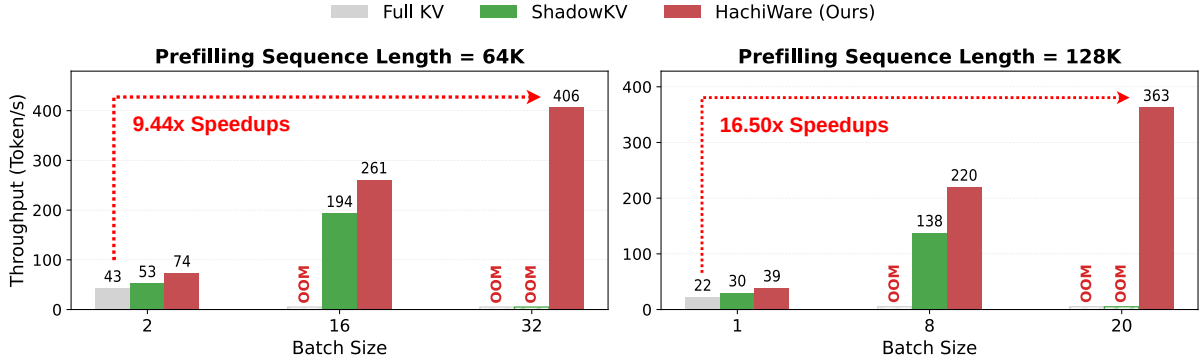


Figure 7: Decoding throughput on Llama-3.1-8B-Instruct at 64K and 128K context lengths using a single NVIDIA A6000 48GB GPU. HachiWare uses the speed-oriented $16\times$, $I = 32$ setting. ShadowKV and HachiWare use the same active-token budget $\kappa = 2048$. All measurements use FlashAttention-2 for attention computation; ShadowKV additionally uses its provided runtime kernel. Higher is better; OOM denotes out-of-memory settings.

with Llama-3.1-8B-Instruct. Figure 6 compares HachiWare with SnapKV and xKV-4, using Full KV as the uncompressed reference. The stress-test row in Table 1 provides a full-average check: at $34.20\times$, HachiWare retains 85.46 accuracy, above the $16\times$ eviction baselines and slightly higher than xKV-4 at $8.03\times$.

HachiWare outperforms both SnapKV and xKV-4 across the evaluated subtasks at high compression ratios. SnapKV remains strong on some token-eviction-friendly cases such as N-MV, but degrades on other retrieval settings under aggressive compression; xKV-4 also drops when the fixed low-rank budget becomes too restrictive. In contrast, HachiWare maintains higher accuracy across N-S3, N-MK2, N-MV, and FWE. These results indicate that maintaining a reconstructable SCB cache while using cheaper eviction for the EB cache is more robust under tight stored-cache budgets than pure token eviction or fixed low-rank grouping.

4.5 Efficiency Studies

Figure 7 reports decoding throughput on Llama-3.1-8B-Instruct at 64K and 128K context lengths using a single NVIDIA A6000 48GB GPU. We use the speed-oriented $16\times$, $I = 32$ setting of HachiWare, with refresh-interval throughput reported in Appendix C.10.3. This setting is accuracy-competitive with ShadowKV in Table 1. For a fair decoding comparison, ShadowKV and HachiWare use the same active-token budget $\kappa = 2048$.

At the largest feasible batch sizes under the same GPU memory limit, HachiWare reaches about 406 tokens/s at 64K and 363 tokens/s at 128K, yielding $9.44\times$ and $16.50\times$ largest-feasible-batch serving-throughput speedups over FullKV, respectively. This comparison captures the serving-capacity benefit of KV cache compression: the reduced memory footprint allows HachiWare to support substantially larger batches on the same GPU, whereas FullKV reaches OOM at smaller batch sizes.

The decoding gains come from selective reconstruction and bounded active-cache refresh: HachiWare limits low-rank reconstruction to the SCB

cache and amortizes selection and reconstruction across multiple decoding steps. Appendix C.10.2 further isolates these costs using an xKV-SR diagnostic, while Appendix A.2 reports auxiliary-buffer-inclusive memory accounting and measured peak GPU allocation. Together with Figure 1, these results show that HachiWare reduces both SVD-related prefill overhead and decoding-time reconstruction overhead while enabling higher serving throughput under the same GPU memory limit.

5 Conclusion

We proposed HachiWare, a one-shot selective SVD framework for KV cache compression. Through its dual-branch design, HachiWare applies SVD only to the SCB cache and uses cheaper eviction for the EB cache, balancing efficiency and information preservation. Compared with broader low-rank methods, it limits prefilling-stage factorization and decoding-time reconstruction overhead while preserving reconstructability for semantic-critical cache states. Bounded active-cache refresh further enables efficient decoding over heterogeneous SCB/EB storage without reconstructing the full SCB cache. Experimental results demonstrate that HachiWare achieves substantial acceleration while maintaining model performance, even under aggressive KV cache compression.

Limitations

HachiWare focuses on compressing the prefilling-stage KV cache. During decoding, newly generated KV states are kept in the active cache but are not incrementally folded back into the one-shot low-rank SCB representation. Very long generation lengths may therefore benefit from additional periodic recompression. HachiWare also relies on offline semantic retrieval scores for branch assignment. These scores are fixed after profiling and require no answer spans, task labels, or per-input scoring during inference. However, a separate profile is required for each backbone, and sensitivity to substantially different profiling or downstream distributions remains to be further studied. Finally, our current evaluation does not include an end-to-end integration with serving frameworks such as vLLM/PagedAttention. The block-granular experiment in Appendix C.6 verifies that HachiWare can operate with block-aligned selection units, while system-level integration with mechanisms such as continuous batching and prefix caching is left for

future work.

Potential risks. HachiWare improves the efficiency of long-context LLM inference by reducing KV-cache memory and access costs. Such efficiency improvements may lower the computational barrier for deploying long-context language models, and may therefore amplify risks already associated with the underlying models, such as incorrect, biased, privacy-sensitive, or unsafe generation in downstream applications. HachiWare does not introduce new training data, collect user data, or add new generation capabilities beyond more efficient inference. Deployments should follow the safety, privacy, and content-moderation practices required for the underlying model and application.

Acknowledgements

This work was supported by the National Science and Technology Council (NSTC), Taiwan, including support through Taiwan-Germany collaborative research projects (Grant no. 115-2923-F-A49-002 and Grant no. 115-2923-F-A49-004).

References

- Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. 2023. [GQA: Training generalized multi-query transformer models from multi-head checkpoints](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4895–4901.
- 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, multi-task benchmark for long context understanding](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3119–3137, Bangkok, Thailand. Association for Computational Linguistics.
- Payman Behnam, Yaosheng Fu, Ritchie Zhao, Po-An Tsai, Zhiding Yu, and Alexey Tumanov. 2025. [RocketKV: Accelerating long-context LLM inference via two-stage KV cache compression](#). In *Proceedings of the 42nd International Conference on Machine Learning*.
- Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, and Wen Xiao. 2025. [PyramidKV: Dynamic KV cache compression based on pyramidal information funneling](#). In *The Second Conference on Language Modeling*.

- Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Chong-Yan Chen, Yu-Fang Hu, Pei-Shuo Wang, Ning-Chi Huang, Luis Ceze, Mohamed S. Abdelfattah, and Kai-Chiang Wu. 2025. [Palu: KV-cache compression with low-rank projection](#). In *The Thirteenth International Conference on Learning Representations*.
- Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Hung-Yueh Chiang, Yash Akhauri, Xilai Dai, Huiqiang Jiang, Yucheng Li, Luis Ceze, Kai-Chiang Wu, and Mohamed S. Abdelfattah. 2026. [xKV: Cross-layer KV-cache compression via aligned singular vector extraction](#). In *Proceedings of the 43rd International Conference on Machine Learning*.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. [Training verifiers to solve math word problems](#). *arXiv preprint arXiv:2110.14168*.
- Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S. Kevin Zhou. 2025. [Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference](#). In *Advances in Neural Information Processing Systems*.
- Yu Fu, Zefan Cai, Abedelkadir Asi, Wayne Xiong, Yue Dong, and Wen Xiao. 2025. [Not all heads matter: A head-level kv cache compression method with integrated retrieval and reasoning](#). In *The Thirteenth International Conference on Learning Representations*.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, et al. 2024. [The Llama 3 herd of models](#). *arXiv preprint arXiv:2407.21783*.
- 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](#). In *Advances in Neural Information Processing Systems*.
- Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesht, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. [RULER: What’s the real context size of your long-context language models?](#) *arXiv preprint arXiv:2404.06654*.
- Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, L  lio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timoth  e Lacroix, and William El Sayed. 2023. [Mistral 7b](#). *arXiv preprint arXiv:2310.06825*.
- Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. [MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention](#). In *Advances in Neural Information Processing Systems*.
- Greg Kamradt. 2023. [Needle in a haystack - pressure testing LLMs](#). GitHub repository.
- Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. 2024. [Gear: An efficient kv cache compression recipe for near-lossless generative inference of llms](#). *arXiv preprint arXiv:2403.05527*.
- Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. 2019. Similarity of neural network representations revisited. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 3519–3529.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. 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.
- Yucheng Li, Huiqiang Jiang, Qianhui Wu, Xufang Luo, Surin Ahn, Chengruidong Zhang, Amir H. Abdi, Dongsheng Li, Jianfeng Gao, Yuqing Yang, and Lili Qiu. 2025. [SCBench: A KV cache-centric analysis of long-context methods](#). In *The Thirteenth International Conference on Learning Representations*.
- Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. [Snapkv: Llm knows what you are looking for before generation](#). In *Advances in Neural Information Processing Systems*.
- Bokai Lin, Zihao Zeng, Zipeng Xiao, Siqu Kou, Tianqi Hou, Xiaofeng Gao, Hao Zhang, and Zhijie Deng. 2025a. [MatryoshkaKV: Adaptive KV compression via trainable orthogonal projection](#). In *The Thirteenth International Conference on Learning Representations*.
- Xiaolin Lin, Jingcun Wang, Olga Kondratyeva, Yiyu Shi, Bing Li, and Grace Li Zhang. 2025b. [Compresskv: Semantic retrieval heads know what tokens are not important before generation](#). *arXiv preprint arXiv:2508.02401*.
- Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024a. [CacheGen: KV cache compression and streaming for fast large language model serving](#). In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 38–56.

- Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhao Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. [Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time](#). In *Advances in Neural Information Processing Systems*.
- Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhao Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024b. [Kivi: A tuning-free asymmetric 2bit quantization for kv cache](#). In *Proceedings of the 41st International Conference on Machine Learning*, pages 32332–32344.
- Mistral AI. 2024. [Mistral-7B-Instruct-v0.3](#). Hugging Face model card.
- Ziran Qin, Yuchen Cao, Mingbao Lin, Wen Hu, Shixuan Fan, Ke Cheng, Weiyao Lin, and Jianguo Li. 2025. [Cake: Cascading and adaptive kv cache eviction with layer preferences](#). In *The Thirteenth International Conference on Learning Representations*.
- Olivier Roy and Martin Vetterli. 2007. The effective rank: A measure of effective dimensionality. In *Proceedings of the 15th European Signal Processing Conference (EUSIPCO)*, pages 606–610.
- Utkarsh Saxena, Gobinda Saha, Sakshi Choudhary, and Kaushik Roy. 2024. [Eigen attention: Attention in low-rank space for kv cache compression](#). In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 15332–15344.
- Konrad Staniszewski and Adrian Łańcucki. 2026. [KV cache transform coding for compact storage in LLM inference](#). In *The Fourteenth International Conference on Learning Representations*.
- Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. 2025. [Shadowkv: Kv cache in shadows for high-throughput long-context llm inference](#). In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 57355–57373. PMLR.
- Hanlin Tang, Yang Lin, Jing Lin, Qingsen Han, Danning Ke, Shikuan Hong, Yiwu Yao, and Gongyi Wang. 2025. [Razorattention: Efficient kv cache compression through retrieval heads](#). In *The Thirteenth International Conference on Learning Representations*.
- Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. [Quest: Query-aware sparsity for efficient long-context llm inference](#). In *Proceedings of the 41st International Conference on Machine Learning*, pages 47901–47911.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. [Attention is all you need](#). In *Advances in Neural Information Processing Systems*, pages 5998–6008.
- Zhongwei Wan, Xinjian Wu, Yu Zhang, Yi Xin, Chaofan Tao, Zhihong Zhu, Xin Wang, Siqu Luo, Jing Xiong, Longyue Wang, and Mi Zhang. 2025. [D₂o: Dynamic discriminative operations for efficient long-context inference of large language models](#). In *The Thirteenth International Conference on Learning Representations*.
- Wenhao Wu, Yizhong Wang, Guangxuan Xiao, Hao Peng, and Yao Fu. 2025. [Retrieval head mechanistically explains long-context factuality](#). In *The Thirteenth International Conference on Learning Representations*.
- Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. 2025. [Duoattention: Efficient long-context llm inference with retrieval and streaming heads](#). In *The Thirteenth International Conference on Learning Representations*.
- Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. [Efficient streaming language models with attention sinks](#). In *The Twelfth International Conference on Learning Representations*.
- An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 23 others. 2024a. [Qwen2.5 technical report](#). *arXiv preprint arXiv:2412.15115*.
- An Yang, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoyan Huang, Jiandong Jiang, Jianhong Tu, Jianwei Zhang, Jingren Zhou, Junyang Lin, Kai Dang, Kexin Yang, Le Yu, Mei Li, Minmin Sun, Qin Zhu, Rui Men, Tao He, and 9 others. 2025. [Qwen2.5-1M technical report](#). *arXiv preprint arXiv:2501.15383*.
- June Yong Yang, Byeongwook Kim, Jeongin Bae, Beomseok Kwon, Gunho Park, Eunho Yang, Se Jung Kwon, and Dongsoo Lee. 2024b. [No token left behind: Reliable KV cache compression via importance-aware mixed precision quantization](#). *arXiv preprint arXiv:2402.18096*.
- Jiayi Yuan, Hongyi Liu, Shaochen Zhong, Yu-Neng Chuang, Songchen Li, Guanchu Wang, Duy Le, Hongye Jin, Vipin Chaudhary, Zhaozhao Xu, Zirui Liu, and Xia Hu. 2024. [KV cache compression, but what must we give in return? a comprehensive benchmark of long context capable approaches](#). In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 4623–4648.
- 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 *Advances in Neural Information Processing Systems*, volume 36.

A Method and Implementation Details

Our implementation builds on an Apache-2.0-licensed codebase, and all external models, benchmarks, and baseline implementations are used under their respective licenses and terms of use.

A.1 Offline SRH Identification Details

Query-head semantic retrieval scores are obtained by reproducing the offline identification protocol of CompressKV (Lin et al., 2025b). This step is performed before evaluation and produces fixed query-head semantic retrieval scores. It is not performed on the evaluation examples of the main benchmarks, and no reference answer span is accessed during inference.

For each backbone model, offline identification prompts are used to collect attention weights during generation. When the generated token belongs to the reference answer span, the attention mass assigned by each query head to the full answer span is accumulated according to Eq. (1). This follows the SRH criterion of CompressKV, which scores heads by answer-span attention mass rather than requiring an exact top-1 attention hit on a single answer token.

The resulting profile is a table of query-head semantic retrieval scores $\{s_{\ell,h}^Q\}$ for each backbone model. These scores are fixed after offline identification and reused for all downstream inputs. HachiWare does not update semantic retrieval scores during decoding and does not recompute them per input.

The precomputed scores are used in two places. First, Section 3.1 aggregates query-head scores into KV-head semantic retrieval scores for SCB/EB routing. Second, the Eviction Branch follows the SRH-driven token-eviction interface of CompressKV: within each layer, high-scoring SRH query heads selected by $s_{\ell,h}^Q$ provide the attention patterns used to compute token-level retention scores for initializing the exact EB cache during prefill-stage cache preparation.

Unless otherwise specified, the SRH-driven eviction settings follow CompressKV (Lin et al., 2025b). HachiWare-specific hyperparameters used in the main experiments are reported in Appendix A.3.

A.2 Compression and Runtime Memory Accounting

Representation-level compression. For accuracy–compression comparisons, we define the reported compression ratio as

$$\text{Comp.} = \frac{\text{Mem}_{\text{full}}}{\text{Mem}_{\text{stored}}},$$

where Mem_{full} is the size of the full-precision prefilling KV cache and $\text{Mem}_{\text{stored}}$ is the size of the stored compressed representation after prefilling.

For stored-cache compression methods, $\text{Mem}_{\text{stored}}$ counts the compressed KV cache representation kept after prefilling. For token-eviction methods, it counts the retained exact KV states. For quantization methods, it counts the quantized KV representation. For selection-only methods such as Quest, $\text{Mem}_{\text{stored}} = \text{Mem}_{\text{full}}$, and therefore $\text{Comp.} = 1.00\times$.

HachiWare stored representation. Let b denote the number of bytes per stored element, S the prefilling sequence length, d_h the head dimension, L_{layer} the number of transformer layers, and H_{KV} the number of KV heads per layer. The full prefilling KV cache size is

$$\text{Mem}_{\text{full}} = 2bSd_hL_{\text{layer}}H_{\text{KV}},$$

where the factor 2 accounts for keys and values.

HachiWare routes M_{SC} KV heads to the Semantic-Critical Branch. In all main experiments, $M_{\text{SC}} = 32$. The SCB stores pre-RoPE keys and values using separate low-rank factors. With key rank r_K and value rank r_V , the stored SCB memory is

$$\text{Mem}_{\text{SCB}} = br_K(S + M_{\text{SC}}d_h) + br_V(S + M_{\text{SC}}d_h).$$

The first term counts the key-side factors, and the second term counts the value-side factors.

The remaining

$$M_{\text{EB}} = L_{\text{layer}}H_{\text{KV}} - M_{\text{SC}}$$

KV heads form the Eviction Branch. With active selection budget κ , which is also used as the initial EB prefill retention budget, the stored EB memory is

$$\text{Mem}_{\text{EB}} = 2b\kappa M_{\text{EB}}d_h.$$

Thus, the HachiWare compression ratio reported in the tables is

$$\text{Comp.}_{\text{HW}} = \frac{\text{Mem}_{\text{full}}}{\text{Mem}_{\text{SCB}} + \text{Mem}_{\text{EB}}}.$$

Method	Stored KV	KV w/ Aux.	Stored Comp.	Comp. w/ Aux.	Peak GPU (BS=1)	Peak GPU (BS=2)
FullKV	8.00 GiB	8.00 GiB	1.00×	1.00×	23.04 GiB	31.07 GiB
HachiWare	0.50 GiB	0.59 GiB	16.04×	13.50×	15.93 GiB	16.85 GiB

Table 4: Memory accounting for Llama-3.1-8B-Instruct at 64K under the 16× HachiWare configuration. Stored KV reports the prefilling KV representation, while KV w/ Aux. additionally includes the auxiliary buffers used by the runtime fast path. Peak GPU allocation is measured during decoding and includes the full runtime allocation.

Model	Target Comp.	SCB heads	Anchor heads	κ	r_K	r_V	Stored Comp.	Comp. w/ Aux.
Llama-3.1-8B	8×	32	4	4096	1536	2304	8.55×	7.54×
Llama-3.1-8B	16×	32	4	2048	864	1296	16.04×	13.50×
Llama-3.1-8B	Stress	32	4	1024	384	576	34.20×	25.64×
Mistral-7B-v0.3	8×	32	2	2048	1536	2304	8.29×	7.56×
Mistral-7B-v0.3	16×	32	2	1024	768	1152	16.58×	14.68×

Table 5: HachiWare configurations used in the main experiments. Target Comp. denotes the intended stored-cache compression regime. Stored Comp. reports the representation-level compression ratio after prefilling. Comp. w/ Aux. additionally includes HachiWare’s auxiliary runtime buffers described in Appendix A.2. κ is the active selection budget per KV head after each refresh. r_K and r_V are the SVD ranks for the pre-RoPE key and value caches.

Auxiliary runtime buffers. The representation-level compression ratio above counts only the stored compressed representation after prefilling. For runtime efficiency, HachiWare additionally materializes auxiliary buffers that trade memory for speed.

First, HachiWare caches the anchor-key states used for shared ordered-list construction. This avoids reconstructing anchor keys from the SCB factors at every refresh boundary. If M_{anc} anchor key heads are selected, this buffer has size

$$\text{Mem}_{\text{anchor}} = bM_{\text{anc}}Sd_h.$$

In our experiments, $M_{\text{anc}} = 4$ for Llama-3.1-8B-Instruct and $M_{\text{anc}} = 2$ for Mistral-7B-Instruct-v0.3.

Second, after each active-cache refresh, HachiWare materializes the selected reconstructed SCB entries and reuses them until the next refresh. This avoids reconstructing SCB entries at every decoding step. For one cache type, this buffer is proportional to

$$bM_{\text{SCB}}\kappa d_h.$$

When both reconstructed keys and values are materialized, the corresponding buffer size is

$$2bM_{\text{SCB}}\kappa d_h.$$

These auxiliary buffers are not included in the representation-level Comp. metric. They are instead part of the runtime memory–speed trade-off and are discussed in the efficiency analysis.

Runtime memory validation. Table 4 reports memory usage for the Llama-3.1-8B-Instruct 64K 16× configuration used in the main efficiency study. The stored HachiWare KV representation occupies 0.50 GiB, compared with 8.00 GiB for FullKV. Including the auxiliary runtime buffers increases this to 0.59 GiB, an overhead of only 0.09 GiB. The corresponding stored-cache and auxiliary-inclusive compression ratios are 16.04× and 13.50×, respectively. Measured peak GPU allocation during decoding also remains lower than FullKV at the same batch sizes.

A.3 Implementation Details

HachiWare hyperparameters. For both backbone models, HachiWare uses an SCB routing budget of 32 KV heads. The KV heads routed to the SCB are selected globally by the KV-head semantic retrieval scores in Eq. (2). The selected SCB KV heads are concatenated along the head/feature dimension and factorized by one-shot SVD, as described in Section 3.2. Pre-RoPE keys and values are factorized separately, and all configurations use a 2:3 key/value SVD rank ratio.

Table 5 lists the HachiWare configurations used in the main experiments. The active-token budget κ denotes the number of selected historical token positions per KV head immediately after each refresh. The reported compression ratio is the representation-level stored-cache compression ratio defined in Appendix A.2.

Baseline hyperparameters. For PyramidKV, SnapKV, KIVI, and Quest, we use the MInference implementations and keep their default implementation-specific settings unless stated otherwise. For PyramidKV and SnapKV, we use an observation window size of 8 to match the SRH-driven eviction observation window used in HachiWare.

For ShadowKV accuracy evaluation, we use an offloading-free simulation of the algorithmic cache representation, since value offloading changes device placement and runtime behavior but does not change the numerical attention result. The compression-matched ShadowKV configurations use budget/rank/chunk-size/outlier-chunk values of 2048/108/8/48 for Llama-3.1-8B-Instruct and 1024/108/8/24 for Mistral-7B-Instruct-v0.3.

Intended use of artifacts. All existing artifacts are used for standard research evaluation of efficient LLM inference. The models are used for inference-only evaluation, the benchmarks are used for their intended evaluation purposes, and the baseline implementations are used to compare KV-cache compression and efficient inference methods under matched settings. We do not use the artifacts for deployment, user-facing decision making, or data collection.

A.4 Active-Cache Refresh Details

This appendix provides the details behind the active-cache refresh mechanism summarized in Section 3.3. We first describe how anchor key heads are selected after SCB routing, then define the shared ordered list used at refresh points, and finally give the refresh procedure and bounded-cache complexity.

A.4.1 Anchor-Head Selection

Anchor selection is performed after SCB routing. Let $\mathcal{M}_\ell^{\text{SC}}$ denote the KV heads in layer ℓ that have been routed to the Semantic-Critical Branch. Since the shared ordered list used during decoding is computed from a single layer, anchor key heads are selected as a same-layer subset of the SCB. This keeps the ordered-list construction overhead small while ensuring that the ordering signal comes from semantic-critical KV heads.

For each layer with non-empty $\mathcal{M}_\ell^{\text{SC}}$, we first sort its SCB KV heads by their KV-level semantic retrieval scores $s_{\ell,m}^{\text{KV}}$. We then apply a relative gap rule to obtain a compact candidate set. Specifically,

Algorithm 1 Adaptive Anchor Key Head Selection

Require: SCB KV-head sets $\{\mathcal{M}_\ell^{\text{SC}}\}$;
KV-head semantic scores $s_{\ell,m}^{\text{KV}}$;
gap ratio γ ;
anchor bounds $n_{\min}, n_{\max}$

Ensure: anchor layer ℓ_a ; anchor key heads $\mathcal{M}^{\text{anc}}$

▷ *Track the best anchor layer*

- 1: $S_{\text{best}}^{\text{anc}} \leftarrow -\infty$
- 2: **for** each layer ℓ with $\mathcal{M}_\ell^{\text{SC}} \neq \emptyset$ **do**
- 3: $\mathcal{H}_\ell \leftarrow \text{Sort}_{\downarrow}(\mathcal{M}_\ell^{\text{SC}}, s_{\ell,m}^{\text{KV}})$
▷ *Keep a top prefix close to the strongest head*
- 4: $\mathcal{M}_\ell^{\text{keep}} \leftarrow \text{GapPrefix}(\mathcal{H}_\ell, s^{\text{KV}}, \gamma, n_{\max})$
- 5: **if** $|\mathcal{M}_\ell^{\text{keep}}| < n_{\min}$ **then**
- 6: **continue**
- 7: **end if**
▷ *Score by weakest kept head and anchor count*
- 8: $w_\ell \leftarrow \text{WeakestScore}(\mathcal{M}_\ell^{\text{keep}}, s^{\text{KV}})$
- 9: $S_\ell^{\text{anc}} \leftarrow w_\ell \sqrt{|\mathcal{M}_\ell^{\text{keep}}|}$
- 10: **if** $S_\ell^{\text{anc}} > S_{\text{best}}^{\text{anc}}$ **then**
- 11: $S_{\text{best}}^{\text{anc}} \leftarrow S_\ell^{\text{anc}}$
- 12: $\ell_a \leftarrow \ell$
- 13: $\mathcal{M}^{\text{anc}} \leftarrow \mathcal{M}_\ell^{\text{keep}}$
- 14: **end if**
- 15: **end for**
- 16: **return** $\ell_a, \mathcal{M}^{\text{anc}}$

GapPrefix returns the longest top prefix of the sorted head list, capped by $n_{\max}$, whose scores are at least γ times the top score. Layers with fewer than $n_{\min}$ retained heads are discarded, so that the shared ordering does not rely on a single isolated high-scoring head.

Among the remaining layers, we select the anchor layer using a robust layer score. For a retained candidate set $\mathcal{M}_\ell^{\text{keep}}$, the score is

$$S_\ell^{\text{anc}} = \text{WeakestScore}(\mathcal{M}_\ell^{\text{keep}}, s^{\text{KV}}) \sqrt{|\mathcal{M}_\ell^{\text{keep}}|}. \quad (5)$$

The weakest-score term favors layers whose retained anchors are consistently strong, while the square-root term gives a sub-linear bonus to layers with multiple retained anchors. The layer with the largest S_ℓ^{anc} is chosen as the anchor layer, and its retained heads become the anchor key heads. Algorithm 1 summarizes this selection procedure.

The selected anchor key heads are used only to construct the shared token ordering during runtime refresh. Value states are not used in this scoring; the

ordering is derived from query–key affinity scores of the selected anchor key heads and is then reused across layers. In our experiments, the procedure selects four anchor key heads for Llama-3.1-8B-Instruct and two anchor key heads for Mistral-7B-Instruct-v0.3.

A.4.2 Shared Ordered-List Construction

At each refresh point, when the runtime active cache is updated, we sort the available token positions $\mathcal{I}$ into a shared ordered list. Here, $\mathcal{I}$ denotes all token positions currently available to attention.

For the initial refresh at the end of prefilling, we use the last prefill token’s query at the anchor layer ℓ_a as the selector query. For later refreshes, we use the current decoding-step query at the anchor layer. The selector query is paired with post-RoPE anchor-key states over $\mathcal{I}$ to score the available token positions.

Let $\mathcal{M}^{\text{anc}}$ denote the anchor key heads. For each anchor key head $m \in \mathcal{M}^{\text{anc}}$, let $\mathcal{Q}_{\ell_a, m}$ denote the query heads in the anchor layer that share this key head. We write $Q_{\ell_a, m}^{\text{anc}} \in \mathbb{R}^{|\mathcal{Q}_{\ell_a, m}| \times d_h}$ for the selector queries of these query heads, and $\tilde{K}_{\ell_a, m}^{\text{anc}} \in \mathbb{R}^{|\mathcal{I}| \times d_h}$ for the corresponding post-RoPE anchor-key states over $\mathcal{I}$.

For each anchor key head m , we compute query–key affinities:

$$P^{(m)} = Q_{\ell_a, m}^{\text{anc}} \left(\tilde{K}_{\ell_a, m}^{\text{anc}} \right)^\top. \quad (6)$$

Here, $P^{(m)} \in \mathbb{R}^{|\mathcal{Q}_{\ell_a, m}| \times |\mathcal{I}|}$, with one row per associated query head and one column per available token position.

For each $j \in \mathcal{I}$, we average affinities first over query heads sharing each anchor key head, then over anchor key heads:

$$\rho_j = \frac{1}{|\mathcal{M}^{\text{anc}}|} \sum_{m \in \mathcal{M}^{\text{anc}}} \left(\frac{1}{|\mathcal{Q}_{\ell_a, m}|} \sum_{h \in \mathcal{Q}_{\ell_a, m}} P_{h, j}^{(m)} \right). \quad (7)$$

Following StreamingLLM (Xiao et al., 2024), sink and recent tokens are always retained as mandatory runtime states. We enforce this by lifting their ordering scores above all other candidates before sorting. Sorting $\mathcal{I}$ by ρ in descending order yields $\mathcal{I}^{\text{ord}}$, with sink/recent indices placed first so they are included under the κ -entry selection.

A.4.3 Refresh from the Shared Ordered List

Active-cache refresh avoids reconstructing the full SVD-compressed SCB during decoding. At each

refresh point, HachiWare uses the shared ordered list to select token entries, reconstructs only the selected SCB entries from low-rank factors, and gathers selected EB entries from retained exact KV states. These selected entries form the runtime active cache used for attention.

At the end of prefilling and at later refresh points, the runtime active cache is refreshed from the shared ordered list $\mathcal{I}^{\text{ord}}$. For the SCB, selected entries are reconstructed from the low-rank factors. For the EB, selected entries are gathered from the current feasible exact cache. Immediately after prefilling, the EB feasible set is initialized by the prefill eviction rule. At later refresh points, it may include EB states retained from the previous refresh, newly decoded states, and mandatory sink/recent states.

For consistency with Eq. (4), let B_ℓ collect the right blocks B_i whose routed SCB KV heads belong to layer ℓ . If layer ℓ has no routed SCB head, B_ℓ is empty. For the EB, let $\mathcal{I}_\ell^{\text{EB}}$ denote the feasible token indices of layer ℓ at the current refresh point, and let $\mathcal{C}_\ell^{\text{EB}}$ denote the corresponding exact KV states.

In Algorithm 2, $\mathcal{I}^{\text{ord}} \cap \mathcal{I}_\ell^{\text{EB}}$ denotes order-preserving filtering: entries are kept in the order of $\mathcal{I}^{\text{ord}}$, while indices not contained in $\mathcal{I}_\ell^{\text{EB}}$ are removed. The active-token budget κ controls the number of entries selected from the resulting ordered list.

Algorithm 2 summarizes the active-cache refresh procedure and outputs the runtime active cache $\mathcal{C}_\ell^{\text{act}}$ for each layer. For compactness, the algorithm omits separate key/value notation; in implementation, refresh is applied to the corresponding key and value representations, and RoPE is recomputed only for reconstructed key states. Between refresh points, newly generated KV states are added to the active cache and kept until the next refresh.

A.4.4 Bounded Active Cache and Complexity

The active-token budget κ in Algorithm 2 controls the number of historical token positions kept per KV head after each refresh. The same κ is used for SCB selection and EB selection during refresh, and also sets the initial EB prefill retention size after SRH-driven eviction. Sink and recent tokens are included in this κ -entry selection through the score lifting described above.

Immediately after a refresh, each KV head attends to κ selected historical token positions from its branch. Between refresh points, newly generated KV states are added to the active cache and

Algorithm 2 Active Cache Refresh

Require: $\mathcal{I}^{\text{ord}}$, active-token budget κ ;
 SCB factors $A, \{B_\ell\}$;
 current feasible EB caches $\{\mathcal{C}_\ell^{\text{EB}}\}$
 with token indices $\{\mathcal{I}_\ell^{\text{EB}}\}$

Ensure: runtime active caches $\{\mathcal{C}_\ell^{\text{act}}\}$

- ▷ *Select SCB token indices*
- 1: $\mathcal{J}^{\text{SC}} \leftarrow \text{FirstK}(\mathcal{I}^{\text{ord}}, \kappa)$
- ▷ *Gather selected rows of A*
- 2: $A_{\text{sel}} \leftarrow \text{Gather}(A, \mathcal{J}^{\text{SC}})$
- 3: **for** each layer ℓ **do**
- ▷ *Reconstruct selected SCB cache entries*
- 4: **if** layer ℓ contains SCB KV heads **then**
- 5: $\hat{\mathcal{C}}_\ell^{\text{SC}} \leftarrow \text{MatMul}(A_{\text{sel}}, B_\ell)$
- 6: **else**
- 7: $\hat{\mathcal{C}}_\ell^{\text{SC}} \leftarrow \emptyset$
- 8: **end if**
- ▷ *Select EB token indices*
- 9: $\mathcal{J}_\ell^{\text{EB}} \leftarrow \text{FirstK}(\mathcal{I}^{\text{ord}} \cap \mathcal{I}_\ell^{\text{EB}}, \kappa)$
- ▷ *Gather selected EB cache entries*
- 10: $\hat{\mathcal{C}}_\ell^{\text{EB}} \leftarrow \text{Gather}(\mathcal{C}_\ell^{\text{EB}}, \mathcal{J}_\ell^{\text{EB}})$
- ▷ *Merge branches for attention*
- 11: $\mathcal{C}_\ell^{\text{act}} \leftarrow \text{Merge}(\hat{\mathcal{C}}_\ell^{\text{SC}}, \hat{\mathcal{C}}_\ell^{\text{EB}})$
- 12: **end for**
- 13: **return** $\{\mathcal{C}_\ell^{\text{act}}\}$

remain active until the next refresh. With refresh interval I , the active token count per KV head ranges from κ to at most $\kappa + I$.

Additional work occurs only at refresh points. $\mathcal{I}^{\text{ord}}$ is recomputed from anchor query–key scores over $\mathcal{I}$, which is dense over the available token positions but restricted to the anchor key heads in the anchor layer. Once $\mathcal{I}^{\text{ord}}$ is available, active-cache refresh reconstructs only the selected SCB entries and gathers selected exact EB entries. These costs are amortized across the I -step refresh interval.

B Analysis Protocols

B.1 Attention Recovery Protocol

Figure 8 measures how well the bounded active cache of HachiWare preserves the full-cache attention distribution. This appendix describes the protocol.

Setup. We first produce a decoded token stream with a full-KV reference run, then use this same stream to teacher-force (i) a HachiWare pass with active selection budget κ and (ii) a full-KV oracle replay. At each decode step t , let $\mathcal{I}_t$ denote the

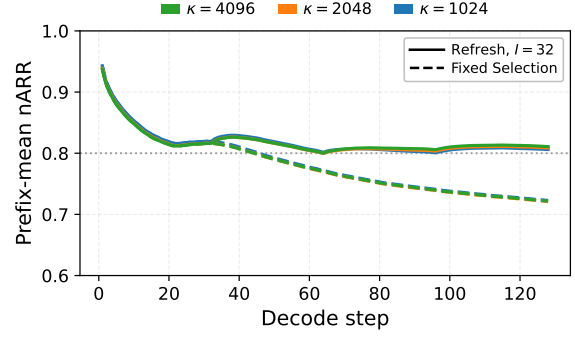


Figure 8: Effect of active-cache refresh on prefix-mean nARR for RULER NIAH-S3 with Llama-3.1-8B-Instruct. Solid lines refresh active-token selection every $I = 32$ decoding steps; dashed lines keep the initial selection fixed. Periodic refresh stabilizes attention recovery, while fixed selection degrades. nARR is normalized by a same-budget Full-KV oracle.

token positions available to attention, including prefill positions and previously decoded positions, and let $\mathcal{J}_t \subseteq \mathcal{I}_t$ denote the active token selection in force for the KV head being evaluated at step t . In Algorithm 2, this corresponds to $\mathcal{J}^{\text{SC}}$ for SCB heads and $\mathcal{J}_\ell^{\text{EB}}$ for EB heads. The selection is held fixed between two refresh points.

For each layer and KV head, the oracle replay produces a post-softmax attention distribution over $\mathcal{I}_t$, GQA-averaged over the query heads sharing that KV head. We write $\alpha_{t,\cdot}$ for one such distribution, with $\sum_{j \in \mathcal{I}_t} \alpha_{t,j} = 1$. The metrics below are defined on a single attention distribution $\alpha_{t,\cdot}$, then averaged across all layers, KV heads, and evaluation samples for the final report.

Attention recall. At step t , the attention recall rate ARR_t is the oracle mass captured by the active selection, and the normalized recall nARR_t divides it by the best top- k mass under the same oracle distribution and budget:

$$\text{ARR}_t = \sum_{j \in \mathcal{J}_t} \alpha_{t,j}, \quad (8)$$

$$\text{nARR}_t = \frac{\text{ARR}_t}{\sum_{j \in \text{TopK}_{k_t}(\alpha_{t,\cdot})} \alpha_{t,j}}, \quad k_t = |\mathcal{J}_t|. \quad (9)$$

Here, $\text{TopK}_k(\cdot)$ returns the indices of the k largest entries. The normalization in Eq. (9) removes the dependence on the inherent peakiness of $\alpha_{t,\cdot}$, placing nARR_t in $[0, 1]$: a value of one means that $\mathcal{J}_t$ covers as much oracle attention mass as an oracle top- k_t selection at step t .

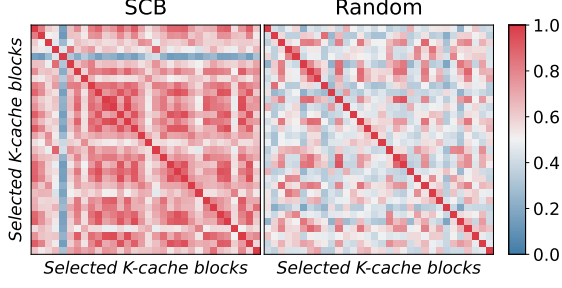


Figure 9: Pairwise CKA analysis of selected K-cache blocks on the pre-RoPE K-cache. SCB denotes the top-32 KV heads ranked by KV-head semantic retrieval scores, while Random denotes 32 KV heads sampled from all layers. Each axis indexes the selected K-cache blocks, and warmer values indicate stronger token-side representation alignment. The SCB grouping shows stronger pairwise alignment than Random.

Prefix mean. nARR_t fluctuates within each refresh interval: it is highest at a refresh point and can decay as more decoding steps are taken under the same frozen selection. Since each decoded token conditions on all preceding ones, generation quality reflects the accumulated recall over the decoded prefix rather than its instantaneous value. We therefore report the prefix mean

$$\widetilde{\text{nARR}}_t = \frac{1}{t} \sum_{\tau=1}^t \text{nARR}_\tau, \quad (10)$$

which cancels intra-interval oscillation and reflects the average recall accumulated from the start of decoding through step t . Figure 8 plots $\widetilde{\text{nARR}}_t$ against t for several active selection budgets κ .

B.2 SCB Structure Analysis Protocol

This appendix provides additional analysis and protocol details for the SCB structure study in Section 3.2. Pairwise CKA in Figure 9 compares SCB with Random to assess token-side representation alignment. Rank sensitivity in Figure 5 compares SCB, Random, xKV-style, and ShadowKV-style groupings to assess singular-value energy retention.

B.2.1 Pairwise CKA Analysis

Figure 9 shows that the SCB grouping has stronger pairwise CKA than Random, indicating greater token-side representation alignment among the selected K-cache blocks. This supports the main-text rank-sensitivity result: SCB routing forms a cache matrix with stronger shared structure and is more amenable to one-shot SVD.

B.2.2 Pairwise CKA Protocol

This appendix describes the protocol behind Figure 9.

Per-head representation. For each prefilling forward pass on the Multi-Query NIAH task from RULER with Llama-3.1-8B-Instruct, we cache the per-layer KV tensors $\mathbf{K}_\ell^{\text{pre}}, \mathbf{K}_\ell^{\text{post}}, \mathbf{V}_\ell \in \mathbb{R}^{N_{\text{ex}} \times H_{\text{KV}} \times L \times d_h}$, where N_{ex} is the number of evaluation examples, H_{KV} is the number of KV heads per layer, L is the prefilling sequence length, and d_h is the head dimension. The figure uses the pre-RoPE keys, since these are the states compressed by one-shot selective SVD. For a selected KV head (ℓ, m) , the head representation is

$$\mathbf{X}_{(\ell, m)} = [\mathbf{K}_\ell^{\text{pre}}]_{:, m, :, :} \in \mathbb{R}^{N_{\text{ex}} \times L \times d_h}. \quad (11)$$

We flatten the example and token axes to obtain $\mathbf{X}_{(\ell, m)} \in \mathbb{R}^{N_{\text{tok}} \times d_h}$, where $N_{\text{tok}} = N_{\text{ex}}L$.

Head selection. We use $M_{\text{eval}} = 32$ KV heads in total. SCB uses the top-32 (ℓ, m) pairs ranked by KV-head semantic retrieval scores, while Random uses 32 KV heads sampled uniformly at random from the same global candidate set. Both selections are drawn once and held fixed across all examples.

Linear centered CKA. For any pair of head representations $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{N_{\text{tok}} \times d_h}$, we first center each column-wise over the flattened example-token axis, giving $\tilde{\mathbf{X}} = \mathbf{X} - \bar{\mathbf{X}}$ and $\tilde{\mathbf{Y}} = \mathbf{Y} - \bar{\mathbf{Y}}$. The linear centered CKA is

$$\text{CKA}(\mathbf{X}, \mathbf{Y}) = \frac{\|\tilde{\mathbf{X}}^\top \tilde{\mathbf{Y}}\|_F^2}{\|\tilde{\mathbf{X}}^\top \tilde{\mathbf{X}}\|_F \|\tilde{\mathbf{Y}}^\top \tilde{\mathbf{Y}}\|_F}, \quad (12)$$

which is the linear-kernel form of CKA (Kornblith et al., 2019) and is invariant to orthogonal transformations and isotropic scaling of either representation. The Gram-matrix norms are accumulated in double precision for numerical stability, and pairs with vanishing denominator ($< 10^{-12}$) are set to zero.

Reporting. Pairwise CKA is computed for every ordered pair of the 32 selected heads, yielding a symmetric 32×32 matrix with unit diagonal. Figure 9 visualizes these matrices; both panels share the same color scale $[0, 1]$. The reported mean similarity excludes the diagonal.

B.2.3 Rank Sensitivity of KV-Head Groupings

This appendix describes the rank-sensitivity protocol behind Figure 5, including how each grouping forms its cache matrix and how the explained-variance curves are computed.

Cache matrix from a head group. Given a set of KV heads $\mathcal{G}$ and per-layer dumps $\mathbf{K}_\ell^{\text{pre}}, \mathbf{V}_\ell \in \mathbb{R}^{N_{\text{ex}} \times H_{\text{KV}} \times L \times d_h}$ collected on the same Multi-Query NIAH samples as the CKA analysis above, we slice the heads in $\mathcal{G}$ and concatenate them along the head/feature dimension. This gives a grouped cache tensor $\mathbf{X}_\mathcal{G} \in \mathbb{R}^{N_{\text{ex}} \times L \times d_\mathcal{G}}$, where $d_\mathcal{G} = |\mathcal{G}|d_h$. We then flatten the example and token axes to form the cache matrix

$$\mathbf{M}_\mathcal{G} \in \mathbb{R}^{N_{\text{tok}} \times d_\mathcal{G}}, \quad N_{\text{tok}} = N_{\text{ex}}L. \quad (13)$$

The matrix is centered column-wise over the flattened example-token axis, $\mathbf{M}_\mathcal{G} \leftarrow \mathbf{M}_\mathcal{G} - \bar{\mathbf{M}}_\mathcal{G}$. The same construction can be applied to values; Figure 5 reports the pre-RoPE key-cache results used in the main analysis.

Group definitions. For the rank-sensitivity analysis in Figure 5, four head groupings are evaluated, each producing one curve:

- **SCB.** $\mathcal{G}$ is the top-32 (ℓ, m) pairs ranked by aggregated KV-head semantic retrieval scores, grouped by layer and concatenated across the selected heads.
- **Random.** $|\mathcal{G}| = 32$ heads are sampled uniformly at random from the same global candidate set.
- **xKV- n_{grp} (Chang et al., 2026).** For every group of n_{grp} consecutive layers, $\mathcal{G}$ is the union of all H_{KV} KV heads in those layers; the reported curve is the mean of the per-group explained-variance curves. The main figure uses $n_{\text{grp}} = 4$.
- **Per-layer.** $\mathcal{G}$ contains all H_{KV} KV heads of a single layer; the reported curve is the mean of the per-layer curves.

Shared- U low-rank model. Following the storage form used in xKV and our SCB compression, each cache matrix is approximated by a rank- r factorization that shares a token-side basis across heads. We absorb the singular values into the right factor:

$$\begin{aligned} \mathbf{M}_\mathcal{G} &\approx \mathbf{U}_r \mathbf{R}_r, \\ \mathbf{U}_r &\in \mathbb{R}^{N_{\text{tok}} \times r}, \quad \mathbf{R}_r \in \mathbb{R}^{r \times d_\mathcal{G}}. \end{aligned} \quad (14)$$

The dense cache matrix has footprint $N_{\text{tok}}d_\mathcal{G}$, while the shared- U representation has footprint $rN_{\text{tok}} + rd_\mathcal{G} = r(N_{\text{tok}} + d_\mathcal{G})$. We therefore index rank sensitivity by the shared- U compression ratio

$$c_\mathcal{G}(r) = \frac{N_{\text{tok}}d_\mathcal{G}}{r(N_{\text{tok}} + d_\mathcal{G})}. \quad (15)$$

Given a target compression ratio c , the corresponding rank is

$$r(c) = \text{round}\left(\frac{N_{\text{tok}}d_\mathcal{G}}{c(N_{\text{tok}} + d_\mathcal{G})}\right), \quad (16)$$

clipped to $[1, \min(N_{\text{tok}}, d_\mathcal{G})]$. This lets curves with different $d_\mathcal{G}$ be compared at matched storage budgets.

Explained variance. We compute the full singular spectrum $\{\sigma_i\}_{i=1}^{q_{\text{max}}}$, where $q_{\text{max}} = \min(N_{\text{tok}}, d_\mathcal{G})$, of the centered $\mathbf{M}_\mathcal{G}$ once per group, and read off the cumulative explained variance at every queried rank:

$$\text{EV}(r) = \frac{\sum_{i=1}^r \sigma_i^2}{\sum_{i=1}^{q_{\text{max}}} \sigma_i^2}. \quad (17)$$

Figure 5 plots $\text{EV}(r)$ against $c_\mathcal{G}(r)$ from Eq. (15).

Effective rank. As a scalar summary independent of the chosen budget, we also report the energy-based effective rank

$$\text{ER}(\mathbf{M}_\mathcal{G}) = \exp\left(-\sum_{i=1}^{q_{\text{max}}} p_i \log p_i\right), \quad (18)$$

where $p_i = \sigma_i^2 / \sum_{j=1}^{q_{\text{max}}} \sigma_j^2$ is the normalized energy of the i -th singular direction. Equation (18) is the exponential of the Shannon entropy over the squared-singular-value energy distribution (Roy and Vetterli, 2007). Lower values indicate that the group’s KV cache concentrates its energy in fewer principal directions and is therefore more amenable to one-shot low-rank compression.

C Additional Results and Ablations

C.1 Benchmark Statistics

Table 6 summarizes the evaluation sizes for RULER, multi-turn NIAH, and GSM8K. For RULER, each subtask is generated with 96 examples using a fixed random seed; multi-turn NIAH subtasks expand each example into five conversational turns. Because tokenization affects how the haystack is packed, RULER examples are regenerated per model template and context length, with no additional subsampling. For GSM8K, we evaluate the full test split with 5-shot prompting.

Benchmark	Split	# examples	Notes
RULER	validation	96 per subtask	Generated with seed 42
RULER multi-turn NIAH	validation	96 per subtask	5 turns per example
GSM8K	test	1,319	Full split, 5-shot

Table 6: Evaluation statistics for RULER, multi-turn NIAH, and GSM8K.

Method	Comp.	N-S1	N-S2	N-S3	N-MK1	N-MK2	N-MQ	N-MV	QA-1	QA-2	VT	FWE	Avg.
FullKV	1.00	100.00	94.79	100.00	97.91	65.62	94.79	95.05	67.70	50.00	91.04	93.05	86.36
KIVI-2	7.10	97.91	78.12	27.08	67.70	9.37	56.77	36.97	57.29	47.91	78.75	89.93	58.89
xKV-4	8.27	100.00	75.00	89.58	86.45	60.41	88.02	75.26	93.33	89.58	60.41	48.95	78.82
PyramidKV	8.00	98.95	91.66	58.33	91.66	30.20	48.43	50.78	69.79	51.04	89.16	85.41	69.58
SnapKV	8.00	100.00	90.62	55.20	89.58	33.33	49.73	47.65	69.79	51.04	87.50	89.93	69.49
Quest	1.00 (8.00)	95.83	72.91	77.08	97.91	66.66	80.20	83.85	70.83	46.87	89.79	82.98	78.63
ShadowKV	1.82 (9.02)	100.00	92.70	77.08	87.50	52.08	70.05	71.09	64.58	50.00	84.79	88.88	76.25
HachiWare_{I=2}	8.29	100.00	96.87	100.00	95.83	58.33	86.19	83.07	67.70	51.04	90.00	80.20	82.66
HachiWare_{I=32}	8.29	100.00	95.83	81.25	95.83	38.54	75.00	80.46	68.75	51.04	88.95	86.11	78.34
PyramidKV	16.00	97.91	77.08	19.79	83.33	20.83	24.73	27.60	67.70	52.08	87.70	81.59	58.21
SnapKV	16.00	97.91	76.04	22.91	83.33	16.66	29.42	23.43	68.75	51.04	87.91	86.11	58.50
HachiWare_{I=2}	16.58	100.00	95.83	100.00	95.83	59.37	85.15	75.52	65.62	51.04	88.33	78.81	81.41
HachiWare_{I=32}	16.58	100.00	94.79	44.79	95.83	20.83	63.80	73.47	67.70	51.04	88.33	80.55	71.01
HachiWare_{I=32}	22.82 [†]	100.00	73.95	42.70	87.50	20.83	55.20	60.67	66.66	50.00	88.54	83.33	66.31

Notes. Task abbreviations follow Table 1. Gray parentheses denote effective GPU-side cache reduction for Quest and ShadowKV. [†] marks the stress-test setting beyond the main 8×/16× regime.

Table 7: Main RULER results on Mistral-7B-Instruct-v0.3 at 32K context length. **Comp.** denotes representation-level compression of the stored KV cache; gray parentheses denote effective GPU-side cache reduction for selection/offloading baselines.

C.2 Full Mistral RULER Results

Table 7 reports the full per-task RULER results on Mistral-7B-Instruct-v0.3 at 32K context length, complementing the average results reported in the main text. The per-task breakdown shows the same overall trend as the main table: HachiWare preserves stronger accuracy than token-eviction and quantization baselines under aggressive stored-cache compression, while the refresh interval controls the accuracy–efficiency trade-off.

At the 8× setting, HachiWare with $I = 2$ reaches 82.66 average accuracy at 8.29× compression, outperforming all compressed baselines. The speed-oriented $I = 32$ variant reaches 78.34, remaining competitive with xKV-4 and Quest while using a compressed stored representation. At the 16× setting, HachiWare remains substantially stronger than the 16× eviction baselines: with $I = 32$, it reaches 71.01 average accuracy, compared with 58.21 for PyramidKV and 58.50 for SnapKV. The accuracy-oriented $I = 2$ variant further improves to 81.41, indicating that the same stored representation can recover additional accuracy when more frequent active-cache refresh is allowed.

The per-task results also show that Mistral-7B-Instruct-v0.3 is more sensitive to the refresh inter-

val than Llama-3.1-8B-Instruct. The gap between $I = 2$ and $I = 32$ is most visible on retrieval-heavy subtasks such as N-S3, N-MK2, and N-MQ. This supports the refresh-interval analysis in Appendix C.9: for Mistral, $I = 32$ provides a speed-oriented operating point, while $I = 2$ serves as an accuracy-oriented reference.

C.3 LongBench Results

Following the LongBench benchmark (Bai et al., 2024) and the evaluation setting of ShadowKV (Sun et al., 2025), we evaluate examples with sequence lengths exceeding 4K.

Table 8 broadens the evaluation beyond RULER-style synthetic retrieval. At 14.40× stored-cache compression, HachiWare achieves an average LongBench score of 52.19, within 2.71 points of the FullKV score of 54.90. Performance remains close to FullKV on NarrativeQA, HotpotQA, QM-Sum, TriviaQA, and PassageRetrieval-en, while larger gaps appear on GovReport and RepoBench-P. Overall, these results provide additional evidence that HachiWare remains effective across a broader range of long-context tasks.

Method	Comp.	NarrativeQA	HotpotQA	2WikiMQA	GovReport	QMSum	TriviaQA	PR-en	RepoBench-P	Avg.
FullKV	1.00×	30.33	58.14	45.23	34.14	25.20	92.37	100.00	53.81	54.90
HachiWare	14.40×	30.01	56.30	41.98	27.02	24.31	92.56	99.50	45.87	52.19

Notes. PR-en denotes PassageRetrieval-en. The evaluated tasks cover long-document QA, multi-hop QA, summarization, retrieval, and code.

Table 8: LongBench results on Llama-3.1-8B-Instruct for examples with sequence lengths exceeding 4K, following the evaluation setting of ShadowKV (Sun et al., 2025). **Comp.** denotes representation-level compression of the stored KV cache. The average is computed over the eight evaluated tasks.

	8× configuration		16× configuration	
	Comp.	Avg.	Comp.	Avg.
SCB heads				
24	8.48×	89.63	15.92×	89.63
32	8.55×	90.12	16.04×	88.99
40	8.63×	90.00	16.15×	90.80

Table 9: SCB routing-budget sensitivity on RULER with Llama-3.1-8B-Instruct at 64K context length. All other HachiWare settings are held fixed within each compression regime.

Anchor-selection policy	RULER Avg.
Default high-score anchor	88.99
Lower-score SCB anchor	87.45

Table 10: Anchor-selection sensitivity on RULER with Llama-3.1-8B-Instruct at 64K under the 16× configuration.

C.4 SCB Routing-Budget Sensitivity

Table 9 evaluates sensitivity to the number of KV heads routed to the SCB on Llama-3.1-8B-Instruct at 64K context length. We vary the SCB budget from 24 to 40 heads while keeping the SVD ranks, active-token budget, refresh interval, and anchor set fixed. Across both compression regimes, accuracy remains stable and does not exhibit a sharp collapse around the default 32-head setting. These results indicate that HachiWare does not rely on a precisely tuned SCB-head cutoff, although the routing budget remains an operating parameter that affects the accuracy–compression trade-off.

C.5 Anchor-Selection Sensitivity

Table 10 evaluates sensitivity to the anchor-selection policy on Llama-3.1-8B-Instruct at 64K under the 16× configuration. We replace the default high-score anchor with a lower-score SCB anchor while keeping the cache representation, active-token budget, compression setting, and refresh interval unchanged. The RULER average decreases from 88.99 to 87.45, a 1.54-point change. This result indicates that the shared ordered list

Method	Comp.	RULER Avg.
FullKV	1.00×	92.30
HachiWare	8.55×	90.12
HachiWare-Block	8.55×	90.55

Table 11: Block-granular selection on RULER with Llama-3.1-8B-Instruct at 64K. The block-granular variant uses contiguous blocks of 8 tokens.

is not highly sensitive to the exact anchor choice, although the default high-score anchor remains preferable.

C.6 Block-Granular Selection

To examine whether HachiWare can operate with block-aligned selection units, we replace token-wise eviction and active-cache selection with contiguous blocks of 8 tokens while keeping the overall compression setting unchanged. Table 11 shows that the block-granular variant achieves 90.55 average RULER accuracy at the same 8.55× stored-cache compression, compared with 90.12 for the token-wise setting. We do not interpret the small difference as a meaningful accuracy improvement; rather, the result shows that block-aligned selection can be used without accuracy loss in this setting.

C.7 Supplementary Results on Qwen2.5 Models

Table 12 reports supplementary RULER results on Qwen2.5-7B-Instruct-1M (Yang et al., 2025) at 64K context length, while Table 13 provides a larger-model check on Qwen2.5-14B-Instruct (Yang et al., 2024a) at the same context length. Qwen2.5-7B-Instruct-1M differs from the two main backbones in its KV-head configuration: it uses 28 layers with 4 KV heads per layer, resulting in a smaller total KV-head pool than Llama-3.1-8B-Instruct and Mistral-7B-Instruct-v0.3. For SCB routing, we use 16 heads for Qwen2.5-7B-Instruct-1M and 18 heads for Qwen2.5-14B-Instruct, reflecting their different model configurations. Following the setup in Section 4.1, we compare only with

Method	Comp.	N-S1	N-S2	N-S3	N-MK1	N-MK2	N-MQ	N-MV	QA-1	QA-2	VT	FWE	Avg.
FullKV	1.00	100.00	100.00	100.00	100.00	100.00	100.00	96.09	83.33	60.41	90.20	87.84	92.53
KIVI-2	7.10	2.08	6.25	0.00	10.41	11.45	0.52	1.56	44.79	44.79	30.62	38.88	17.40
xKV-4	6.62	100.00	94.79	46.87	85.41	98.95	85.41	76.82	64.58	54.16	66.04	68.40	76.49
PyramidKV	7.00	100.00	92.70	20.83	96.87	26.04	70.57	70.83	83.33	54.16	91.25	79.16	71.43
SnapKV	7.00	100.00	96.87	60.41	97.91	45.83	87.76	78.90	83.33	55.20	92.08	78.86	79.74
HachiWare_{I=32}	7.06	100.00	100.00	96.87	100.00	78.12	98.17	92.70	83.33	59.37	91.66	76.38	88.78

Notes. N-S, N-MK, N-MQ, and N-MV denote Needle single-key, multi-key, multi-query, and multi-value tasks. I denotes the active-cache refresh interval used by HachiWare.

Table 12: Supplementary RULER results on Qwen2.5-7B-Instruct-1M at 64K context length. **Comp.** denotes representation-level compression of the stored KV cache.

Method	Comp.	Avg.
FullKV	1.00×	85.68
SnapKV	7.00×	77.96
xKV-4	6.88×	78.07
HachiWare	7.12×	80.70

Table 13: Average RULER results on Qwen2.5-14B-Instruct at 64K context length. **Comp.** denotes representation-level compression of the stored KV cache.

baselines whose reported compression corresponds to a reduced stored KV cache representation, excluding selection-only and offloading-based baselines such as Quest and ShadowKV.

HachiWare remains effective on Qwen2.5-7B-Instruct-1M. At $7.06\times$ stored-cache compression with refresh interval $I = 32$, it reaches 88.78 average RULER accuracy, outperforming KIVI-2, xKV-4, PyramidKV, and SnapKV under comparable stored-cache compression. The improvement is especially clear on retrieval-heavy subtasks, where HachiWare preserves high accuracy across several NIAH variants while the low-rank and eviction baselines show larger degradation.

On Qwen2.5-14B-Instruct, HachiWare achieves 80.70 average RULER accuracy at $7.12\times$ stored-cache compression, outperforming SnapKV and xKV-4 by 2.74 and 2.63 points, respectively, at comparable compression.

Together, the Qwen results extend the evaluation beyond the main Llama and Mistral backbones, covering both a different KV-head configuration and a larger 14B-scale model.

C.8 Effect of One-Shot Selective SVD

Table 14 compares the default low-rank SCB with an exact-SCB variant that stores semantic-critical states in full precision under the same SCB routing, EB retention budget, active selection budget, and runtime refresh procedure. One-shot selective

κ	SCB repr.	SVD Comp.	Comp.	Δ Comp.	Mean	Δ Mean
2048	Full	—	6.56	—	89.35	—
2048	Low-rank	3.57	16.04	+9.48	88.99	−0.36
2048	Low-rank	8.03	23.31	+16.75	87.60	−1.75
1024	Full	—	7.21	—	88.38	—
1024	Low-rank	3.57	20.54	+13.33	88.47	+0.09
1024	Low-rank	8.03	34.20	+26.99	85.46	−2.92

Table 14: Ablation of one-shot selective SVD on RULER with Llama-3.1-8B-Instruct at 64K context length. Full keeps semantic-critical states in full precision, while Low-rank applies one-shot selective SVD with the reported SVD compression ratio. Δ values are computed relative to Full under the same active budget κ .

SVD substantially increases compression with limited accuracy loss. At $\kappa = 2048$, compression improves from $6.56\times$ to $16.04\times$, while the RULER mean drops by only 0.36 points. At $\kappa = 1024$, the $34.20\times$ setting incurs a larger 2.92-point drop, showing that the low-rank SCB is the main mechanism that moves HachiWare beyond the compression limit of exact semantic-critical storage.

C.9 Refresh-Interval Sensitivity

The refresh interval I controls how often the shared ordered list and active cache are updated. Table 15 shows model-dependent sensitivity. Llama-3.1-8B-Instruct remains stable even at $I = 32$, reaching 90.12 RULER mean, comparable to more frequent refresh settings. Mistral-7B-Instruct-v0.3 is more sensitive, decreasing from 82.66 at $I = 2$ to 78.34 at $I = 32$, but the $I = 32$ setting remains competitive with the main-table baselines. This explains the two refresh regimes used in the main results.

C.10 Additional Efficiency Results

C.10.1 Prefill-Stage Latency

Table 16 reports full prefill-stage latency scaling on A6000. Total latency measures the time to produce a ready-to-decode cache, including the model

Model	$I = 1$	$I = 2$	$I = 4$	$I = 8$	$I = 16$	$I = 32^\dagger$
Llama, 64K	89.63	89.62	90.07	90.38	89.86	90.12
Mistral, 32K	82.84	82.66	81.16	79.45	77.61	78.34

Table 15: Effect of the active-cache refresh interval I on RULER Mean. Llama denotes Llama-3.1-8B-Instruct at 64K context length with $\kappa = 4096$ and Comp. = $8.55\times$; Mistral denotes Mistral-7B-Instruct-v0.3 at 32K context length with $\kappa = 2048$ and Comp. = $8.29\times$. $I = 1$ refreshes at every decoding step, and † denotes the speed-oriented setting used for efficiency evaluation.

prefill pass and method-specific cache-preparation work. The lower block reports the fraction of total prefill-side latency spent on SVD.

For implementation, ShadowKV is evaluated with its offloading path enabled, reflecting its GPU-memory-saving setting, and uses a full SVD routine for cache preparation. In contrast, xKV and HachiWare use `torch.svd_lowrank` for low-rank cache factorization. This routine computes a randomized approximate low-rank SVD rather than a full SVD, trading exact singular-vector computation for lower cache-preparation overhead in our long-context setting.

Thus, the table reflects the prefill-stage latency of the evaluated implementations, including their different SVD routines and memory-management paths. The overhead is most visible at shorter context lengths: at 8K, ShadowKV reaches $7.87\times$ FullKV latency and xKV reaches $4.45\times$, whereas HachiWare remains at $1.83\times$. Across the full 8K–64K range, HachiWare stays below $2\times$ FullKV latency, indicating that one-shot selective SVD keeps prefill-stage overhead controlled.

C.10.2 Decode-Time Materialization Cost

Table 17 isolates decode-time active-token selection and low-rank reconstruction costs on Llama-3.1-8B-Instruct at 64K with $\kappa = 2048$. We use xKV-SR, the xKV variant with ShadowKV-style per-layer selection, as a common-path diagnostic for selective low-rank reconstruction. This comparison is intended to isolate materialization cost rather than provide an optimized end-to-end runtime comparison.

Even at $I = 1$, before refresh-interval amortization, HachiWare reduces both selection and reconstruction cost relative to xKV-SR. With $I = 32$, these costs are further amortized across decoding steps. This shows that the decoding efficiency comes from a smaller selective-reconstruction scope together with refresh amortization, rather

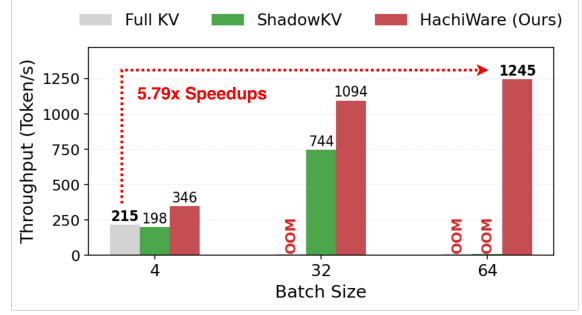


Figure 10: Additional decoding-throughput results at 64K context length on a single NVIDIA H100 80GB GPU. HachiWare uses the speed-oriented $16\times$, $I = 32$ setting. Methods with runtime selection or selective reconstruction use the same active budget $\kappa = 2048$ where applicable. Higher is better; OOM denotes out-of-memory settings.

than from the refresh interval alone.

C.10.3 Decode Throughput under Refresh Intervals

Table 18 reports decoding throughput under different active-cache refresh intervals on a single NVIDIA A6000 48GB GPU. This experiment follows the 64K-context decoding setup in Section 4.5. Each method is evaluated at its largest feasible batch size under the memory limit: FullKV supports batch size 2, while HachiWare supports batch size 32. For HachiWare, increasing I refreshes the active cache less frequently, amortizing ordered-list construction, selection, and SCB reconstruction over more decoding steps. The results show a consistent throughput increase as the refresh interval becomes larger, indicating that refresh overhead is effectively amortized during decoding. This supports using $I = 32$ as the speed-oriented setting in the main efficiency study.

C.10.4 Additional H100 Decode Throughput

Figure 10 reports additional decoding-throughput results on Llama-3.1-8B-Instruct at 64K context length using a single NVIDIA H100 80GB GPU. The setting follows Section 4.5: HachiWare uses the speed-oriented $16\times$, $I = 32$ configuration, and methods with runtime selection or selective reconstruction use the same active budget $\kappa = 2048$ where applicable.

The H100 results show the same trend as the A6000 results in Figure 7. HachiWare reaches 1245 tokens/s at batch size 64, compared with 215 tokens/s for the largest feasible FullKV setting, yielding a $5.79\times$ throughput improvement. This

Method	8K	16K	24K	32K	40K	48K	56K	64K
Total latency seconds, relative to FullKV in parentheses								
FullKV	1.20 (1.00×)	2.74 (1.00×)	4.82 (1.00×)	7.73 (1.00×)	10.24 (1.00×)	13.43 (1.00×)	17.23 (1.00×)	21.19 (1.00×)
xKV	5.34 (4.45×)	7.38 (2.69×)	9.58 (1.99×)	12.10 (1.57×)	15.73 (1.54×)	18.99 (1.41×)	23.05 (1.34×)	27.65 (1.30×)
ShadowKV	9.44 (7.87×)	11.22 (4.09×)	13.95 (2.89×)	17.01 (2.20×)	28.29 (2.76×)	36.37 (2.71×)	46.17 (2.68×)	50.47 (2.38×)
HachiWare	2.20 (1.83×)	3.47 (1.27×)	5.95 (1.23×)	9.40 (1.22×)	11.81 (1.15×)	15.02 (1.12×)	19.57 (1.14×)	23.68 (1.12×)
SVD time ratio percentage of total prefill-side latency								
FullKV	—	—	—	—	—	—	—	—
xKV	29.3%	29.4%	26.9%	23.8%	20.8%	20.7%	18.8%	17.2%
ShadowKV	31.8%	31.4%	28.0%	23.3%	15.1%	13.8%	11.3%	11.1%
HachiWare	17.5%	15.7%	11.0%	7.8%	7.1%	6.8%	5.6%	5.1%

Notes. Relative latency is computed against FullKV at the same context length. SVD time ratio is the fraction of total prefill-side latency spent on SVD.

Table 16: Full prefill-side latency scaling on A6000. The upper block reports total latency in seconds, with latency relative to FullKV in parentheses. The lower block reports the fraction of total latency spent on SVD.

Configuration	Selection	Reconstruction
HachiWare, $I = 1$	1.399	1.880
xKV-SR diagnostic	4.451	5.824
HachiWare, $I = 32$	0.044	0.062

Table 17: Decode-time materialization-cost diagnostic on Llama-3.1-8B-Instruct at 64K with $\kappa = 2048$. Times are reported in ms per decoding step; the $I = 32$ costs are amortized over the refresh interval.

Method	Batch Size	Refresh I	Tok/s
FullKV	2	—	43
HachiWare	32	1	80
HachiWare	32	2	136
HachiWare	32	4	200
HachiWare	32	8	252
HachiWare	32	16	323
HachiWare	32	32	406

Notes. Throughput is measured in generated tokens per second. FullKV is shown as the largest feasible uncompressed-cache reference. All HachiWare rows use the same stored-cache configuration and differ only in the active-cache refresh interval.

Table 18: Decoding throughput under different refresh intervals on A6000 48GB at 64K context length. Each row uses the largest feasible batch size under the memory limit.

confirms that the decoding-speed gain is not specific to A6000 and remains effective on a higher-bandwidth GPU.

C.11 Integration with Quantization

HachiWare can be combined with orthogonal cache-compression techniques such as quantization. As a preliminary compatibility study, we apply 4-bit round-to-nearest (RTN) quantization on top of the stored HachiWare cache representation after

prefilling. The quantized representation includes both the SCB low-rank factors and the retained EB KV states; the routing, active-cache refresh, and refresh interval are kept unchanged. All variants use $I = 32$, and the quantization group size is 32.

Table 19 shows that quantization further increases the stored-cache compression ratio with limited additional accuracy loss. For example, the $8.55\times$ HachiWare setting becomes $27.36\times$ with 4-bit quantization while preserving a RULER average of 89.87, close to the unquantized 90.12. The more aggressive $16.04\times$ and $34.20\times$ settings further increase to $51.33\times$ and $109.44\times$, respectively, showing that HachiWare remains compatible with low-precision cache storage.

D Additional Related Work

KV cache reduction. KV cache compression has been studied through eviction, quantization, and runtime cache selection. Eviction-based methods reduce memory by retaining only a subset of token states. H₂O (Zhang et al., 2023) keeps heavy-hitter and recent tokens, Scissorhands (Liu et al., 2023) exploits persistence in token importance, and StreamingLLM (Xiao et al., 2024) maintains attention sinks and recent tokens for streaming generation. More recent methods improve selection or allocation policies: SnapKV (Li et al., 2024) selects prompt-side important tokens using attention statistics, PyramidKV (Cai et al., 2025) allocates budgets according to layer-wise information funneling, CAKE (Qin et al., 2025) uses cascading and adaptive eviction with layer preferences, D²O (Wan et al., 2025) applies dynamic discriminative operations, and Ada-KV (Feng et al., 2025) introduces adaptive head-wise budget allocation. These meth-

Method	Comp.	N-S1	N-S2	N-S3	N-MK1	N-MK2	N-MQ	N-MV	QA-1	QA-2	VT	FWE	Avg.
FullKV	1.00	100.00	100.00	100.00	100.00	91.66	100.00	98.69	82.29	60.41	97.91	84.37	92.30
HachiWare	8.55	100.00	100.00	100.00	100.00	91.66	99.73	96.61	83.33	58.33	92.91	68.75	90.12
HachiWare-4bit	27.36	100.00	100.00	100.00	98.95	91.66	99.73	96.61	81.25	60.41	91.25	68.75	89.87
HachiWare	16.04	100.00	100.00	93.75	100.00	88.54	99.47	97.91	81.25	58.33	87.08	72.56	88.99
HachiWare-4bit	51.33	100.00	100.00	94.79	98.95	87.50	99.21	93.48	80.20	56.25	84.37	70.13	87.72
HachiWare	34.20	100.00	97.91	83.33	95.83	86.45	97.65	95.57	82.29	58.33	74.99	67.70	85.46
HachiWare-4bit	109.44	100.00	100.00	77.08	92.70	79.16	96.87	91.92	81.25	57.29	68.54	66.66	82.86

Notes. N-S, N-MK, N-MQ, and N-MV denote Needle single-key, multi-key, multi-query, and multi-value tasks. HachiWare-4bit applies 4-bit non-symmetric quantization with group size 32 on top of the corresponding HachiWare configuration. All HachiWare variants use refresh interval $I = 32$.

Table 19: Integration of HachiWare with 4-bit KV cache quantization on Llama-3.1-8B-Instruct at 64K context length. **Comp.** denotes representation-level compression of the stored KV cache after optional quantization.

ods are efficient because removed states no longer consume memory or attention computation, but the discarded KV states are not explicitly reconstructable.

Quantization and mixed-precision methods reduce the representation cost of retained cache states. KIVI (Liu et al., 2024b) uses asymmetric low-bit KV cache quantization, KVQuant (Hooper et al., 2024) targets ultra-low-precision KV cache quantization for very long contexts, GEAR (Kang et al., 2024) combines quantization with low-rank and sparse correction, and No Token Left Behind (Yang et al., 2024b) retains evicted information in lower precision. These methods preserve approximate cache information, but they do not explicitly allocate a more expressive reconstructable representation to semantically critical heads.

Runtime selection methods reduce the active attention view during decoding. QUEST (Tang et al., 2024) performs query-aware sparse KV-page selection, while RocketKV (Behnam et al., 2025) combines coarse-grained cache reduction with fine-grained sparse access for long-context inference. These approaches are complementary to storage compression: they control which cache entries are accessed at a decoding step, whereas HachiWare first decides which KV heads should receive reconstructable low-rank storage and then applies bounded active-cache refresh over the resulting branches.

Low-rank and transform-based KV cache storage. Another line of work studies compressed KV representations that can be approximately reconstructed or consumed by modified attention computation. Eigen Attention (Saxena et al., 2024) performs attention in a compressed low-rank space. Palu (Chang et al., 2025) stores

low-rank latent KV representations through static projection decompositions. ShadowKV (Sun et al., 2025) combines low-rank key storage with sparse runtime reconstruction and value offloading. MatryoshkaKV (Lin et al., 2025a) learns adaptive orthogonal projections for KV compression. xKV (Chang et al., 2026) applies cross-layer SVD to grouped KV caches, exploiting redundancy across adjacent layers. KVTC (Staniszewski and Łańcucki, 2026) uses transform coding for compact reusable KV cache storage.

These methods show that KV caches contain substantial redundancy and can be represented in lower-dimensional or coded forms. Their primary focus is how to compress a selected cache representation. In contrast, HachiWare focuses on where reconstructability should be allocated: semantic-critical KV heads are routed to a low-rank Semantic-Critical Branch, while the remaining heads are handled by a cheaper Eviction Branch. This makes low-rank reconstruction selective rather than uniformly applied across the cache.

Head-aware cache policies and retrieval heads.

Recent studies show that attention heads are functionally heterogeneous. Retrieval Head (Wu et al., 2025) identifies a small subset of heads that play a disproportionate role in long-context factual retrieval. This observation motivates head-aware cache policies. DuoAttention (Xiao et al., 2025) separates retrieval and streaming heads and assigns different cache policies to them. RazorAttention (Tang et al., 2025) preserves full caches for retrieval heads while compressing non-retrieval heads. HeadKV (Fu et al., 2025) allocates KV cache budgets according to head-level importance.

CompressKV (Lin et al., 2025b) is the closest semantic-head prior to our work. It identifies Se-

mantic Retrieval Heads (SRHs) by aggregating attention mass over the reference answer span during answer generation, and uses them to guide token retention and layer-adaptive eviction. HachiWare uses this semantic retrieval behavior for a different purpose. Rather than using SRHs only to select token positions under an eviction framework, it aggregates semantic retrieval scores into KV-head scores and uses them to route heads between heterogeneous storage branches. Thus, HachiWare combines irreversible eviction and reconstructable low-rank storage at the granularity of semantically critical KV heads.

E Use of AI Assistants

AI assistants were used for language polishing and checklist-preparation assistance. All technical content, experimental results, claims, and final text were reviewed and verified by the authors. AI assistants were not used to generate experimental results or make authorship decisions.