SAGE: Shortlist-Aware Guided Encoding for KV-Cache Compression

Anonymous ACL submission

Abstract

Long-context LLM serving needs KV-cache compression that improves latency and cache footprint while remaining compatible with existing attention stacks. Sparse-dictionary key compression is attractive because it can be applied to all retained keys or composed after token eviction, but its full-dictionary encoder can be slower than the long-context forward pass. We introduce SAGE (Shortlist-Aware Guided Encoding), a drop-in key-cache compressor that uses a per-layer-head MLP to predict a high-recall dictionary shortlist before sparse coding. The MLP is distilled from cached sparse-coding supports, so the learned stage narrows atom search while the constrained sparse-coding stage still solves the final least-squares coefficients. On two open-weight 8B LMs, SAGE preserves the full-search sparse-coding baseline's quality on GSM8K, LongBench, and RULER while substantially reducing compression and end-to-end latency. When composed with a learned KV-eviction pipeline, paired bootstrap and fixed-margin tests support quality parity with the full-search sparse-coding baseline. The same sparse codes also provide an off-cache sparse-storage path for reducing key-cache footprint, while realized VRAM savings require a sparse-storage or fused reconstruction-attention implementation.

1 Introduction

Long-context LLM serving is increasingly constrained by the KV cache. Eviction reduces the number of cached tokens, quantization reduces scalar precision, and representation-change methods compress each retained vector. Sparse-dictionary key coding is attractive in this last category because it can be used on all retained keys or after an existing eviction policy has already selected which tokens to keep.

The obstacle is the encoding cost of sparse-dictionary coding. Sparse codes can be produced by several pursuit-style solvers; the prior KV-cache compressor uses orthogonal matching pursuit (OMP), a simple standard instantiation that builds each code through repeated full-dictionary atom search. The compressed representation can therefore be cheap to store but expensive to produce. In long-context requests, this sparse-representation search can turn cache compression from a memory-saving stage into a latency bottleneck. This paper asks whether sparse-dictionary key compression can keep its reconstruction quality while avoiding the full search.

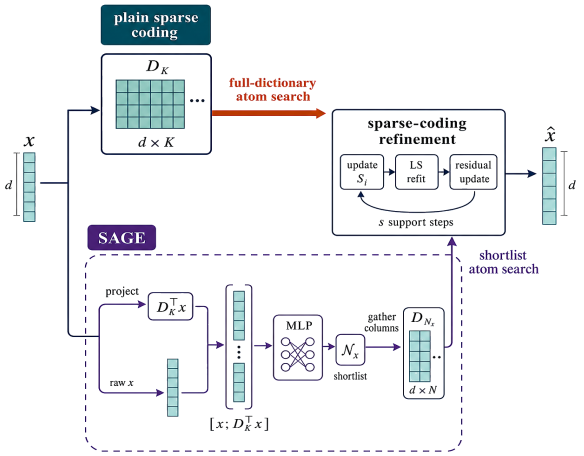

Figure 1: **SAGE makes sparse-dictionary cache compression deployable by guiding sparse coding with a learned shortlist.** It predicts a high-recall shortlist and runs the same sparse-coding solver on fewer candidates.

We introduce **SAGE** (Shortlist-Aware Guided Encoding), a drop-in encoder for sparse-dictionary key caches. SAGE learns to predict a small high-recall shortlist of candidate atoms, then runs the same sparse-coding refit on that shortlist. The learned stage therefore does not have to solve sparse reconstruction by itself. It only has to keep useful atoms available, while the classical solver still chooses the support and coefficients. Figure 1

illustrates this separation between learned candidate generation and sparse reconstruction.

This design keeps the deployment interface narrow. SAGE operates on retained key vectors, so it can be applied as a standalone key-cache compressor or composed after a token-eviction policy. It also leaves the attention path conceptually unchanged: the encoder changes how retained keys are represented, not which tokens the upstream policy keeps or how the model scores attention once keys are reconstructed.

On two representative open-weight 8B LMs, SAGE matches the full-search sparse-coding baseline within 0.5 score points on GSM8K, LongBench, and RULER headline cells while reducing compression time by 3–6$\times$. At long-context operating points, this yields up to a $4.4\times$ end-to-end speed-up for the prefill, compression, and short decode pipeline. When inserted behind a fixed KV-eviction mask, SAGE preserves measured quality parity with the full-search baseline while reducing compression time.

This paper makes three contributions:

- We identify full-dictionary sparse-coding search as a practical bottleneck in sparse-dictionary KV-cache compression, using a standard full-search pursuit routine as the sparse-encoder instantiation.
- We introduce SAGE, an MLP-guided sparse-coding encoder that predicts high-recall dictionary shortlists while retaining the solver's least-squares reconstruction step.
- We evaluate the method across quality, latency, storage accounting, ablations, and composition with KV eviction.

2 Related Work

KV-cache eviction and selection. Token-selection methods reduce the time axis of the cache by retaining positions, layers, or heads estimated to matter, using attention sinks, recent windows, attention mass, prompt statistics, query-aware retrieval, or learned budgets (Xiao et al., 2024; Zhang et al., 2023; Liu et al., 2023; Ge et al., 2024; Li et al., 2024; Cai et al., 2024; Adnan et al., 2024; Tang et al., 2024; Feng et al., 2024; Devoto et al., 2024; Tang et al., 2025; Xiao et al., 2025; Wang et al., 2025; Kim et al., 2025a; Jegou and Jeblick, 2026). Feature- or depth-oriented variants such as SparQ and MiniCache reduce other cache axes (Ribar et al., 2024; Liu et al., 2024a). These methods are complementary to SAGE: eviction decides which tokens remain; our method changes how each retained key is encoded. The integration experiments therefore hold the keep-mask fixed and isolate the encoder replacement.

Quantization and basis-change compression. KV quantizers reduce per-scalar precision (Liu et al., 2024b; Hooper et al., 2024); GEAR combines quantization with low-rank residual correction (Kang et al., 2024), and ShadowKV stores a low-rank key cache with sparse recovery (Sun et al., 2025). These methods change scalar precision or the vector space used to store attention information. Sparse-dictionary KV compression instead stores each retained vector as a sparse combination of atoms. Kim et al. (2025b) train dictionaries on pre-RoPE cache vectors and encode each key with a full-search pursuit routine, which is our closest baseline. We keep its dictionary and reconstruction interface, but replace the repeated full-dictionary atom search with a learned shortlist. SAGE is therefore not a new eviction policy, value compressor, or attention kernel; it targets the encoder-search bottleneck inside one sparse-dictionary compressor and can be used alongside eviction or quantization.

Sparse coding and learned sparse encoders. Matching pursuit and OMP are classical greedy sparse-approximation methods (Mallat and Zhang, 1993; Pati et al., 1993), with recovery theory for random measurements (Tropp and Gilbert, 2007). K-SVD and online dictionary learning provide standard dictionary-training recipes (Aharon et al., 2006; Mairal et al., 2009). Learned sparse encoders such as LISTA and later deep sparse-support predictors approximate iterative solvers directly (Gregor and LeCun, 2010; Xin et al., 2016; Sreter and Giryes, 2018). SAGE uses a weaker learned target: it predicts a high-recall candidate set and leaves coefficient fitting to the constrained solver, rather than replacing the sparse-code output itself. If the shortlist contains the atoms that the full-search solver would select, the downstream least-squares refit recovers the same reconstruction; the method section formalizes this support-recall guarantee.

3 Method

Sparse-dictionary KV coding. Let $x \in \mathbb{R}^d$ be a retained pre-RoPE key vector and let $D_K \in \mathbb{R}^{d \times K}$ be the frozen per-module key dictionary follow-

ing Kim et al. (2025b). Our implementation compresses keys and leaves values dense. Sparse coding targets the ideal ℓ_0-constrained dictionary approximation

$$E_K^\star(x) = \min_{\|c\|_0 \le s} \|x - D_K c\|_2^2. \tag{1}$$

The nonzero indices form the sparse support and the values are the stored coefficients. This nonconvex objective is approximated in practice by greedy pursuit. Following the prior sparse-dictionary KV compressor, our full-search baseline uses the same standard pursuit routine. The method targets the full-dictionary atom-search bottleneck rather than changing the sparse-representation objective or the least-squares refit. Unless stated otherwise, the dictionary columns $d_j = D_{K,:,j}$ are ℓ_2-normalized. At step i, the pursuit routine scores atoms by $|d_j^\top r_i|$ (or by $|d_j^\top r_i|/\|d_j\|_2$ for unnormalized columns), adds the best atom, and re-fits least-squares coefficients on the selected support. The small LS system is cheap for $s \le 32$; repeated K-atom residual search is the bottleneck.

SAGE changes the candidate set, not the baseline sparse-coding routine. Let $\mathcal{S}_s^{\text{fs}}(x)$ be the ordered support returned by the full-search encoder after s steps, and let $E_K^{\text{fs}}(x)$ be its achieved reconstruction error. Given a predicted shortlist $\mathcal{N}_x$, let $E_{\mathcal{N}}^{\text{fs}}(x)$ be the error obtained by running the same pursuit routine with atom search restricted to $\mathcal{N}_x$. We define the baseline-relative shortlist regret as

$$\Delta_{\mathcal{N}}^{\text{fs}}(x) = E_{\mathcal{N}}^{\text{fs}}(x) - E_K^{\text{fs}}(x). \tag{2}$$

This regret is relative to the full-search sparse-coding baseline rather than to the global optimum of Eq. (1). Coherent dictionaries often have alternate supports with nearly identical error, so the learned stage is a high-recall retrieval problem, not a direct coefficient solver.

We use the same frozen pre-RoPE dictionary for the full-search baseline and SAGE, so score differences come from shortlisting rather than from a different basis. RoPE is applied after reconstruction. Let $e_t = x_t - \hat{x}_t$ be the pre-RoPE key reconstruction error at key position t, and let R_u and R_t be the orthogonal RoPE rotations at query position u and key position t. The attention-logit perturbation is

$$\delta\ell_{u,t} = \frac{(R_u q_u)^\top R_t e_t}{\sqrt{d}} = \frac{q_u^\top R_u^\top R_t e_t}{\sqrt{d}}, \tag{3}$$

$$|\delta\ell_{u,t}| \le \frac{\|q_u\|_2 \|e_t\|_2}{\sqrt{d}}. \tag{4}$$

The bound follows because $R_u^\top R_t$ is orthogonal under standard RoPE. Thus reconstruction loss controls a worst-case key-side logit perturbation up to the query norm; we use it as a proxy, not a task-quality guarantee.

SAGE. For each layer-head key module $m \in \{1, \ldots, LH_{\text{kv}}\}$, SAGE trains an MLP $f_m : \mathbb{R}^{d+K} \to \mathbb{R}^K$ with two GELU hidden layers of width 1024. The input is $[x; D_{K,m}^\top x]$, which exposes both the raw vector and its dictionary correlations. This per-layer-head encoder bank is fixed serving state rather than per-token cache state; at our headline width it is therefore a substantial deployment overhead, quantified in Appendix A. For compactness, write $z_m(x) = f_m([x; D_{K,m}^\top x])$. At inference, $z_m(x)$ produces K scores; we keep the N largest absolute-score indices as $\mathcal{N}_x \subset \{1, \ldots, K\}$ and gather $D_{K,m}^{(\mathcal{N}_x)}$. The sparse-coding routine then runs for s iterations only on this sub-dictionary. The output is still an original-K support, coefficients, and reconstruction $\hat{x} = D_K \hat{c}$, which we write back into the dense key cache. Thus downstream attention sees the same cache shape as in the full-search sparse-coding prototype.

Figure 2 separates the lifecycle into offline distillation, prefill-time sparse-key encoding, and key restoration before attention. For each retained key x_t, SAGE computes $p_t = D_K^\top x_t$, scores atoms with $z_m(x_t)$, keeps the largest absolute scores, and runs the same pursuit routine inside the gathered sub-dictionary $D_{\mathcal{N}_t}$. The encoded record is (S_t, c_t), and restoration forms $\hat{x}_t = D_{K,S_t} c_t$ before the unchanged attention interface. Our latency prototype performs this materialization immediately and writes dense keys back to the ordinary cache; a sparse-storage path can instead keep (S_t, c_t) until reconstruction is needed.

The shortlist is larger than the target support: headline settings use $N \in \{128, 192\}$ and $s \in \{24, 32\}$, with additional sweeps at $N = 64$ and $s = 16$. This overcomplete candidate set matches the coherent-dictionary regime: false positives mostly add a small search cost, whereas false negatives can change reconstruction quality because a missing atom cannot be selected downstream.

Support-recall guarantee. The formal requirement on the shortlist is recall rather than exact ranking. Assume both sparse-coding routines use the same globally defined deterministic tie-breaking rule over original atom indices. Let $\mathcal{S}_i^{\text{fs}}(x)$ be the

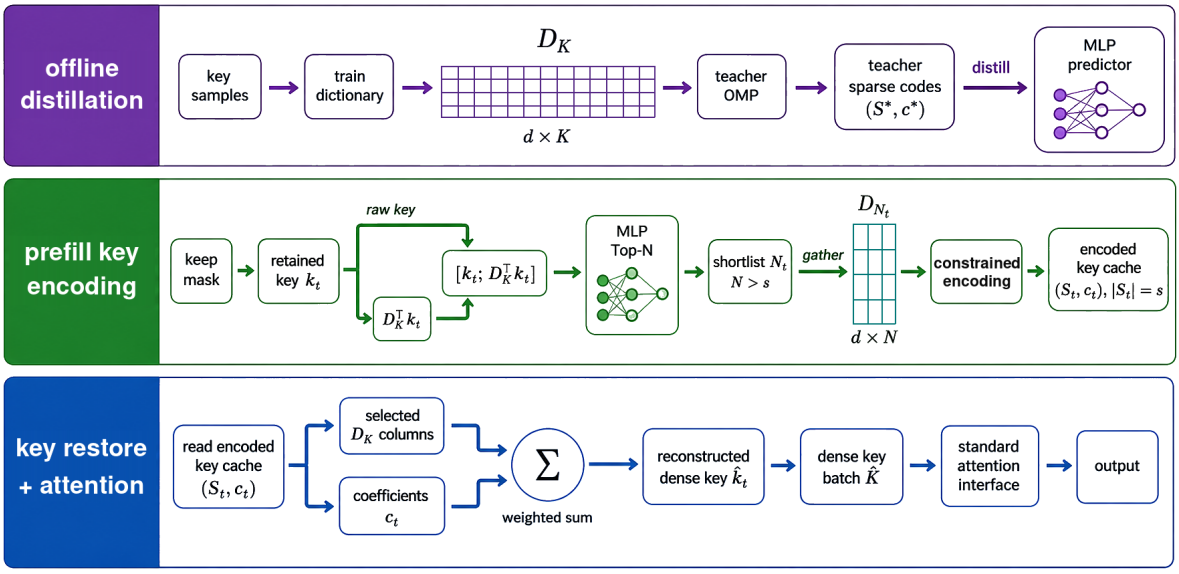

Figure 2: **SAGE lifecycle.** Offline sparse-coding supports supervise the shortlist predictor. During prefill, retained keys are encoded as sparse records through MLP shortlisting and constrained encoding. The restore lane shows how the restored key cache is consumed by the standard attention interface; values remain dense and are not shown.

ordered support selected by the full-search routine after i greedy steps, and let $\widetilde{\mathcal{S}}_i(x)$ be the support selected by the same pursuit routine restricted to $\mathcal{N}_x$. If $\mathcal{S}_s^{\mathrm{fs}}(x) \subseteq \mathcal{N}_x$, then $\widetilde{\mathcal{S}}_i(x) = \mathcal{S}_i^{\mathrm{fs}}(x)$ for every $i \leq s$. Consequently, the final least-squares problem is identical, so the reconstruction is identical to the full-search baseline, the coefficients are identical under the same least-squares convention, and $\Delta_{\mathcal{N}}^{\mathrm{fs}}(x) = 0$.

The proof is by induction. If both encoders have the same support before step i, their LS refits and residuals are the same. The full-search atom is inside $\mathcal{N}_x$ by assumption and no other atom has larger correlation, so the shortlisted routine chooses the same atom under the same tie-breaking rule. The supports and final LS problem therefore match. This does not prove that the MLP always recovers the support; it states the condition under which recall is sufficient to reproduce the full-search baseline and false positives are mostly a latency cost.

The same view gives a softer condition when the exact full-search atom is not present. Let $\widetilde{r}_i = x - P_{\widetilde{\mathcal{S}}_i} x$ be the current residual of the shortlisted routine, where $P_{\widetilde{\mathcal{S}}_i}$ projects onto the span of the selected atoms. For normalized atoms, define the residual-correlation recall at step i as

$$\rho_i(x) = \frac{\max_{j \in \mathcal{N}_x \setminus \widetilde{\mathcal{S}}_i} |\langle d_j, \widetilde{r}_i \rangle|}{\max_{j \notin \widetilde{\mathcal{S}}_i} |\langle d_j, \widetilde{r}_i \rangle|}. \tag{5}$$

The constrained step then decreases residual energy by at least

$$\|\widetilde{r}_i\|_2^2 - \|\widetilde{r}_{i+1}\|_2^2 \geq \rho_i(x)^2 \left(\max_{j \notin \widetilde{\mathcal{S}}_i} |\langle d_j, \widetilde{r}_i \rangle| \right)^2. \tag{6}$$

For unnormalized dictionaries, the same statement uses normalized correlations $|\langle d_j, \widetilde{r}_i \rangle| / \|d_j\|_2$. The bound formalizes the weaker requirement that the shortlist preserve most of the correlation mass driving the shortlisted routine's residual reduction.

Distillation objective. Directly predicting exact sparse-coding coefficients is brittle because $K \gg d$ dictionaries are coherent: many supports reconstruct nearly the same vector. We instead run the full-search sparse encoder once on Wikipedia key samples and cache $(\mathrm{idx}_{\mathrm{fs}}, \mathrm{scl}_{\mathrm{fs}})$ labels at $s_{\mathrm{teacher}} = 32$. These labels define a dense target $c^\star$: entries in the teacher support receive their sparse-code coefficients and all other entries are zero. The MLP is trained with a weighted MSE on this target, using $w_k = \alpha$ on selected teacher atoms and $w_k = 1$ elsewhere, plus a reconstruction regularizer through D_K:

$$\mathcal{L}_{\mathrm{sel}} = \sum_k w_k \left(z_m(x)_k - c_k^\star \right)^2, \tag{7}$$

$$\mathcal{L}_{\mathrm{rec}} = \|D_K \, \mathrm{TopK}_s(z_m(x)) - x\|_2^2, \tag{8}$$

$$\mathcal{L} = \mathcal{L}_{\mathrm{sel}} + \lambda_{\mathrm{rec}} \mathcal{L}_{\mathrm{rec}}. \tag{9}$$

Here TopK_s keeps the s largest-magnitude predicted coefficients and zeros the rest, and λ_{rec} con-

trols the reconstruction penalty. We do not claim to differentiate through the discrete support decision itself: the rank boundary is treated as a selection operation, gradients from $\mathcal{L}_{\text{rec}}$ flow to the selected predicted coefficient values, and the dense weighted-MSE term supplies the main support-recall pressure. This matches the inference use of the MLP as a shortlist retriever rather than as the final sparse solver. Optimizer settings and ablation summaries are in Appendix A and Appendix B.

The coefficient target is also a sufficient surrogate for support retrieval. Let $S^\star = \text{supp}(c^\star)$ and $\beta = \min_{j \in S^\star} |c_j^\star|$. If $N \geq |S^\star|$ and a predictor satisfies $\|z_m(x) - c^\star\|_\infty < \beta/2$, then every support entry has predicted magnitude strictly above $\beta/2$ and every non-support entry has predicted magnitude strictly below $\beta/2$. Thus every teacher atom is ranked above every non-teacher atom, and $S^\star \subseteq \text{TopN}(|z_m(x)|)$. The weighted MSE does not guarantee this uniformly, but it penalizes violations of a recall-sufficient condition. The reconstruction term prevents index-space matches that reconstruct poorly under D_K, and teacher labels can be reused for any inference shortlist size N.

Cost and composition. Here d is the key-head dimension, K is the dictionary size, s is the sparse-code budget, N is the shortlist size, and h is the MLP hidden width. Let $T_{\text{LS}}(s, d)$ collect the small least-squares updates. The per-vector encoder costs are

$$T_{\text{full}} = O(sKd) + T_{\text{LS}}(s, d), \tag{10}$$

$$T_{\text{SAGE}} = O(Kd) + O(h(d + K) + h^2 + hK) + O(sNd) + T_{\text{LS}}(s, d). \tag{11}$$

SAGE therefore does not remove all K-dependent work: it still computes the one-time projection $D_K^\top x$ consumed by the predictor. It replaces the repeated full-dictionary residual rescoring inside sparse coding with shortlisted rescoring. With $K = 4096$, $N = 128$ or 192, $s = 24$, and $d = 128$, the iterative residual-search term alone shrinks by $K/N = 21$–$32\times$ before projection and MLP overhead. If one counts the projection together with residual dot products but omits MLP and LS terms, the ideal dot-product ratio is

$$\frac{sKd}{Kd + sNd} = \frac{sK}{K + sN}. \tag{12}$$

For $N = 128$ and $N = 192$, this ratio is $13.7\times$ and $11.0\times$, respectively. SAGE composes with any token-eviction method that emits a keep-mask: dropped tokens disappear, retained keys are reconstructed with SAGE, and values bypass the sparse-coding path. Its MLP overhead is paid once per retained key, not at every later attention step. Thus it helps when saved residual rescoring dominates the added projection and MLP work; at very short prompts or very low sparsity, the full-search encoder can already be cheap. The static encoder-bank cost also makes parameter sharing, smaller hidden widths, distillation into grouped modules, or other encoder compression natural deployment optimizations; we leave those variants to future work rather than claiming them in the reported numbers. The latency prototype writes reconstructed dense keys back into the cache, while a production sparse-storage path would store s atom indices and coefficients per retained key; Appendix D gives the byte-level accounting for that path.

4 Experiments

Setup. We evaluate Qwen3-8B and Llama-3.1-8B-Instruct with one $K = 4096$ pre-RoPE key dictionary per layer-head key module. Methods share the same dictionary: Full Cache, a full-search sparse-coding baseline, and SAGE with shortlist size N. GSM8K uses the 1,319-sample test split and a 5-shot prompt. LongBench reports the 8-task average over qasper, qmsum, multi_news, trec, triviaqa, samsum, lcc, and repobench-p with the task-aware preprocessing protocol of sparse-dictionary prior work. RULER reports the 13-task average at 4K, 8K, or 16K. Accuracy runs use bf16 model weights and fp32 dictionaries/coders on H100 80 GB GPUs.

We report two latency views. The encoder latency isolates the compression stage after the model has produced the pre-RoPE keys; this is the stage that differs between the full-search baseline and SAGE. End-to-end latency adds prompt forward, compression, and 128 generated tokens. The distinction is necessary because GSM8K-style workloads are decode-heavy, while LongBench and synthetic long-context probes expose the compression cost more directly. Unless otherwise stated, timings use $T = 8192$ and a single request, and all methods reconstruct into the same dense cache layout before decoding.

Cache-state memory. Because SAGE preserves the sparse code produced by the dictionary encoder, the same method that reduces encoder la-

Model	Shared setting	Benchmark	Score		t_c (s)		t_{e2e} (s)		Speed-up	
			Full	SAGE (Δ)	Full	SAGE	Full	SAGE	t_c	E2E
Llama-3.1-8B-Inst.	$T=8192$, $N=128$, $s=24$	GSM8K	76.80	76.57 (-0.23)	20.19	3.13	22.20	5.11	6.5×	4.4×
		LongBench	51.59	51.39 (-0.20)	20.19	3.13	22.20	5.11	6.5×	4.4×
Qwen3-8B	$T=8192$, $N=192$, $s=24$	GSM8K	90.14	89.76 (-0.38)	22.66	4.85	25.28	7.48	4.7×	3.4×
		LongBench	51.49	51.06 (-0.43)	22.66	4.85	25.28	7.48	4.7×	3.4×
	$T=16384$, $N=192$, $s=32$	RULER-16K	92.75	92.62 (-0.13)	—	—	—	—	2.5–3.0×	—

Table 1: Headline quality and latency at recommended operating points. "Full" denotes the full-search sparse-coding baseline. Parenthesized deltas are SAGE − Full; speed-ups are versus Full. Bracketed settings are shared by spanned rows: T is context length, N is shortlist size, and s is the sparse-code budget.

Model / context	Full KV (MiB)	Sparse-key KV (MiB)	Ratio
Llama, 8K	1,024	704	1.45×
Llama, 16K	2,048	1,408	1.45×
Qwen3, 8K	1,152	792	1.45×
Qwen3, 16K	2,304	1,584	1.45×

Table 2: Algorithmic KV-cache footprint under sparse-key storage at $s = 24$. Values are cache-state VRAM components, not measured peak GPU VRAM.

tency also reduces the key-cache VRAM component when deployed with sparse storage. Table 2 reports the resulting cache-state footprint at the main sparsity. The key component shrinks by $2.67\times$, while total KV shrinks by $1.45\times$ because values remain dense. We separate this algorithmic cache footprint from measured peak GPU memory: our latency prototype writes reconstructed dense keys back into the cache, so it does not realize the VRAM reduction without a sparse-storage or fused reconstruction-attention path. For a key-head dimension d, key/value scalar widths b_k, b_v, and sparse records with b_{idx}-byte indices and b_{coef}-byte coefficients, the per-vector ratios are

$$R_K = \frac{b_k d}{s(b_{\text{idx}} + b_{\text{coef}})}, \tag{13}$$

$$R_{\text{KV}} = \frac{d(b_k + b_v)}{s(b_{\text{idx}} + b_{\text{coef}}) + d b_v}. \tag{14}$$

With $b_k = b_v = b_{\text{idx}} = b_{\text{coef}} = 2$ bytes, $d = 128$, and $s = 24$, these give $R_K = 2.67$ and $R_{\text{KV}} = 1.45$.

Quality. Table 1 shows that constraining the sparse-coding step to a learned shortlist preserves the full-search baseline score at the operating points we use for latency and integration. The largest headline drop is 0.43 pp on Qwen LongBench; Llama LongBench drops 0.20 pp, and Qwen RULER-16K drops 0.13 pp. Per-task LongBench variation and non-headline sparsities are reported in the appendix. The main failure mode is aggressive sparsity, not shortlisting: at $s = 4$ the dictionary itself can lose quality, and the MLP overhead can dominate because the baseline encoder is already cheap.

The headline cells also show that the useful shortlist size is model-dependent but modest. Llama is stable at $N = 128$ for both GSM8K and LongBench. Qwen benefits from $N = 192$ on the long-context headline cells, although its GSM8K score is already close at smaller shortlists. This matches the design goal: the MLP does not need to replace the sparse solver, only to reduce the atom universe to a candidate set large enough for the solver to recover a high-quality reconstruction. Appendix Figure 4 summarizes the shortlist, dictionary-size, and sparsity sweeps.

Latency and Pareto. The latency columns of Table 1 show where the speed-up comes from. At $T = 8192$, Llama's full-search encoder with $s = 24$ spends 20.19 s in compression; SAGE($N = 128, s = 24$) spends 3.13 s, which reduces end-to-end latency from 22.20 s to 5.11 s for a 128-token decode. Qwen shows the same pattern: the full-search encoder spends 22.66 s, while SAGE($N = 192, s = 24$) spends 4.85 s. Prefill and decode throughput are unchanged because SAGE writes reconstructed dense keys back into the same cache shape. The speed-up is therefore concentrated exactly where sparse-dictionary coding was impractical: long-context compression.

Figure 3 confirms the same pattern under the KV-eviction integration setting on RULER-4K, for both Llama-3.1-8B-Instruct ($\tau = -9$) and Qwen3-8B ($\tau = -4$). In each row, the inset plots latency against the sparsity knob $s \in \{16, 24, 32\}$: full

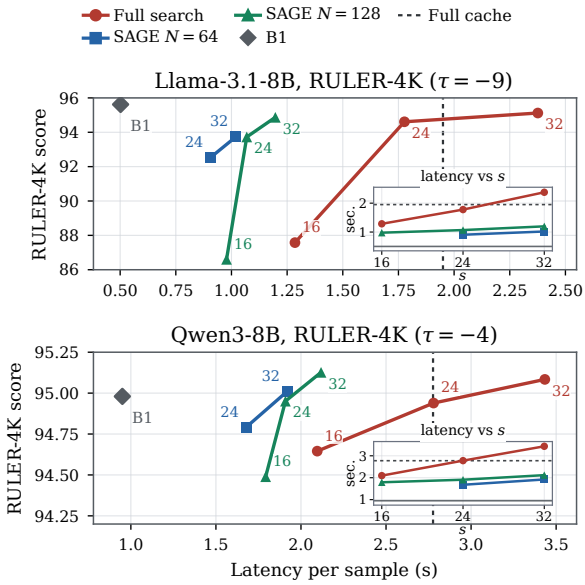

Figure 3: KV-eviction integration on RULER-4K. Each row shows the score–latency Pareto for one backbone; the inset shows the corresponding latency–sparsity sweep at the same eviction threshold.

search rises steeply because every step rescans all $K = 4096$ atoms, while SAGE($N = 64$) and SAGE($N = 128$) stay nearly flat with N controlling a small additive offset. The main axis plots the resulting speed–accuracy Pareto: at matched RULER-4K score, SAGE sits to the left of full search at every operating sparsity. The Full-Cache line is the no-compression latency anchor, and the B1 marker is the eviction-only KVzap point that wins this particular RULER-4K cell on score (see §4.1). This sweep intentionally includes $N = 64$ and $s = 16$ below the headline settings to show the tradeoff shape. SAGE occupies the useful middle band: $s = 24$ or 32 with $N \leq 128$ keeps the quality band tight and makes the encoder latency comparable to the rest of the request rather than orders of magnitude larger.

4.1 Integration with KV-Eviction

We integrate SAGE into the KVzap-style learned-threshold eviction pipeline by replacing the second-stage sparse-coding encoder and keeping the keep-mask fixed. The eviction-only B1 baseline stores retained tokens densely; the full-search baseline and SAGE apply sparse coding to retained keys, giving the same theoretical $\mathrm{CR_K}$ at fixed s. We use this setting to answer a narrower question than "is sparse coding always better than eviction?" The keep-mask is fixed within each threshold, so the full-search baseline and SAGE see the same retained tokens and have the same nominal key compression ratio. Any difference between them is therefore attributable to the encoder. B1 is included to show when a deployment should stop after eviction rather than pay for a second sparse-coding stage.

Table 3 gives the deployment picture. If the second sparse-coding stage is required, SAGE is the better encoder: it preserves the full-search score under paired bootstrap and reduces compression time. Whether the second stage is worth using over B1 is workload-dependent. B1 is stronger on Llama RULER-4K and GSM8K, where the extra key compression does not improve score; LongBench-e is the case where sparse coding is most useful, because it is at parity with B1 while roughly halving the theoretical retained-key footprint.

The bootstrap analysis is deliberately paired. For each task cell, the full-search baseline and SAGE are evaluated on the same prompt set, and we resample within that matched set. This avoids mistaking sample noise for a shortlist effect. Across the 10 full-evaluation headline comparisons summarized in Table 3 (plus the Llama GSM8K $\tau = -6$ smoke-subset sanity check), the 95% confidence interval for SAGE−full search contains zero. In plain terms, the data are compatible with no score change, so we do not detect a quality regression from replacing the full sparse-coding encoder with the learned shortlist encoder in the integration setting. Because "no detected difference" is weaker than proving equivalence, we also apply a fixed-margin two one-sided test at margin ± 0.5 pp (Appendix C, Figure 7). This test establishes equivalence on 6 of 10 full-evaluation cells and non-inferiority at -0.5 pp on 8 of 10; the remaining failures are driven by bootstrap variance at the available n rather than by point deltas exceeding the margin. The B1 comparisons are different: they test a system-level tradeoff between retaining dense keys for fewer tokens and adding a sparse-dictionary representation for retained keys.

4.2 Ablations and Operating Points

The shortlist size need not be large. On Llama at $s = 24$, increasing N from 64 to 128 closes nearly all of the quality gap; $N = 192$ mostly buys margin at extra latency. Sparsity is more important: $s \geq 24$ is the reliable regime, while $s \leq 8$ exposes dictionary-capacity limits. We therefore use $N = 128$ for Llama LongBench-style operating points and $N = 192$ when matching the stronger Qwen

Workload	B1	Full search	SAGE	Δ range	Lat ratio
Llama RULER-4K	95.11	94.67	94.69	$[-0.13, +0.15]$	0.68–0.71
Llama GSM8K	81.73	79.57	80.14	$[+0.00, +1.14]$	0.95–0.98
Qwen GSM8K	78.33	77.30	77.39	$[-0.38, +0.53]$	decode-bound
Qwen LB-e	49.63	49.56	49.49	$[-0.35, +0.16]$	≈ 0.62

Table 3: KV-eviction integration summary over 14 threshold cells. Δ range is SAGE $-$ full search over full-evaluation cells; Lat ratio is SAGE/full-search compression time.

or RULER settings. Full ablation tables are in Appendix B.

Two secondary ablations guide those recommendations. First, dictionary size helps only up to the point where the downstream sparse code can use the extra atoms. In the Llama probe, $K = 2048$ and 4096 have similar held-out reconstruction error at $s = 32$, while the larger dictionary increases the full-dictionary atom-search cost. We keep $K = 4096$ because it is the strong prior sparse-dictionary setting and because SAGE directly targets the large-K bottleneck. Second, sparsity is the more sensitive knob. Very small s values reduce compute but discard too much dictionary capacity, especially on LongBench. This is why the main text uses $s = 24$ for Llama and Qwen LongBench-style runs and $s = 32$ for RULER and KVzap integration cells that need additional margin.

Overall, the experiments support a bounded claim. SAGE is not a new cache-eviction policy, a quantizer, or a learned attention kernel. It is a faster encoder for an existing sparse-dictionary cache compressor. When sparse coding is the chosen second stage, the learned shortlist removes most of the encoder atom-search latency while preserving the quality of the full-search baseline within the resolution of our benchmarks and bootstrap tests.

5 Conclusion

SAGE turns sparse-dictionary coding from an accurate but expensive prototype into a practical KV-cache compression stage for long-context LLM serving. It can be applied to the retained key cache on its own, or composed after an eviction policy has selected which tokens to keep. In both cases, the compressed representation is the same sparse key code that enables lower key-cache footprint under an off-cache storage path, while the measured prototype keeps the downstream decoding interface unchanged by reconstructing dense keys. Empirically, this gives large compression-stage and end-to-end decode-pipeline speed-ups without a detected quality penalty relative to the full-search sparse-coding baseline, including in the KV-eviction integration setting. The practical caveats are equally important: realized VRAM savings require sparse storage or a fused reconstruction-attention path, and sparse coding is useful only when the deployment needs the additional per-key compression beyond what eviction alone provides.

Limitations

We evaluate SAGE on two open-weight 8B model families, one H100 80 GB platform, and one encoder-training seed per layer-head module. We leave larger models, other accelerators, multilingual and safety robustness, and multi-seed encoder variance to future work.

The memory savings are sparse-storage accounting. Our latency prototype reconstructs dense keys before attention, so peak VRAM savings require sparse storage or fused reconstruction-attention. Values remain dense, limiting total KV reduction to $1.45\times$ at $s = 24$. SAGE also adds an MLP bank, so eviction-only caching or full-search encoding may be preferable for short prompts or small dictionaries.

Ethical considerations

This work studies a systems method for KV-cache compression on public benchmarks and open-weight models. It does not collect user data, involve human subjects, or introduce new safety-sensitive datasets. The main deployment risk is overclaiming compression neutrality: compressed caches can change outputs, so users should validate quality and safety on their own workloads. AI assistance was used for editing and formatting; the authors remain responsible for all claims and text.

References

Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant J. Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: KV cache reduction

through key tokens selection for efficient generative inference. In *Proceedings of Machine Learning and Systems*, volume 6, pages 114–127.

Michal Aharon, Michael Elad, and Alfred M. Bruckstein. 2006. K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Transactions on Signal Processing*, 54(11):4311–4322.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, and Wen Xiao. 2024. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*.

Alessio Devoto, Yu Zhao, Simone Scardapane, and Pasquale Minervini. 2024. A simple and effective L_2 norm-based strategy for KV cache compression. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 18476–18499.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S. Kevin Zhou. 2024. Ada-KV: Optimizing KV cache eviction by adaptive budget allocation for efficient LLM inference. *arXiv preprint arXiv:2407.11550*. Accepted at NeurIPS 2025.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2024. Model tells you what to discard: Adaptive KV cache compression for LLMs. In *The Twelfth International Conference on Learning Representations*.

Karol Gregor and Yann LeCun. 2010. Learning fast approximations of sparse coding. In *Proceedings of the 27th International Conference on Machine Learning*, pages 399–406. Omnipress.

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*, volume 37, pages 1270–1303.

Simon Jegou and Maximilian Jeblick. 2026. KVzap: Fast, adaptive, and faithful KV cache pruning. *arXiv preprint arXiv:2601.07891*.

Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. 2024. GEAR: An efficient error reduction framework for KV cache compression in LLM inference. In *Proceedings of the 4th NeurIPS Efficient Natural Language and Speech Processing Workshop*, volume 262 of *Proceedings of Machine Learning Research*, pages 305–321. PMLR.

Jang-Hyun Kim, Jinuk Kim, Sangwoo Kwon, Jae W. Lee, Sangdoo Yun, and Hyun Oh Song. 2025a. KVzip: Query-agnostic KV cache compression with context reconstruction. In *Advances in Neural Information Processing Systems*.

Junhyuck Kim, Jongho Park, Jaewoong Cho, and Dimitris Papailiopoulos. 2025b. Lexico: Extreme KV cache compression via sparse coding over universal dictionaries. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 30672–30687. PMLR.

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*, volume 37, pages 22947–22970.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. 2024a. MiniCache: KV cache compression in depth dimension for large language models. In *Advances in Neural Information Processing Systems*, volume 37, pages 139997–140031.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time. In *Advances in Neural Information Processing Systems*, volume 36, pages 52342–52364.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo 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*, volume 235 of *Proceedings of Machine Learning Research*, pages 32332–32344. PMLR.

Julien Mairal, Francis R. Bach, Jean Ponce, and Guillermo Sapiro. 2009. Online dictionary learning for sparse coding. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 689–696. ACM.

Stéphane Mallat and Zhifeng Zhang. 1993. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing*, 41(12):3397–3415.

Yagyensh Chandra Pati, Ramin Rezaiifar, and P. S. Krishnaprasad. 1993. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Proceedings of the 27th Asilomar Conference on Signals, Systems and Computers*, pages 40–44.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. 2024. SparQ attention: Bandwidth-efficient LLM inference. In *Proceedings of the 41st International Conference on Machine Learning*, Proceedings of Machine Learning Research, pages 42558–42583. PMLR.

Hillel Sreter and Raja Giryes. 2018. Learned convolutional sparse coding. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 2191–2195.

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 *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*, volume 235 of *Proceedings of Machine Learning Research*, pages 47901–47911. PMLR.

Joel A. Tropp and Anna C. Gilbert. 2007. Signal recovery from random measurements via orthogonal matching pursuit. *IEEE Transactions on Information Theory*, 53(12):4655–4666.

Zihao Wang, Bin Cui, and Shaoduo Gan. 2025. SqueezeAttention: 2D management of KV-cache in LLM inference via layer-wise optimal budget. In *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 *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 *International Conference on Learning Representations*, volume 2024, pages 21875–21895.

Bo Xin, Yizhou Wang, Wen Gao, David P. Wipf, and Baoyuan Wang. 2016. Maximal sparsity with deep networks? In *Advances in Neural Information Processing Systems*, volume 29, pages 4340–4348.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark 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, pages 34661–34710.

A Implementation and Training Details

Encoder forward pass. For each layer-head key module $m \in \{1, \ldots, LH_{\text{kv}}\}$, the MLP is

$$f_m(x) = W_3\, \sigma\big(W_2\, \sigma(W_1\, [x; D_{K,m}^{\top} x])\big), \quad (15)$$

with $\sigma = \text{GELU}$, hidden width $h = 1024$, and $W_1 \in \mathbb{R}^{h \times (d+K)}$, $W_2 \in \mathbb{R}^{h \times h}$, $W_3 \in \mathbb{R}^{K \times h}$. With $d = 128$ and $K = 4096$ this is ≈ 9.6 M parameters per layer-head module including biases. Qwen3-8B has $36 \cdot 8$ such modules, so a fully materialised key-side encoder bank is ≈ 2.76 B parameters (≈ 11.0 GB at fp32, ≈ 5.5 GB at bf16). This is a fixed serving overhead, not per-token KV-cache state.

Constrained sparse coding. We implement the shortlisted pursuit routine as a vectorised batch operation over all tokens in a chunk (default chunk size 16,384). The shortlist is a per-token tensor $\mathcal{N}_x \in \mathbb{R}^{N_c \times N}$ produced by top-N over $|f_m(x)|$. The sub-dictionary $D_x \in \mathbb{R}^{N_c \times d \times N}$ is formed by indexing D_K with $\mathcal{N}_x$. Each token then runs the greedy sparse-coding routine for s iterations, scoring the N atoms against the residual and re-fitting the least-squares coefficients with Cholesky (jitter 10^{-6}); we fall back to `torch.linalg.lstsq` on rare near-singular Gram matrices. The reconstructed $\hat{x}_t = D_x \hat{c}_t$ is written in place of the original k_t in the key cache; the matching value vector remains dense, leaving SDPA and decoding paths intact.

Teacher labels and optimization. We mine $\approx$ 5 M tokens of pre-RoPE keys from Wikipedia, the same corpus used to fit the dictionary (Kim et al., 2025b). Full-search sparse-code labels are generated once at $s = 32$ and stored as $(\text{idx}_{32}, \text{scl}_{32})$ pairs. The released shard layout occupies ≈ 8.6 GB for Qwen3-8B and ≈ 7.6 GB for Llama-3.1-8B-Instruct. Table 4 gives the optimizer and training settings used to distill the shortlist predictor.

optimizer	AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.999$)
learning rate	1×10^{-3}, cosine $\rightarrow 10^{-5}$
weight decay	10^{-2}
batch size	1,024 tokens
epochs	18 (Qwen), 24 (Llama)
s_{teacher}	32
α (weight)	20
λ_{rec}	0.1 (Qwen), 0.5 (Llama D)
grad clip	1.0 max-norm
seed	42

Table 4: Encoder training hyper-parameters.

B Additional Empirical Evidence

Ablation sweeps. Figure 4 consolidates the shortlist, dictionary-size, and sparsity sweeps. The main pattern is stable across models: $N = 128$ is already close to saturated on Llama, $N = 192$ buys extra

margin for Qwen long-context settings, and quality degrades mainly when the sparse-code budget s becomes too small. Dictionary size helps only until the sparse code can no longer use the additional atoms; this is why we keep the prior $K = 4096$ setting while focusing the method on the large-K search bottleneck.

Per-task LongBench. Figure 5 expands the headline LongBench average into the eight task scores. The two models differ most on code tasks, but the averages remain close at the selected SAGE settings: 51.39 for Llama and 51.06 for Qwen3.

Operating-point companions. The recommended SAGE settings at $s = 24$ are $(N = 64)$ for Qwen3 GSM8K (89.99, 5.7 s e2e), $(N = 192)$ for Qwen3 LongBench (51.06, 8.6 s e2e), and $(N = 128)$ for both Llama GSM8K (76.57, 5.1 s e2e) and Llama LongBench (51.39, 5.1 s e2e). On Llama at $T = 8192$, the compression-stage latency is 20.19 s for the full-search baseline at $s = 24$ versus 2.71/3.13/4.30 s for SAGE at $N = 64/128/192$. On Qwen3 LongBench, the end-to-end companion to Figure 3 is 2.99 s for Full Cache, 25.28 s for full search, and 5.41/6.19/7.48 s for SAGE at $N = 64/128/192$; GSM8K remains decode-dominated near 2.62 s.

Integration with eviction. Figure 6 merges the Llama GSM8K and Qwen3 KVzap integration grids. Within each threshold, the full-search baseline and SAGE use the same keep-mask and nominal key compression ratio, so their score difference isolates the encoder change. SAGE tracks full search across the grid while cutting Qwen3 LongBench-e second-stage compression from roughly 7.9–8.2 s to 4.6–5.0 s. Eviction-only B1 remains a deployment choice when score is better without the sparse-coding second stage.

C Statistical Validation

We use paired bootstrap distributions to test whether replacing full search with SAGE changes quality in the integration setting. For Llama RULER-4K, per-sample macro scores are resampled within each task; for Llama GSM8K we resample the binary released scorer outputs; for Qwen3 LongBench-e we use a task-level paired bootstrap over the 11-task macro average. All bootstrap runs use $B = 10{,}000$ and seed 20260518. The scripts are `paper/scripts/bootstrap_paired.py`, `paper/scripts/bootstrap_gsm8k.py`, `paper/scripts/bootstrap_longbench.py`, and `paper/scripts/tost_equivalence.py`.

Result. At a fixed ± 0.5 pp margin, 6 of 10 full-evaluation cells pass the TOST equivalence test: all four Llama RULER-4K cells and Qwen LongBench-e at $\tau \in \{-4, -3\}$. Non-inferiority at the same -0.5 pp margin passes on 8 of 10 cells. The two non-inferiority failures are Llama GSM8K $\tau = -9$, where the point delta is 0.0 pp but the binary-task confidence interval is wide, and Qwen LongBench-e $\tau = -5$, where the 90% lower bound is -0.68 pp. The Llama GSM8K $\tau = -6$ row is a 132-sample smoke subset and is not counted in the full-evaluation totals.

D Artifact, Measurement, and Memory Accounting

Artifact. The anonymous artifact linked from the OpenReview submission page contains (i) the SAGE encoder training pipeline (`code/train.py`), (ii) the inference-time press implementations for the full-search baseline and SAGE, (iii) the per-cell `summary.json/predictions_*.csv` behind the headline, integration, and bootstrap analyses, and (iv) the figure-generation scripts used for this appendix. The released artifact also includes the exact `transformers/kvpress` versions (4.57.6 / 0.4.3), seed (42 for training and evaluation; 20260518 for the bootstrap), and hardware configuration (H100 80 GB, SDPA attention) used in each run.

Latency measurement caveats. We do not study FlashAttention-3 effects: SDPA is used uniformly. The 4×H100 measurements time only one GPU at a time; we do not exploit DP/TP. Decode throughput is averaged over 128 new tokens, so very short decode budgets may favour different operating points.

Sparse-storage bytes. This appendix decomposes the theoretical per-token KV-cache footprint under the off-cache sparse-storage path discussed in §3. All numbers use the dtypes in our implementation (bf16 keys/values, fp16 coefficients, 16-bit indices padded from $\lceil \log_2 K \rceil = 12$ bits for alignment). They are not measured GPU peak VRAM: the latency-measurement prototype reconstructs dense keys back into the cache, so its peak VRAM matches Full Cache.

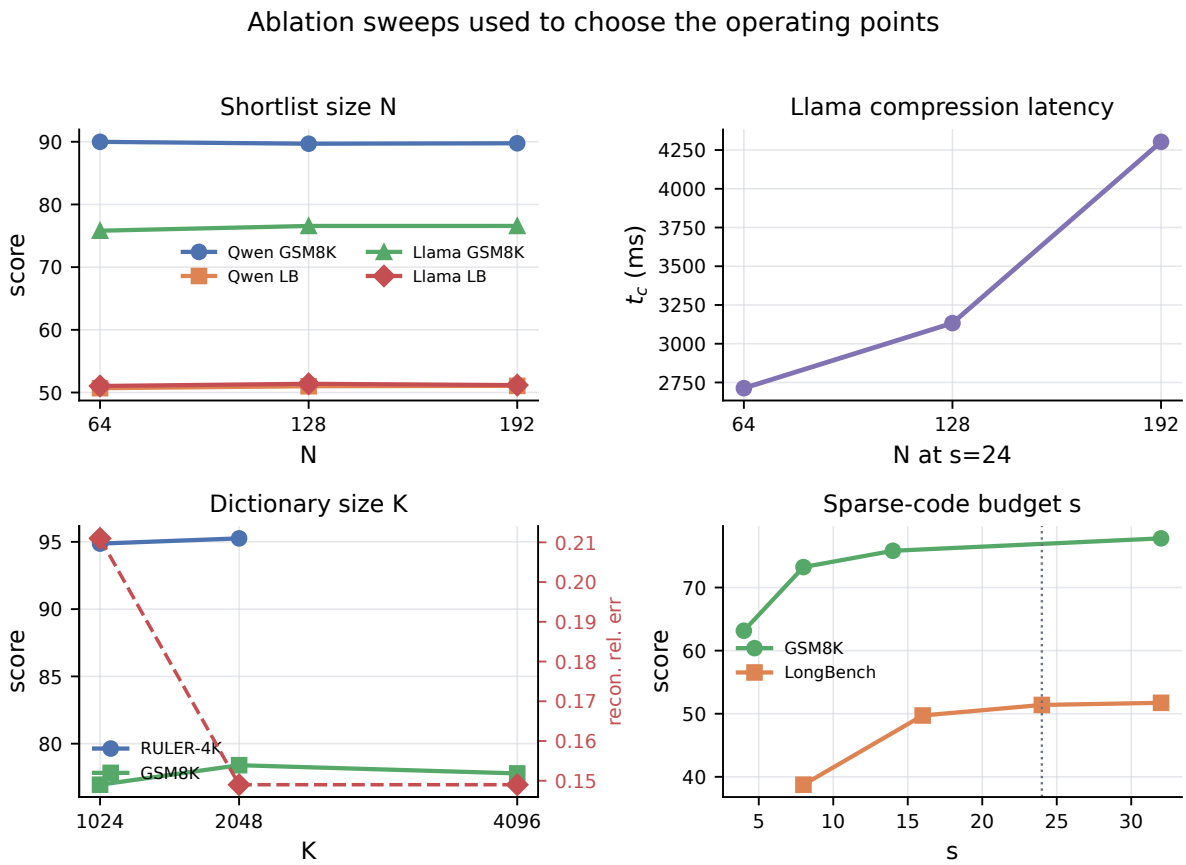

Figure 4: Appendix ablations for shortlist size, dictionary size, sparsity, and Llama compression latency.

Component	Llama	Qwen3
$L \cdot H_{kv}$ (key modules)	$32 \cdot 8$	$36 \cdot 8$
Dense key, bytes/token	65,536	73,728
Sparse key codes ($s = 24$)	24,576	27,648
Dense value, bytes/token	65,536	73,728
Recon. scratch (per key-vector call)	6,144	6,144
Key-only ratio	$2.67\times$	$2.67\times$
Total KV ratio	$1.45\times$	$1.45\times$

Table 5: Per-token sparse-storage byte decomposition.

The encoder and frozen key dictionary D_K are fixed serving overheads, not per-token state: they are loaded once per deployment and amortised across every retained token of every request. The dictionary checkpoint at fp32 occupies ≈ 134 MiB on Llama and ≈ 151 MiB on Qwen3; the per-layer-head MLP encoder adds ≈ 9.6 M parameters per module (Appendix A). At $s = 24$, sparse-key storage shrinks the key cache by $2.67\times$, but because the value cache remains dense the total KV cache shrinks only by $1.45\times$. Driving the total ratio higher requires either a sparse value path or composition with value-cache quantization.

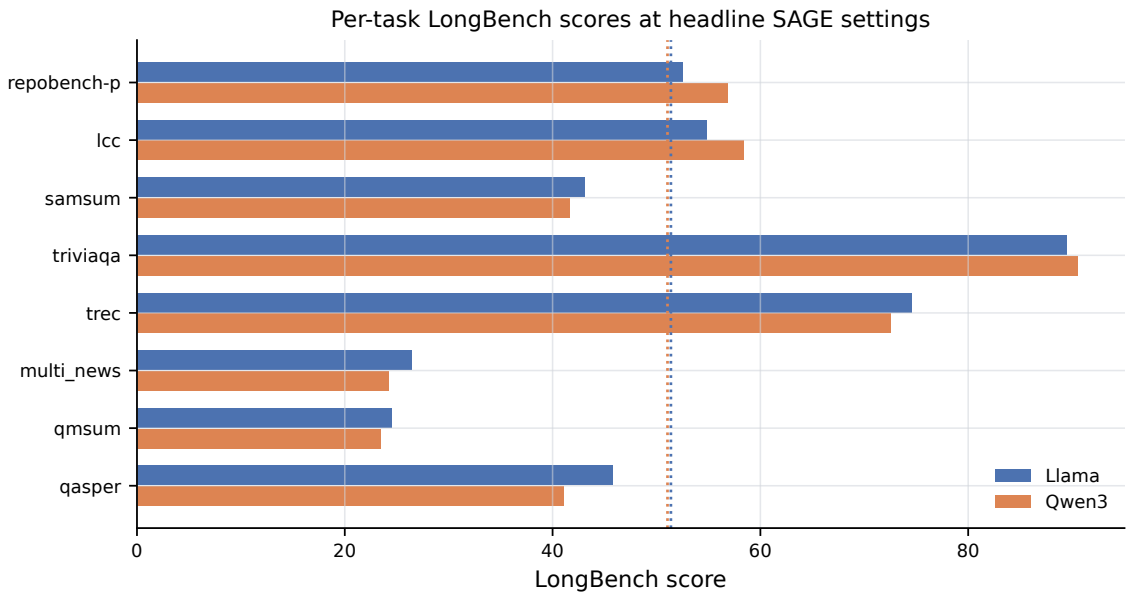

Figure 5: Per-task LongBench scores at headline SAGE settings.

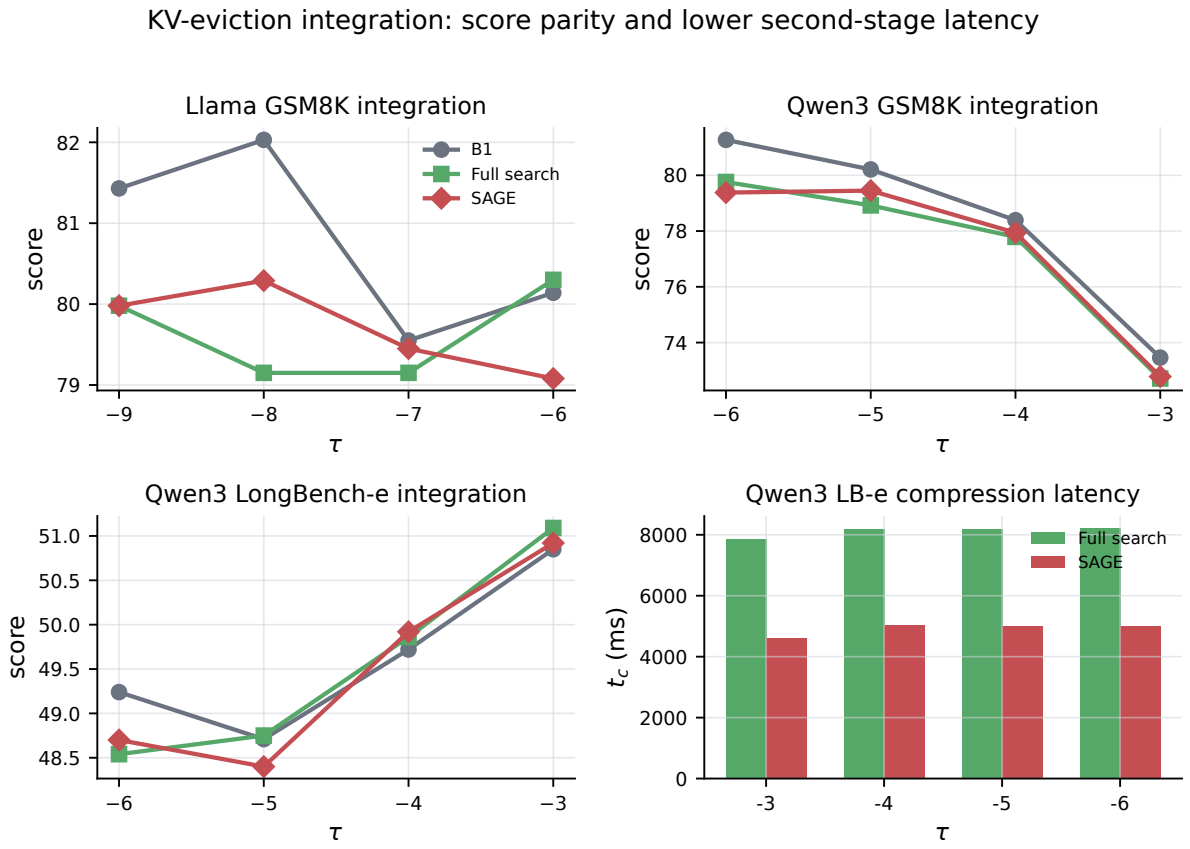

Figure 6: KVzap integration grids across threshold settings.

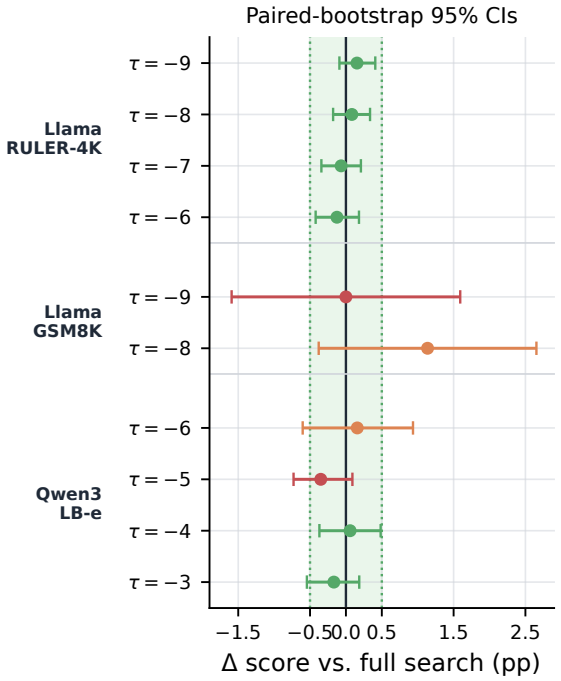

Figure 7: Paired-bootstrap intervals for SAGE−full search.

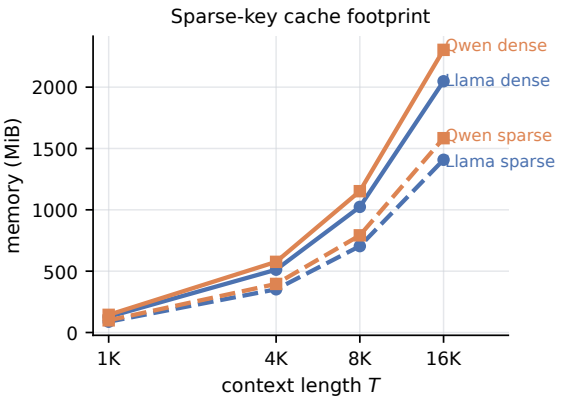

Figure 8: Cache-state memory scaling under sparse-key storage.