SEALKV: ACCURATE LOW-BIT KV COMPRESSION FOR FASTER LLM INFERENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-context reasoning and agentic workloads have made the key–value (KV) cache the dominant cost of LLM serving: it is re-read at every decoding step and grows with every generated token, bounding both decoding speed and concurrency. Compressing it pays off only if it combines high quality, low memory use, and fast serving, yet existing quantization methods sacrifice at least one: they require offline calibration, lose accuracy at low bit widths, or save memory without a measured speedup. We introduce SealKV, the *first* KV-cache compression method that achieves all three, by bringing a trellis-coding quantizer to a cache that grows continuously. Our key insight is that quantization can take place in sealed blocks, each coded only once all its tokens have arrived; sealing decouples trellis encoding from the newest token and exposes a subtractable per-channel offset, shared by neighboring tokens, that rotation alone cannot remove; subtracting it also lifts three existing quantizers by up to 32 points on GPT-OSS-120B retrieval. We also observe that a shorter trellis is also affordable online at little loss: the code's quality depends far less on its window size than its encoding cost does. Co-designed fused dequantization–attention kernels read the codes directly and reconstruct values in registers without writing them back to memory. Without calibration, SealKV is effectively lossless at three bits and above and state-of-the-art at two, where it shrinks KV cache memory by $6.6\times$ while losing as little as 0.37 points on Qwen3-14B and outscoring all evaluated baselines. Integrated end to end into vLLM, SealKV raises peak decode throughput at 64K context to $2.81\times$ that of BF16 at 4 bits and $3.44\times$ at 2 bits, ahead of all evaluated baselines.

1 INTRODUCTION

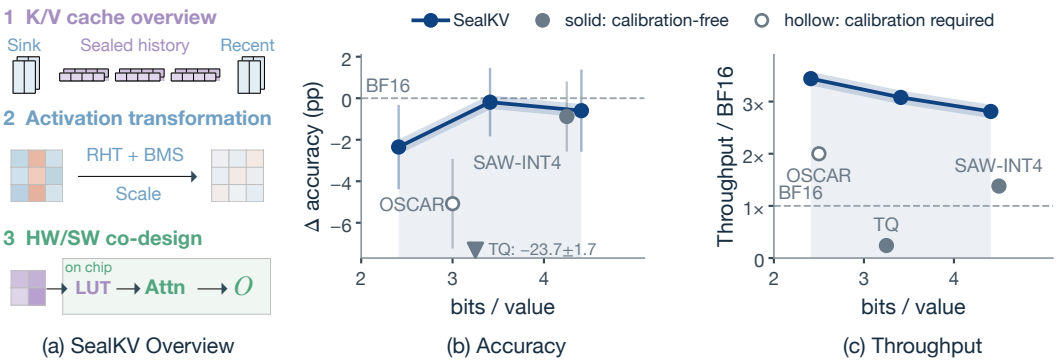
Reasoning and agentic workloads have turned the key–value (KV) cache from an implementation detail of Transformer inference (Vaswani et al., 2017) into its dominant cost. Chain-of-thought reasoning and test-time scaling improve capability via longer decoding (Wei et al., 2022; Snell et al., 2025; Muennighoff et al., 2025; DeepSeek-AI, 2025), and multi-turn tool outputs extend agents' contexts (Yao et al., 2023; Yang et al., 2024; Wang et al., 2026). The cache grows with context length and batch size and is fully read at every decode step, so memory bandwidth limits decoding speed and capacity caps concurrency (Pope et al., 2023; Shah et al., 2024; Kwon et al., 2023). Compression can relieve both, but only if it delivers three properties at once: **high quality**, **low memory**, and **fast serving**, so that more requests are served faster at equivalent quality.

Existing methods achieve at most two of the three (Figure 1). Eviction discards tokens that later queries may need (Xiao et al., 2024; Zhang et al., 2023; Li et al., 2024); scalar quantization codes coordinates independently and falls well short of the rate–distortion limit at low bit widths (Liu et al., 2024; Hooper et al., 2024). Rotation- and vector-quantization methods close much of this gap (Ashkboos et al., 2024; Zandieh et al., 2026; Zhou et al., 2026; Jia et al., 2026) but each gives up calibration-freedom, serving gains, or accuracy at very low bit widths.

Trellis coding (Marcellin & Fischer, 1990) nears the limit by quantizing a vector jointly: a Viterbi search picks one bitstream for it, and each value is decoded from a bit window overlapping its neighbors'. On a Gaussian source, four-bit CPTC (Anonymous, 2026)[1], a general-purpose trellis code, is as accurate as an ideal 3.93-bit code, whereas INT4 matches only a 3.15-bit one (Table 1). Earlier trellis quantizers such as QTIP (Tseng et al., 2024b) are among the strongest weight quantizers, but their encoding is too costly to run online, keeping trellis coding out of the KV cache.

[1] CPTC is submitted concurrently to ICLR 2027; its anonymized draft is attached as supplementary material.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



(a) SealKV Overview (b) Accuracy (c) Throughput

Figure 1: 3 and 4 bits are effectively lossless and 2 bits beat all evaluated 2-bit baselines, while every setting reaches 2.81–3.44× BF16 decode throughput at 64K context, ahead of all baselines. SealKV (K2V2, K4V2, K4V4) versus low-bit KV-cache baselines (gray) on Qwen3-8B. (a) Block-sealed KV caching; RHT, BMS, and per-token scaling; and fused on-chip attention. (b) Mean accuracy change from BF16 versus bits/value; (c) Peak end-to-end decode throughput at 64K context.

We show that this asymmetry stems from design choices for weights, not trellis coding itself. First, search cost grows exponentially with the trellis window, yet quality is far less sensitive to it: shortening CPTC's window from $L = 16$ to $L = 10$ loses only 0.14 effective bits, cuts search work $64\times$, and fits the codebook on chip. Long windows suit weights, which pay the search once; a growing cache pays it at every step. Second, a cache need not code tokens on arrival but can quantize a block after all its tokens arrive, so encoding is batched across layers and never waits on the newest token.

With these insights, *we introduce SealKV, the first KV-cache compression method that achieves high quality, low memory, and fast serving at once.* Despite requiring no calibration, SealKV is effectively lossless on Qwen3 at three bits and above, the most accurate evaluated two-bit method, and up to $3.44\times$ faster than BF16 in decode throughput, ahead of all evaluated baselines (Figure 1b,c). Reaching all three properties requires solving two remaining challenges.

Varied KV distributions require transforms across channels and tokens. A vector-quantization codebook targets a fixed source distribution, yet in keys and values a few channels carry most of the energy, token magnitudes differ, and, as we observe, neighboring tokens share a per-channel offset that rotation redistributes but cannot remove. Rotation and per-token scaling handle the first two (Ashkboos et al., 2024; Ailon & Chazelle, 2009; Chee et al., 2023; Liu et al., 2024), but a downstream quantizer still spends precision on the offset in every token. Because tokens are sealed in complete blocks, SealKV measures the offset once and subtracts it (block-mean subtraction, BMS), reserving precision for token-specific variation.

Turning memory savings into faster serving requires hardware/software co-design. Decode attention is memory-bandwidth-bound, so compression helps only when the time it saves on memory traffic exceeds the time spent encoding new entries and reconstructing old ones. A format designed without its kernel forfeits these savings: TurboQuant (Zandieh et al., 2026) stores fewer bytes per token, yet its upstream vLLM kernel decodes more slowly than BF16 (Kurtić et al., 2026) in our comparison (Figure 1c). SealKV co-designs format and kernel: sealing codes only older tokens and is batched across layers, reconstruction happens in registers within attention, and the shared mean adds a single term per block, so no dequantized cache is written to HBM.

We summarize our main contributions as follows:

- Bring trellis coding to online KV cache compression for the first time by exploiting an asymmetry of CPTC: shorter trellises cut encoding cost exponentially at little quality loss.
- Identify a per-channel offset, shared by adjacent tokens, that rotation cannot remove; subtracting it once per sealed block reduces error for CPTC and existing quantizers.
- Co-design the sealing algorithm, Viterbi encoder, and attention kernel with the hardware and integrate them into vLLM, raising peak decode throughput over BF16 at 64K context to $2.81\times$ at four bits and $3.44\times$ at two bits. Across different model families (Qwen, 2025; OpenAI, 2025), SealKV is effectively lossless at three and four bits and push the frontier for two bits on comprehensive benchmarks on reasoning, retrieval, and multi-turn agentic.

Table 1: Effective bits of four-bit quantizers on a Gaussian source (limit 4.00): scalar formats, prior trellis codes, and CPTC. SealKV uses the shaded L=10 setting at every payload width.

Quantizer	Eff. bits ↑	Quantizer	Eff. bits ↑	Quantizer	Eff. bits ↑
INT4	3.15	QTIP (L=16)	3.86	CPTC (L=16)	3.93
Lloyd-Max	3.36	RPTC (L=16)	3.90	**CPTC** (L=10)	**3.79**

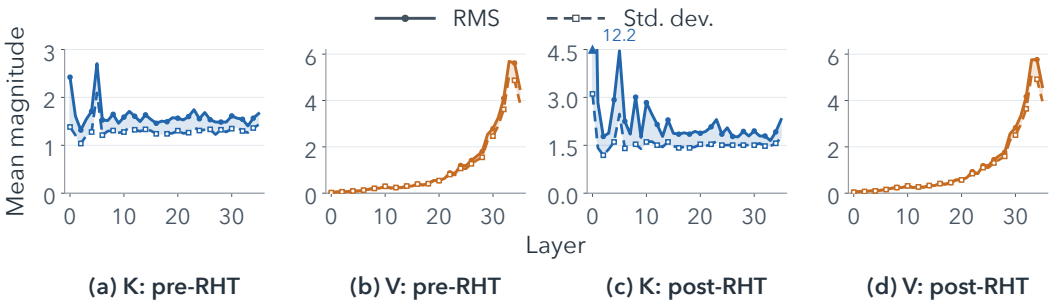


Figure 2: **Channel offsets persist after RHT.** Per-layer averages of per-(head, channel) RMS and standard deviation of keys and values over all tokens of 24 AIME-25 traces for Qwen3-8B, before (a,b) and after (c,d) RHT; the cross-layer non-zero mean motivates BMS in the rotated basis.

2 Background and Observations

2.1 CPTC: Quality and Computational Cost

Trellis coding (Marcellin & Fischer, 1990) quantizes a vector jointly against a codebook but restricts its codes to paths through a trellis, so the best code is found in time linear, rather than exponential, in the vector length. Computed Permutation Trellis Coding (CPTC) (Anonymous, 2026) codes a vector $z \in \mathbb{R}^T$ as a circular bitstream $c \in \{0,1\}^{bT}$ whose L-bit window advances b bits per value (Mao & Gray, 2010):

$$w_d(c) = \sum_{i=0}^{L-1} 2^i \, c[(bd+i) \bmod bT], \qquad \widehat{z}_d = \mathcal{C}_L[w_d(c)], \tag{1}$$

where $\mathcal{C}_L$ is the reconstruction table and b the payload bit budget; tail-biting avoids storing an initial state. Encoding seeks a low-distortion path, $c^\star \in \arg\min_{c\in\{0,1\}^{bT}} \sum_{d=0}^{T-1} \big(z_d - \mathcal{C}_L[w_d(c)]\big)^2$, by a Viterbi search (Viterbi, 1967; Marcellin & Fischer, 1990); for a fixed boundary, the reduced trellis has 2^{L-b} states and 2^b branches per state, giving $O(T2^L)$ search work. We fused it with SealKV's per-token normalization, nibble packing and store writes, so one launch seals a whole layer's K and V, 2.3× faster than CPTC's native kernel on H100. For an i.i.d. unit-variance Gaussian source, the squared-error distortion–rate function is $\mathcal{D}_{\min}(b) = 2^{-2b}$ (Cover & Thomas, 2006), so we express quantizer accuracy as effective bits, $b_{\text{eff}} = -\frac{1}{2}\log_2 \mathcal{D}$ with $\mathcal{D} = \mathbb{E}[\|z - \widehat{z}\|_2^2/T]$.

Table 1 compares four-bit quantizers on this source, excluding metadata. Coding coordinates independently, the best scalar quantizer (Lloyd-Max) reaches 3.36 effective bits; coding the vector jointly, CPTC reaches 3.93 at $L = 16$, close to the limit of 4.00. Shortening CPTC's window to $L = 10$ retains 3.79 effective bits, a loss of only 0.14 for $2^{16-10} = 64\times$ less search work at matched T and b, and shrinks the tabulated codebook 2^6-fold, from 128 KiB of BF16 entries at $L = 16$ to 2 KiB. We therefore fix $L = 10$ and specialize the Viterbi encoder to it: one warp encodes each 128-value row, holding the 2^{10}-entry codebook in registers and the search history in shared memory, so only codes and scales are written to HBM.

2.2 Shared Channel Offsets Across Tokens

Let $X \in \mathbb{R}^{B\times D}$ contain B consecutive key or value vectors from one head, and write $X = \mathbf{1}\mu + E$ with $\mathbf{1} \in \mathbb{R}^{B\times 1}$ and $\mu = (\mu_1, \ldots, \mu_D) \in \mathbb{R}^{1\times D}$ the per-channel means. For channel d,

$$\mu_d = \frac{1}{B}\sum_{t=1}^{B} X_{t,d}, \quad r_d^2 = \frac{1}{B}\sum_{t=1}^{B} X_{t,d}^2, \quad \sigma_d^2 = \frac{1}{B}\sum_{t=1}^{B}(X_{t,d} - \mu_d)^2, \tag{2}$$

$$r_d^2 = \sigma_d^2 + \mu_d^2, \qquad \|X\|_F^2 = \|E\|_F^2 + B\|\mu\|_2^2. \tag{3}$$

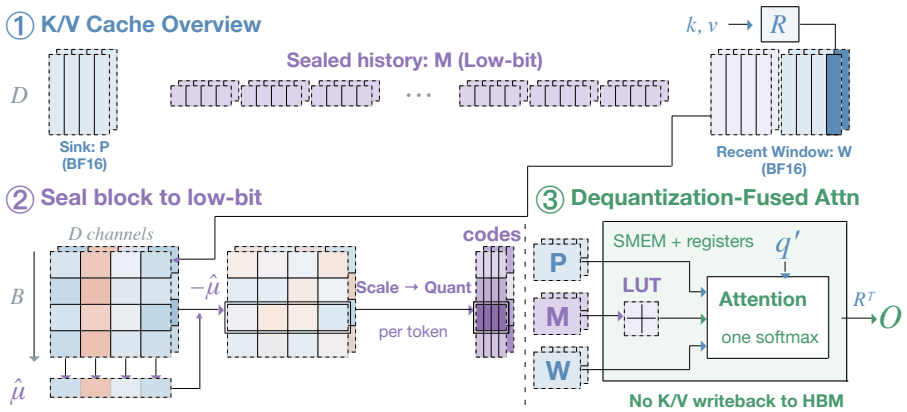


Figure 3: **One decode step in SealKV.** (1) The sink P and recent window W stay in BF16; the sealed middle M holds low-bit codes. (2) A completed block $B \times D$ supplies a per-channel mean, subtracted before the row is scaled per token and CPTC-coded. (3) Lookups and attention share one online softmax on chip; one inverse rotation yields O, with no K/V writeback to HBM.

Thus $r_d > \sigma_d$ indicates a nonzero shared offset, whose energy is the *difference of squares*, not the gap itself. Qwen3-8B traces on AIME-25 show this gap between per-channel RMS and standard deviation for keys and values, before and after randomized Hadamard rotation (RHT; Figure 2).

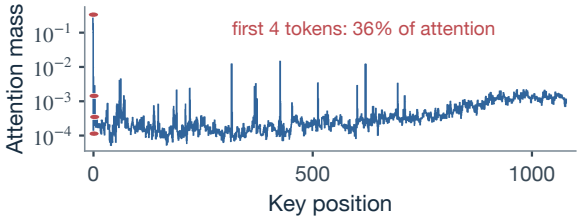
The offset survives orthogonal mixing: for $R^\top R = I$, the rotated mean is μR and $\|\mu R\|_2 = \|\mu\|_2$, so rotations in low-bit quantizers redistribute this component but cannot remove it (Tseng et al., 2024b; Jia et al., 2026). Attention needs the offset, but representing it in every token spends precision that could describe E, and no per-token statistic can isolate structure *across tokens*. SageAttention subtracts a sequence-wide key mean (Zhang et al., 2025), a statistic unavailable during generation; SealKV instead applies block-mean subtraction (BMS) in the rotated basis, with one mean per completed key or value block, fixed once its B tokens arrive and available to attention thereafter.

3 SealKV

SealKV is a calibration-free KV-cache compression system with a flexible number of bits per value. The cache of each KV head is stored as a frame of three regions (Figure 3): a BF16 *sink* of the first P positions, a BF16 *recent window* of the last W, and the *sealed middle* between them, whose tokens are each quantized once, as part of their block, a fixed number of decoding steps after the block completes (Section 3.1). Sealing shapes each token for the trellis code with a rotation, a block mean, and a per-token scale (Section 3.2), and a co-designed kernel dequantizes inside attention, so the compressed cache never makes a round trip through memory (Section 3.3).

3.1 Anatomy of a Decode Step

The sink protects the positions whose error attention amplifies most: on Qwen3-8B and AIME-25, the first 16 tokens are 1.5% of the cache but 38% of the attention-output error under two-bit key quantization (Figure 4). The window protects recency and doubles as a staging area: the block mean requires a *complete* block of B tokens, so a block stays resident until it is sealed, and once computed the mean never changes. Appendix H details the served frame behind Algorithm 1.



Prefix tokens	Cache share	Error share
4	0.4%	37%
16	1.5%	38%
128	11.9%	42%

Figure 4: Attention, and the error it amplifies, concentrate on the earliest positions. Left: attention mass by key position. Right: cache share of the beginning tokens against their share of attention-output error under 2-bit key quantization.

Sealing fires once every B steps: when the length becomes N, the block starting at $N - \Delta$ is sealed as a whole, with a seal distance $B < \Delta \leq W$ that keeps the block complete and resident in every

layer's window. Each block is thus coded exactly once, every position is sealed at least $W - \Delta + 1$ steps before leaving the window, and the sealed middle never has a gap. Sealing runs synchronously within the decode step but never waits on the newest token (Algorithm 1). When a block falls due, layer 0 gathers its rows for all layers and heads in one launch, computes the mean records, and hands the residuals to the quantizer, so every launch has a fixed shape and each step issues a small, context-independent number of them.

Algorithm 1 SealKV inference.
Input: prompt. **Output:** per-step logits.
Parameters: block size B, sink P, window W, seal distance Δ ($B < \Delta \leq W$), payload bits b_K, b_V, mean bits β_μ, trellis window L, and codebook std. $\sigma_{\mathcal{C}}$. Positions are 0-indexed.

PREFILL

1: $N \leftarrow$ prompt length; run full-precision attention; $K' \leftarrow \mathrm{RoPE}(K)R$, $V' \leftarrow VR$
2: Initialize $\mathsf{P}_\ell, \mathsf{W}_\ell$ from K', V'; SEAL(j) for all $j \geq 0$ with $P + jB \leq N - \Delta$

DECODE

1: **for** each decode step **do**
2: $N \leftarrow N + 1$
3: **for** each layer ℓ **do**
4: $(q, k, v) \leftarrow W^{\ell}_{qkv} h$; apply RoPE to q, k
5: $(q', k', v') \leftarrow (qR, kR, vR)$
6: Append (k', v') to W_ℓ; also to P_ℓ if $N - 1 < P$
7: **if** $\ell = 0$ and $N - \Delta - P \in \{0, B, 2B, \ldots\}$ **then**
8: SEAL$\big((N - \Delta - P)/B\big)$
9: $o' \leftarrow$ FUSEDATTENTION$\big(q'; \mathsf{P}_\ell[0, p), \mathsf{M}_\ell[p, r), \mathsf{W}_\ell[r, N)\big)$
10: $o \leftarrow o' R^\top$
11: Emit $\mathrm{lm_head}(h)$

SEAL(j)

1: **for** each layer ℓ', head, and $X \in \{K, V\}$ **do**
2: $X' \leftarrow$ gather rotated block j
3: $(m_j, u_j) \leftarrow$ MEANRECORD(X') §3.2
4: $\widehat{\mu}_j \leftarrow m_j(u_j - 2^{\beta_\mu - 1})$
5: **for** each row t in block j **do**
6: $e_t \leftarrow x'_t - \widehat{\mu}_j$
7: $\rho_t \leftarrow \max(\mathrm{std}(e_t), \epsilon)$
8: $a_t \leftarrow \mathrm{BF16}(\rho_t / \sigma_{\mathcal{C}})$
9: $\{c_t\} \leftarrow$ BATCHEDVITERBI$_{L, b_{K/V}, T=D}\big(\{\sigma_{\mathcal{C}} e_t / \rho_t\}\big)$
10: Store (c_t, a_t) at $\mathsf{M}_{\ell'}[t - P]$ and (m_j, u_j) per block

$p = \min(P, N)$, $r = \max(p, N - W)$ partition sink, middle, and window. Gather reads B rows from prefill K', V' or decode $\mathsf{W}_{\ell'}$. h: newest token's hidden state; W^{ℓ}_{qkv}: QKV projections; R: RHT; $\mathcal{C}_L$: codebook.

3.2 SEALING: SHAPING THE TRELLIS INPUT

SealKV mixes channels, subtracts shared offsets across tokens, and scales each token for CPTC's fixed codebook $\mathcal{C}_L$. The codebook targets $\mathcal{N}(0, \sigma^2_{\mathcal{C}})$ (Sec. 2.1); raw keys and values are not naturally Gaussian.

Mix channels and tokens via RHT and BMS. RHT mixes channels using the normalized Walsh–Hadamard matrix H_D and a fixed balanced sign vector s:

$$\begin{aligned} R = H_D \,\mathrm{diag}(s), \qquad R^{-1} = R^\top = \mathrm{diag}(s) H_D, \\ (K', V', q') = (KR, VR, qR) \rightarrow q' K'^\top = q K^\top, \mathrm{softmax}(\cdot)\, V' = \big(\mathrm{softmax}(\cdot)\, V\big) R. \end{aligned} \tag{4}$$

Keys are rotated after RoPE. Attention runs in the rotated basis, with one inverse rotation $R^\top$ applied to its output. Since $\sqrt{D}\, H_D$ is a ± 1 matrix, each rotation is one BF16 tensor-core product; cached vectors are never re-rotated. RHT spreads coordinate-aligned outliers (Ailon & Chazelle, 2009; Chee et al., 2023) (Figure 5) but preserves the shared mean's norm (Section 2.2).

BMS averages each channel over a completed block X' and stores the mean as one record (m, u) shared by all B tokens: a BF16 scale m and β_μ-bit offset-binary codes u (Equation 5; format in Appendix H). Residuals subtract the *stored* mean $\widehat{\mu}$, absorbing its rounding error, so the codes spend their bandwidth on token-specific information.

SealKV then scales each residual row to the codebook's standard deviation before trellis coding:

$$e_t = x'_t - \widehat{\mu}, \rho_t = \max(\mathrm{std}(e_t), \epsilon), a_t = \mathrm{BF16}(\rho_t / \sigma_{\mathcal{C}}) \rightarrow c_t = \mathrm{Viterbi}_{L,\, b,\, T=D}\big(\sigma_{\mathcal{C}} e_t / \rho_t\big).$$

Scaling uses the standard deviation over D channels without assuming zero channel mean. Each normalized row is cast to FP16 before search and coded by one trellis ($T = D$); keys and values may use different payloads (b_K, b_V). Figure 5 shows that matching the codebook's standard deviation does not make activations Gaussian. For prefill we specialize it: the same search, but keeping only

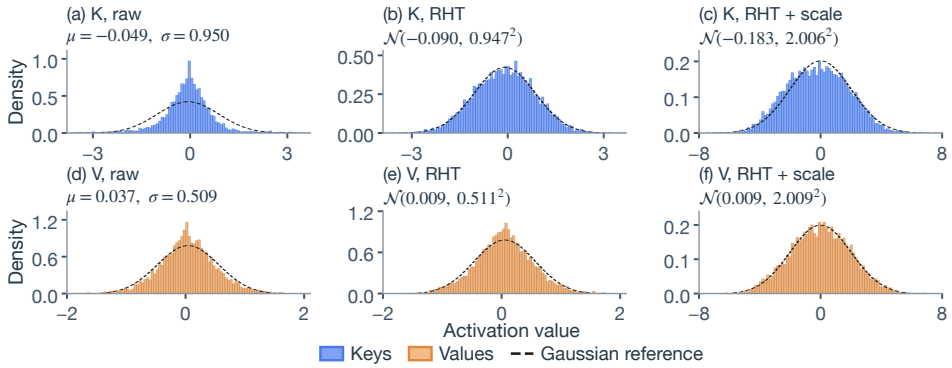


Figure 5: RHT and scaling reshape activation distributions. Qwen3-8B on AIME-25, layer 16, head 0: raw activations (a, d), after RHT (b, e), and after RHT and per-token scaling (c, f); dashed curves are the fitted Gaussians, labeled by μ, σ (raw) or $\mathcal{N}(\mu, \sigma^2)$.

the minimum costs in shared memory and tracing back just the winning path, fused with SealKV's per-token normalization, nibble packing, and stores, so one launch seals a whole layer's K and V, $2.3\times$ faster than CPTC's kernel on the same rows, with bit-identical output (Appendix F), while decode-time sealing calls CPTC's fused Viterbi kernel (Section 3.1).

Count the bits. The codes, token scale, and shared mean reconstruct a sealed token as $\widehat{x}'_{t,d} = a_t\, \mathcal{C}_L[w_d(c_t)] + \widehat{\mu}_d$, with w_d defined in Equation 1. Their total cost is

$$r_{\text{sealed}} = b + \frac{16}{D} + \frac{1}{B}\left(\beta_\mu + \frac{16}{D} + \pi\right) \text{ bits per value,} \tag{5}$$

where π is the record's padding in bits per channel (derived in Appendix H). The rotation and codebook are model-independent; only means and scales are estimated online, so no calibration data are needed.

3.3 Attention Reads Codes, Not Values

The reconstruction $\widehat{x}'_{t,d}$ is linear in the stored data, so it can be substituted into the $q'K'^{\top}$ and PV' contractions and fused with them, and the dequantized cache is never materialized. For a sealed token t in block $j(t)$, with $\bar{k}_t = \mathcal{C}_L[w(c_t^K)]$ and $\bar{v}_t = \mathcal{C}_L[w(c_t^V)]$ the unscaled lookups,

$$\ell_t = \frac{a_t^K\,(q' \cdot \bar{k}_t) + q' \cdot \widehat{\mu}_{j(t)}^K}{\sqrt{D}}, \tag{6}$$

$$o' = \sum_{t \in \text{dense}} p_t\, v'_t + \sum_{t \in \text{sealed}} p_t\, a_t^V\, \bar{v}_t + \sum_j \Big(\sum_{t \in \text{sealed: } j(t)=j} p_t \Big) \widehat{\mu}_j^V, \tag{7}$$

with p one online softmax over sink, sealed middle, and recent window. The key-mean product $q' \cdot \widehat{\mu}_j^K$ is computed once per block and folded into the multiply-add that applies the scale; the value mean enters as one rank-1 update per block, weighted by the attention mass of its sealed tokens, which the online softmax accumulates anyway, so BMS adds no per-token instruction at read time.

We design the fused kernel of each payload pair (b_K, b_V) to approach the memory-bandwidth ceiling. Once the cache is several times smaller, however, bytes arrive faster than the instructions decoding them can issue, so the kernel is bound by instruction issue and shared-memory loads rather than HBM bandwidth (Appendix H.1), and two design choices follow. First, dequantized values never return to HBM or shared memory: packed codes are staged with asynchronous copies, and each codebook lookup yields a BF16 register value that is itself the tensor-core operand, saving a shared-memory store and load per value. Second, reconstruction is a pure lookup: the 2^L CPTC reconstructions are tabulated once instead of computed per value (two multiplications each), and the table is replicated per lane in shared memory with each entry's bank equal to its reading lane, so lookups never conflict and cost only three cheap hardware instructions (a shift, a mask, and a load). Sink and window blocks bypass staging and feed BF16 straight into the same fragments, and the stages of all sequences and heads are split evenly across warps (32 consecutive threads), so ragged batches share one launch and one online softmax.

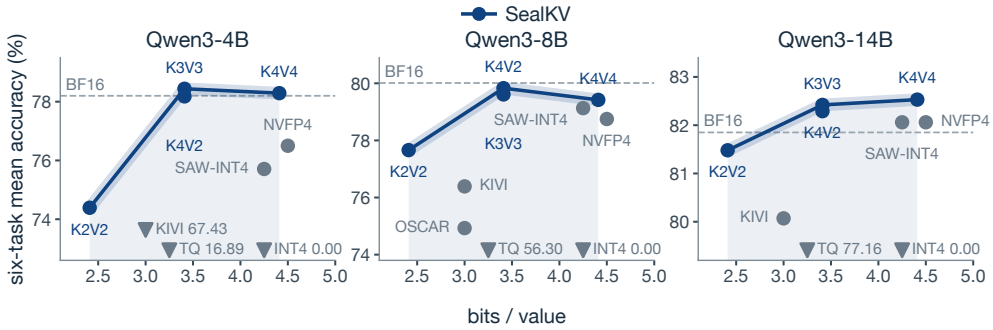


Figure 6: SealKV sets the quality–storage frontier on Qwen3-4B, -8B, and -14B: six-task mean accuracy (reasoning, coding, retrieval) versus bits/value. Triangles mark methods below the plotted range, means beside them; per-task scores and uncertainties: Table 4 (Appendix A).

4 Evaluation

We evaluate SealKV's accuracy, fused kernels, and end-to-end vLLM serving; Appendix E ablates K/V bit allocation (E.1), each component's contribution (E.2), and block-mean precision (E.3).

Experiment Setup. Reasoning, coding, and retrieval runs use Qwen3-4B, -8B, and -14B (Qwen, 2025) on GPQA-D (Rein et al., 2024), HumanEval (Chen et al., 2021), AIME-25/26 (AoPS, 2026), MATH-500 (Hendrycks et al., 2021; Lightman et al., 2024), and 32K multi-key needle-in-a-haystack (NIAH) retrieval (Kamradt, 2023; Hsieh et al., 2024), with comprehensive LongBench tasks in Appendix A.1. GPT-OSS-20B and -120B (OpenAI, 2025) test transfer to another model family and larger scale, and SWE-bench Verified (Jimenez et al., 2024) and τ^2-bench airline and retail (Barres et al., 2025) test multi-step agentic execution. K4V2 denotes 4-bit key and 2-bit value payloads, a nominal average of three bits. Reported bits/value include scales and block-mean metadata. We compare NVFP4, SAW-INT4 (Jia et al., 2026), KIVI (Liu et al., 2024), TurboQuant (Zandieh et al., 2026), and OSCAR (only Qwen3-8B available due to calibration) (Zhou et al., 2026) where reported, alongside BF16, FP8, and a naive INT4 control, on one H100 GPU.

4.1 Reasoning and Retrieval Evaluation

At three and four bits, SealKV is effectively lossless. K3V3, K4V2, and K4V4 stay within 0.7 points of BF16 in six-task mean at every size and above every compressed baseline (Figure 6); both three-bit allocations use 3.41 bits/value, a $4.7\times$ memory reduction without calibration. At two bits, it sets the state of the art. K2V2 uses 2.41 bits/value, a $6.6\times$ memory reduction, and loses 3.81, 2.35, and 0.37 mean-accuracy points on the three models, yet beats calibration-free KIVI (3.00 bits/value) on every model and, on 8B, calibrated OSCAR by 2.73 points.

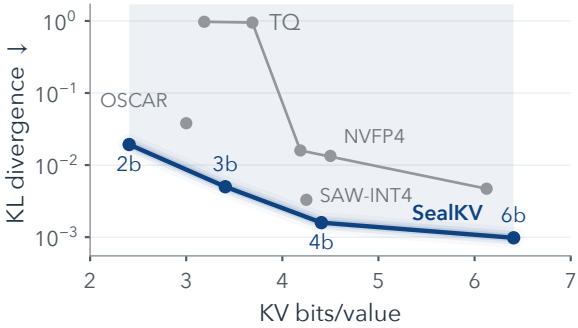


Figure 7: KL divergence on WikiText-2, Qwen3-8B, where SealKV forms a Pareto frontier; shaded: worse than ours.

Table 2: Agentic accuracy (%) on Qwen3-Coder-30B-A3B. τ^2: airline/retail pass rate; subscripts: uncertainty; bold: best compressed score, excluding BF16. Protocol: Appendix C.

Method	Bits ↓	SWE ↑	τ^2-bench ↑ Airline	τ^2-bench ↑ Retail
BF16	16.00	$23.2_{\pm 1.9}$	$43.0_{\pm 5.0}$	$35.4_{\pm 0.9}$
SAW	4.25	$21.8_{\pm 1.8}$	$36.5_{\pm 4.1}$	$35.2_{\pm 3.8}$
NVFP4	4.50	$18.6_{\pm 1.7}$	$34.5_{\pm 6.6}$	$33.1_{\pm 3.8}$
K2V2	2.41	$22.4_{\pm 1.6}$	$35.5_{\pm 3.4}$	$35.1_{\pm 2.6}$
K4V2	3.41	$\mathbf{23.4}_{\pm 1.7}$	$38.0_{\pm 2.3}$	$\mathbf{36.4}_{\pm 2.9}$
K4V4	4.41	$22.6_{\pm 1.9}$	$\mathbf{42.5}_{\pm 4.7}$	$35.5_{\pm 3.4}$

4.2 KL Divergence on WikiText-2

We measure KL divergence from BF16 for Qwen3-8B on WikiText-2 (Merity et al., 2017) (Figure 7). At approximately 4.41 bits/value, SealKV reaches 1.6×10^{-3}, versus 3.3×10^{-3} for SAW-INT4 at 4.25 bits and 1.34×10^{-2} for NVFP4 at 4.50 bits; at 2.41 bits/value, its 1.9×10^{-2} is below

Table 3: GPT-OSS accuracy (%) and KL divergence from BF16 on AIME-25, GPQA-Diamond, and NIAH at 16K and 32K. Each +BMS row gives the change from adding BMS to the baseline above it (SealKV includes BMS); bold marks the best score among SealKV and the unmodified low-bit baselines, excluding BF16 and FP8.

		GPT-OSS-20B					GPT-OSS-120B				
Method	Bits	AIME ↑	GPQA ↑	KLD ↓	16K ↑	32K ↑	AIME ↑	GPQA ↑	KLD ↓	16K ↑	32K ↑
BF16	16.00	89.2	73.0	0.000	96.6	86.6	95.0	80.3	0.000	99.0	83.9
FP8	8.00	85.0	72.3	0.068	94.3	88.3	92.5	79.8	0.075	91.3	82.3
SAW-INT4	4.50	78.3	70.5	0.291	88.7	76.3	73.3	77.0	0.286	84.1	61.1
+BMS	+0.25	+7.5	+2.0	−0.145	+6.5	+9.6	+17.5	+3.0	−0.151	+10.7	+20.2
TQ-4bit	4.25	77.9	68.7	0.336	87.8	77.5	73.3	74.7	0.338	79.1	55.7
+BMS	+0.25	+4.6	+3.8	−0.177	+8.0	+8.1	+16.7	+5.3	−0.196	+14.6	+24.7
NVFP4	4.50	75.4	70.5	0.226	87.8	76.6	70.0	73.2	0.234	70.1	48.2
+BMS	+0.25	+8.8	+2.2	−0.089	+6.8	+9.5	+18.3	+7.6	−0.103	+23.7	+32.2
SealKV	4.52	**90.8**	**73.0**	**0.047**	**95.0**	**86.0**	**94.2**	**80.4**	**0.088**	**98.0**	**83.0**

calibrated OSCAR's 3.8×10^{-2} at the 3-bit level. In the component ablation (Appendix E.2), CPTC after RHT and BMS has about $1.5\times$ lower four-bit KL than scalar INT at matched rate.

4.3 Multi-turn Agentic Benchmarks

On Qwen3-Coder-30B-A3B (Table 2), K4V4 resolves 22.6% of SWE-bench Verified and passes 42.5% and 35.5% of τ^2-bench airline and retail trials, vs. BF16's 23.2%, 43.0%, and 35.4%, at $3.6\times$ less KV cache memory. Lower budgets hold on SWE-bench and retail but degrade airline, yet no compressed arm differs significantly from BF16 on either τ^2 domain, and every SealKV setting beats NVFP4 on all three. A Qwen3-32B user simulator judges retail's natural-language assertions identically across arms. SealKV's trace generation lengths stay closest to BF16's in Appendix D.

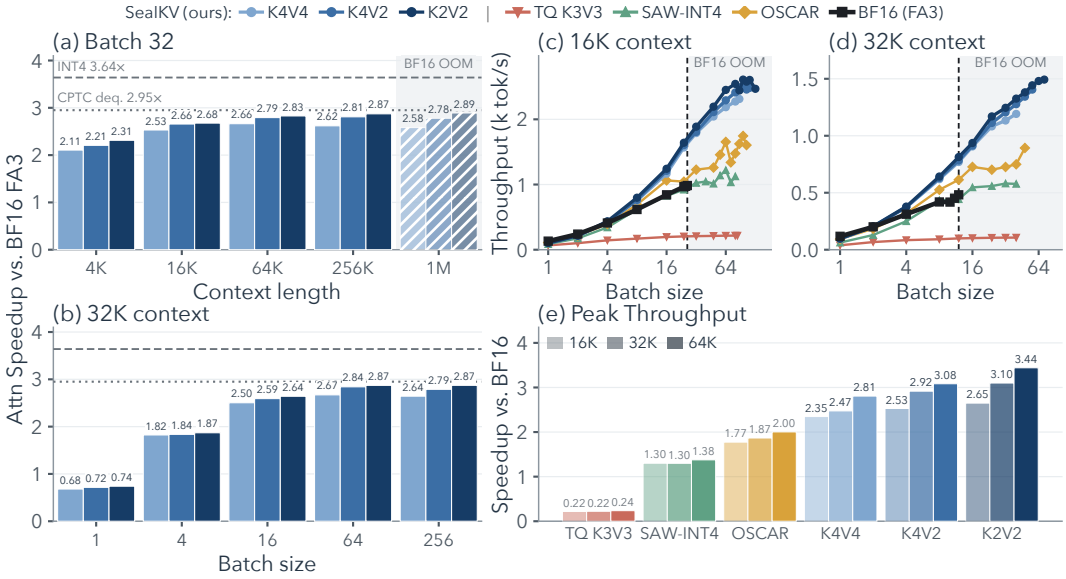


Figure 8: Qwen3-8B on one H100. (a, b) Attention speedup over BF16 FA3; horizontal lines mark our speed-of-light fused INT4 attention kernel and a CPTC dequantize-only kernel; hatched bars project FA3 where BF16 KV exceeds HBM. (c, d) Decode throughput versus batch size; BF16's KV pool is full past the dashed line. (e) Peak throughput relative to BF16 with FA3 decoding.

4.4 Transfer to GPT-OSS and to Other Quantizers

Four-bit SealKV stays within 1.6 points of BF16 on every task and model (Table 3). On GPT-OSS-120B, it reaches 94.2% on AIME-25 and 83.0% on 32K retrieval (BF16: 95.0% and 83.9%), whereas the unmodified four-bit baselines reach only 70.0–73.3% and 48.2–61.1%. SealKV also has the lowest KL divergence of any four-bit method on both models, so the benefit is not specific to Qwen3. To test whether the channel mean matters beyond CPTC, we add BMS to each baseline at $+0.25$

bits/value (+BMS rows in Table 3); BMS improves all 30 model–quantizer–metric combinations: on GPT-OSS-120B, AIME-25 gains 16.7–18.3 points and 32K retrieval 20.2–32.2; on GPT-OSS-20B, 4.6–8.8 and 8.1–9.6; KL divergence falls by 0.089–0.196 in all six configurations. Removing shared offsets thus helps several quantizers, and helps CPTC more than INT on Qwen3-8B (Appendix E.2).

4.5 Kernel Performance and End-to-End Serving

Against BF16 FlashAttention-3 (FA3) (Shah et al., 2024) on the Qwen3-8B decode shape, SealKV at batch 32 is 2.1–$2.3\times$ faster at 4K context and 2.5–$2.9\times$ from 16K to 1M, at most 10% slower than CPTC dequantize-only attention at the same shape, so block means add little (Figure 8a). Speedups across payload widths differ by at most 12%, although K2V2 reads 45% fewer bytes than K4V4: the shared-memory lookup, not HBM traffic, bounds the step. At 32K, K4V4's speedup rises from $0.68\times$ at batch 1 to 2.6–$2.7\times$ from batch 64 (Figure 8b).

End to end, with our fused kernels on quantization and dequantization, we serve Qwen3-8B at 16/32/64K context with 512 generated tokens per request, normalized to each engine's strongest BF16. At equal batch, SealKV is 1.6–$1.7\times$ faster as BF16's KV pool approaches capacity (batch 24 at 16K, 12 at 32K; Figure 8c,d). Peak gains grow with context as KV memory caps BF16's batch, from 2.35–$2.65\times$ at 16K to 2.81–$3.44\times$ at 64K, against at most $1.38\times$ for SAW-INT4, $2\times$ for OSCAR, and $0.24\times$ for TurboQuant K3V3 (Figure 8e).

5 Related Work

Weight and activation quantization. Scalar formats quantize each value on its own grid, using per-channel or per-group scales (Dettmers et al., 2022; Xiao et al., 2023; Lin et al., 2024; Zhao et al., 2024), hardware micro-scaling as in NVFP4 (NVIDIA, 2025), or outlier-spreading rotations (Ashkboos et al., 2024); at any scale granularity, a scalar quantizer remains bounded by the one-dimensional rate–distortion limit (Table 1). Vector quantizers code coordinates jointly: QuIP# pairs incoherence processing with E_8 lattice codebooks (Chee et al., 2023; Tseng et al., 2024a), and QTIP replaces stored codebooks with a computed bitshift trellis (Mao & Gray, 2010; Marcellin & Fischer, 1990) that approaches the Gaussian bound (Tseng et al., 2024b); nested lattices have matching optimality results (Ordentlich & Polyanskiy, 2026), and NestQuant applies them to weights, activations, and KV caches (Savkin et al., 2025). QuIP# and QTIP quantize weights once, offline; SealKV adapts the CPTC quantizer (Anonymous, 2026) via a shorter trellis window and faster Viterbi encoder, combined with block sealing and fused attention for KV cache (Section 3).

KV-cache compression. Eviction discards tokens (Xiao et al., 2024; Zhang et al., 2023; Li et al., 2024) and query-aware sparsity reads part of it per step (Tang et al., 2024; Yang et al., 2025; 2026), whereas quantization keeps every token at fewer bits (Liu et al., 2024; Hooper et al., 2024; Jia et al., 2026). KIVI uses cross-token statistics, but its grouped asymmetric quantizer folds the channel offset into its scalar grid's zero-point; SealKV removes the offset as an explicit vector before coding and carries it through attention in closed form (Section 3.3). Rotated and vector-quantized caches reach lower bit widths (Li et al., 2025; Zhou et al., 2026; Zandieh et al., 2026) but need per-checkpoint calibration or show little end-to-end serving gain (Kurtić et al., 2026).

6 Limitations

SealKV's main systems bottleneck is lookup-table-based reconstruction in software: fewer payload bits save HBM traffic but not the per-value shared-memory lookup, which caps further speedups. Native hardware support for the quantization datatype, would bypass this decoding overhead and let SealKV's speedup scale more closely with the inverse bit width.

7 Conclusion

SealKV shows that KV-cache compression can deliver high quality, low memory, and fast serving at once. Sealing complete blocks is the key: it lets a short-window trellis code run on a growing cache and exposes a per-channel offset that block-mean subtraction removes. Fused kernels then reconstruct values inside attention, so the smaller cache also decodes faster. Without calibration, SealKV is effectively lossless on Qwen3 at three and four bits and, at two bits, the most accurate evaluated method, cutting KV cache memory by $6.6\times$ at a six-task cost of 0.37–3.81 points; four-bit results hold on GPT-OSS and multi-turn agentic tasks. In vLLM at 64K context, K4V4 and K2V2 reach $2.81\times$ and $3.44\times$ BF16's peak decode throughput, ahead of all evaluated baselines.

REPRODUCIBILITY STATEMENT

The appendices provide supporting uncertainty estimates and serving aggregates. Algorithmic constants and the distinction between accuracy and serving configurations appear in Appendix H. Code for full reproduction will be released upon publication.

AI USE STATEMENT

We used generative AI tools to assist implementation, visualization, and manuscript editing. The authors are responsible for the final methods, measurements, citations, and presentation.

ETHICS STATEMENT

This work compresses an inference-time data structure and does not modify model weights. All evaluations use publicly released model weights and public benchmarks; no human subjects and no private or customer data are involved.

REFERENCES

Nir Ailon and Bernard Chazelle. The fast Johnson–Lindenstrauss transform and approximate nearest neighbors. *SIAM Journal on Computing*, 39(1):302–322, 2009.

Anonymous. Computed permutation trellis coding: Near-optimal quantization designed for native hardware acceleration, 2026. Under review as a conference paper at ICLR 2027. Anonymized manuscript included in the supplementary material.

AoPS. AIME problems and solutions. `https://artofproblemsolving.com/wiki/index.php/AIME_Problems_and_Solutions`, 2026. AIME 2025 and 2026 problem sets.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. QuaRot: Outlier-free 4-bit inference in rotated LLMs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 100213–100240, 2024.

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. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3119–3137, 2024.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-Bench: Evaluating conversational agents in a dual-control environment, 2025. URL `https://arxiv.org/abs/2506.07982`.

Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. QuIP: 2-bit quantization of large language models with guarantees. In *Advances in Neural Information Processing Systems*, volume 36, pp. 4396–4429, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. Evaluating large language models trained on code, 2021. URL `https://arxiv.org/abs/2107.03374`.

Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2nd edition, 2006.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning, 2025. URL `https://arxiv.org/abs/2501.12948`.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems*, volume 35, pp. 30318–30332, 2022.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. In *Advances in Neural Information Processing Systems*, volume 37, pp. 1270–1303, 2024.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*, 2024. URL https://arxiv.org/abs/2404.06654.

Jinda Jia, Jisen Li, Zhongzhu Zhou, Jung Hwan Heo, Jue Wang, Tri Dao, Shuaiwen Leon Song, Ben Athiwaratkun, Chenfeng Xu, Tianyi Zhang, and Xiaoxia Wu. SAW-INT4: System-aware 4-bit KV-cache quantization for real-world LLM serving, 2026. URL https://arxiv.org/abs/2604.19157.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Greg Kamradt. Needle in a haystack – pressure testing LLMs. https://github.com/gkamradt/LLMTest_NeedleInAHaystack, 2023.

Eldar Kurtić, Michael Goin, and Alexandre Marques. A first comprehensive study of TurboQuant: Accuracy and performance. vLLM Blog, May 2026. URL https://vllm.ai/blog/2026-05-11-turboquant. Published May 11, 2026. Accessed September 8, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Junyan Li, Yang Zhang, Muhammad Yusuf Hassan, Talha Chafekar, Tianle Cai, Zhile Ren, Pengsheng Guo, Foroozan Karimzadeh, Colorado Reed, Chong Wang, and Chuang Gan. CommVQ: Commutative vector quantization for KV cache compression. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 36831–36845, 2025.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems*, volume 37, pp. 22947–22970, 2024.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=v8L0pN6EOi.

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

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 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*, pp. 32332–32344, 2024.

Mark Z. Mao and Robert M. Gray. Stationary and trellis encoding for IID sources and simulation. In *2010 Data Compression Conference (DCC)*, pp. 3–12, 2010. doi: 10.1109/DCC.2010.8.

Michael W. Marcellin and Thomas R. Fischer. Trellis coded quantization of memoryless and Gauss-Markov sources. *IEEE Transactions on Communications*, 38(1):82–93, 1990.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *5th International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Byj72udxe.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 20275–20321, 2025. doi: 10.18653/v1/2025.emnlp-main.1025.

NVIDIA. Introducing NVFP4 for efficient and accurate low-precision inference. NVIDIA Technical Blog, 2025. URL https://developer.nvidia.com/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/.

OpenAI. Introducing gpt-oss, 2025. URL https://openai.com/index/introducing-gpt-oss/.

Or Ordentlich and Yury Polyanskiy. Optimal quantization for matrix multiplication. *IEEE Transactions on Information Theory*, 72(3):1943–1972, 2026.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems*, volume 5, pp. 606–624, 2023.

Qwen. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level Google-proof Q&A benchmark. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=Ti67584b98.

Semyon Savkin, Eitan Porat, Or Ordentlich, and Yury Polyanskiy. NestQuant: Nested lattice quantization for matrix products and LLMs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 53042–53062, 2025.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. In *Advances in Neural Information Processing Systems*, volume 37, pp. 68658–68685, 2024.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4FWAwZtd2n.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 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*, pp. 47901–47911, 2024.

Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. QuIP#: Even better LLM quantization with Hadamard incoherence and lattice codebooks. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 48630–48656, 2024a.

Albert Tseng, Qingyao Sun, David Hou, and Christopher De Sa. QTIP: Quantization with trellises and incoherence processing. In *Advances in Neural Information Processing Systems*, volume 37, pp. 59597–59620, 2024b.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

Zhenting Wang, Huancheng Chen, Jiayun Wang, and Wei Wei. Memex(RL): Scaling long-horizon LLM agents via indexed experience memory, 2026. URL https://arxiv.org/abs/2603.04257.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 38087–38099, 2023.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=NG7sS51zVF.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024.

Lijie Yang, Zhihao Zhang, Zhuofu Chen, Zikun Li, and Zhihao Jia. TidalDecode: Fast and accurate LLM decoding with position persistent sparse attention. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=EkfLaCJ7bk.

Lijie Yang, Zhihao Zhang, Arti Jain, Shijie Cao, Baihong Yuan, Yiwei Chen, Zhihao Jia, and Ravi Netravali. Less is more: Fast and accurate reasoning with cross-head unified sparse attention. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=trSWJ99WzS.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab Mirrokni. TurboQuant: Online vector quantization with near-optimal distortion rate. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=tO3ASKZlok.

Jintao Zhang, Jia Wei, Pengle Zhang, Jun Zhu, and Jianfei Chen. SageAttention: Accurate 8-bit attention for plug-and-play inference acceleration. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OL44KtasKc.

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. H2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 34661–34710, 2023.

Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate LLM serving. In *Proceedings of Machine Learning and Systems*, volume 6, pp. 196–209, 2024.

Zhongzhu Zhou, Donglin Zhuang, Jisen Li, Ziyan Chen, Shuaiwen Leon Song, Ben Athiwaratkun, and Xiaoxia Wu. OSCAR: Offline spectral covariance-aware rotation for 2-bit KV cache quantization, 2026. URL https://arxiv.org/abs/2605.17757.

APPENDIX OVERVIEW

The appendices hold the evidence behind SealKV's three properties. **High quality**: complete Qwen3 results with LongBench (Appendix A), GPT-OSS results (Appendix B), agentic generation lengths (Appendix D), and codec ablations (Appendix E). **Low memory**: the stored-rate derivation (Appendix H). **Fast serving**: the serving and kernel-sweep protocols (Appendix C), the fused kernel (Appendix H.1), and the backend's serving scope (Appendix H.2). Appendix F restates CPTC's reconstruction, and Appendix G gives supporting illustrations and reproduction settings.

A COMPLETE QWEN3 REASONING AND RETRIEVAL RESULTS

Table 4: Qwen3 task accuracy (%, higher is better) across all four SealKV precisions. Subscripts: standard deviation across runs; bits/value includes its metadata. Bold/underlined: best/second-best compressed score, excluding BF16. Mean: six-task average; Δ = Mean − BF16 Mean in percentage points, computed before rounding (negative: drop).

Qwen3	Method	Bits ↓	GPQA-D	HumanEval	AIME-25	AIME-26	MATH-500	NIAH	Mean ↑	Δ ↑
4B	BF16	16.00	$55.18_{\pm 1.33}$	$95.43_{\pm 0.61}$	$67.92_{\pm 7.33}$	$62.92_{\pm 7.65}$	$93.75_{\pm 0.66}$	94.00	78.20	—
	NVFP4	4.50	$51.89_{\pm 2.24}$	$93.75_{\pm 1.35}$	$64.17_{\pm 5.69}$	$60.00_{\pm 2.72}$	$93.20_{\pm 0.43}$	96.00	76.50	−1.70
	SAW-INT4	4.25	$52.40_{\pm 1.67}$	$88.87_{\pm 1.35}$	$60.00_{\pm 5.44}$	$63.33_{\pm 4.71}$	$93.65_{\pm 0.19}$	96.00	75.71	−2.49
	Naive INT4	4.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	−78.20
	TurboQuant	3.25	$33.08_{\pm 1.57}$	$14.79_{\pm 2.36}$	$4.17_{\pm 1.67}$	$1.67_{\pm 3.33}$	$47.65_{\pm 1.32}$	0.00	16.89	−61.31
	KIVI	3.00	$48.48_{\pm 1.07}$	$91.16_{\pm 0.91}$	$57.50_{\pm 5.95}$	$58.33_{\pm 6.87}$	$93.10_{\pm 1.10}$	56.00	67.43	−10.77
	SealKV K2V2	2.41	$54.04_{\pm 1.75}$	$94.05_{\pm 1.15}$	$62.92_{\pm 5.47}$	$\mathbf{66.67}_{\pm 4.71}$	$93.65_{\pm 0.19}$	75.00	74.39	−3.81
	SealKV K3V3	3.41	$53.41_{\pm 3.12}$	$\mathbf{94.51}_{\pm 0.86}$	$\underline{65.00}_{\pm 6.90}$	$\underline{66.25}_{\pm 3.30}$	$93.45_{\pm 0.72}$	$\underline{98.00}$	**78.44**	+0.24
	SealKV K4V2	3.41	$\underline{54.29}_{\pm 0.65}$	$\underline{94.36}_{\pm 0.77}$	$\mathbf{66.25}_{\pm 6.77}$	$63.33_{\pm 5.04}$	$\underline{93.85}_{\pm 0.47}$	97.00	78.18	−0.02
	SealKV K4V4	4.41	$\mathbf{54.67}_{\pm 1.99}$	$\underline{94.36}_{\pm 0.58}$	$\underline{65.00}_{\pm 3.98}$	$62.50_{\pm 6.11}$	$\mathbf{94.20}_{\pm 0.43}$	**99.00**	$\underline{78.29}$	+0.09
8B	BF16	16.00	$60.35_{\pm 1.05}$	$92.53_{\pm 0.77}$	$68.75_{\pm 4.96}$	$65.00_{\pm 5.35}$	$94.45_{\pm 0.25}$	99.00	80.01	—
	NVFP4	4.50	$55.68_{\pm 3.06}$	$91.62_{\pm 0.77}$	$\mathbf{67.50}_{\pm 4.30}$	$65.83_{\pm 4.19}$	$93.85_{\pm 0.30}$	98.00	78.75	−1.27
	SAW-INT4	4.25	$59.22_{\pm 4.67}$	$91.77_{\pm 0.79}$	$64.17_{\pm 4.19}$	$66.67_{\pm 2.72}$	$93.95_{\pm 0.62}$	$\underline{99.00}$	79.13	−0.88
	Naive INT4	4.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	−80.01
	TurboQuant	3.25	$55.30_{\pm 0.51}$	$75.00_{\pm 2.68}$	$42.50_{\pm 4.19}$	$50.00_{\pm 4.71}$	$90.00_{\pm 0.54}$	25.00	56.30	−23.71
	OSCAR	3.00	$56.82_{\pm 0.36}$	$91.77_{\pm 1.29}$	$56.67_{\pm 9.43}$	$63.33_{\pm 4.71}$	$94.00_{\pm 0.28}$	87.00	74.93	−5.08
	KIVI	3.00	$59.22_{\pm 4.45}$	$91.77_{\pm 2.13}$	$62.50_{\pm 4.93}$	$64.17_{\pm 1.44}$	$93.70_{\pm 0.50}$	87.00	76.39	−3.62
	SealKV K2V2	2.41	$\underline{60.35}_{\pm 1.87}$	$\mathbf{93.14}_{\pm 2.13}$	$62.50_{\pm 7.72}$	$\mathbf{67.92}_{\pm 5.02}$	$94.05_{\pm 0.25}$	88.00	77.66	−2.35
	SealKV K3V3	3.41	$58.46_{\pm 1.04}$	$92.53_{\pm 1.04}$	$65.00_{\pm 7.35}$	$\underline{67.50}_{\pm 3.45}$	$\underline{94.15}_{\pm 0.50}$	**100.00**	$\underline{79.61}$	−0.41
	SealKV K4V2	3.41	$\mathbf{60.73}_{\pm 1.39}$	$91.92_{\pm 1.04}$	$65.42_{\pm 3.96}$	$\underline{67.50}_{\pm 4.96}$	$\mathbf{94.40}_{\pm 0.49}$	$\underline{99.00}$	**79.83**	−0.19
	SealKV K4V4	4.41	$60.23_{\pm 3.83}$	$\underline{92.99}_{\pm 1.61}$	$\underline{66.67}_{\pm 5.63}$	$65.00_{\pm 6.11}$	$93.60_{\pm 0.33}$	98.00	79.42	−0.60
14B	BF16	16.00	$61.74_{\pm 1.95}$	$96.95_{\pm 0.86}$	$67.08_{\pm 5.76}$	$72.08_{\pm 2.48}$	$94.25_{\pm 0.19}$	99.00	81.85	—
	NVFP4	4.50	$61.11_{\pm 1.24}$	$95.58_{\pm 1.52}$	$\mathbf{70.83}_{\pm 5.69}$	$70.42_{\pm 4.30}$	$94.40_{\pm 0.33}$	**100.00**	82.06	+0.21
	SAW-INT4	4.25	$\mathbf{63.01}_{\pm 2.52}$	$95.58_{\pm 1.26}$	$\underline{70.00}_{\pm 4.30}$	$69.17_{\pm 7.88}$	$\underline{94.60}_{\pm 0.54}$	**100.00**	82.06	+0.21
	Naive INT4	4.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	−81.85
	TurboQuant	3.25	$60.48_{\pm 0.64}$	$93.75_{\pm 1.35}$	$63.33_{\pm 2.72}$	$56.67_{\pm 9.03}$	$91.70_{\pm 0.48}$	97.00	77.16	−4.69
	KIVI	3.00	$60.48_{\pm 2.24}$	$93.90_{\pm 2.44}$	$\underline{70.00}_{\pm 4.08}$	$\mathbf{73.33}_{\pm 5.77}$	$\mathbf{94.70}_{\pm 0.10}$	88.00	80.07	−1.78
	SealKV K2V2	2.41	$\underline{62.75}_{\pm 0.64}$	$\mathbf{97.10}_{\pm 1.04}$	$66.25_{\pm 6.53}$	$70.42_{\pm 6.03}$	$94.35_{\pm 0.50}$	98.00	81.48	−0.37
	SealKV K3V3	3.41	$62.25_{\pm 2.59}$	$96.65_{\pm 0.35}$	$68.75_{\pm 3.54}$	$\underline{72.50}_{\pm 5.27}$	$94.40_{\pm 0.23}$	**100.00**	$\underline{82.42}$	+0.58
	SealKV K4V2	3.41	$61.87_{\pm 1.96}$	$\underline{96.95}_{\pm 1.22}$	$\underline{70.00}_{\pm 6.17}$	$70.42_{\pm 4.52}$	$94.50_{\pm 0.26}$	**100.00**	82.29	+0.44
	SealKV K4V4	4.41	$\underline{62.75}_{\pm 3.38}$	$96.19_{\pm 0.77}$	$\mathbf{70.83}_{\pm 3.88}$	$72.08_{\pm 7.11}$	$94.35_{\pm 0.30}$	$\underline{99.00}$	**82.53**	+0.68

Table 4 reports the complete per-task scores and available uncertainty estimates underlying the quality–storage frontier in Figure 6. The mean is unweighted across the six tasks. The naive INT4 control uses a coarse whole-cache quantizer.

A.1 LONGBENCH

Table 5 covers four LongBench domains (Bai et al., 2024): single-document QA (MFQA: MultiFieldQA-en; NrtQA: NarrativeQA; Qasp: Qasper), multi-document QA (2Wiki: 2WikiMultihopQA; HotQA: HotpotQA), and few-shot in-context QA (TrQA: TriviaQA).

Table 5: LongBench scores on Qwen3 (%, higher is better; F1) on six tasks. Configurations and bits/value follow Table 4. Mean: six-task average; Δ: change from BF16 (points), computed before rounding. Bold/underlined: best/second-best compressed scores, excluding BF16.

			Single-doc QA			Multi-doc QA		Few-shot		
Qwen3	Method	Bits ↓	MFQA	NrtQA	Qasp	2Wiki	HotQA	TrQA	Mean ↑	Δ ↑
4B	BF16	16.00	54.35	27.26	43.12	44.74	55.72	88.76	52.33	—
	NVFP4	4.50	52.54	26.52	43.00	37.72	52.35	**90.36**	50.42	−1.91
	SAW-INT4	4.25	52.21	25.31	41.49	36.92	53.23	89.30	49.74	−2.58
	Naive INT4	4.25	2.02	0.15	0.69	0.92	0.36	53.72	9.64	−42.68
	TurboQuant	3.25	20.53	2.99	17.07	10.64	4.04	24.28	13.26	−39.07
	KIVI	3.00	50.29	24.39	37.25	36.09	47.79	85.63	46.91	−5.42
	SealKV K2V2	2.41	51.03	24.46	42.19	42.05	52.67	87.94	50.06	−2.27
	SealKV K4V2	3.41	51.96	**27.43**	42.86	42.86	**55.93**	89.55	51.77	−0.56
	SealKV K4V4	4.41	**54.42**	26.73	**44.39**	**44.82**	55.09	88.76	**52.37**	+0.04
8B	BF16	16.00	53.75	27.09	48.22	42.45	59.12	90.71	53.56	—
	NVFP4	4.50	52.01	25.91	43.28	40.24	57.58	90.03	51.51	−2.05
	SAW-INT4	4.25	51.36	25.85	43.66	41.43	56.39	90.46	51.53	−2.03
	Naive INT4	4.25	15.88	1.51	14.12	12.31	8.69	71.29	20.63	−32.92
	TurboQuant	3.25	36.09	11.14	38.06	25.29	11.33	59.37	30.21	−23.34
	OSCAR	3.00	44.60	15.67	33.68	38.12	47.42	81.56	43.51	−10.05
	KIVI	3.00	49.42	23.22	41.67	40.01	52.03	89.26	49.27	−4.29
	SealKV K2V2	2.41	52.96	27.14	43.96	39.87	56.39	90.36	51.78	−1.78
	SealKV K4V2	3.41	53.25	27.56	47.13	42.36	**60.06**	90.16	53.42	−0.14
	SealKV K4V4	4.41	**54.11**	**27.99**	**48.17**	**43.20**	58.89	**90.71**	**53.85**	+0.29
14B	BF16	16.00	52.75	28.29	45.76	54.43	63.55	92.00	56.13	—
	NVFP4	4.50	**53.87**	27.63	**46.07**	52.06	61.16	92.13	55.49	−0.64
	SAW-INT4	4.25	53.72	29.36	45.62	53.05	**64.15**	91.25	56.19	+0.06
	Naive INT4	4.25	15.35	1.27	7.92	6.86	6.30	59.88	16.26	−39.87
	TurboQuant	3.25	51.43	28.18	42.39	50.36	59.44	90.87	53.78	−2.35
	KIVI	3.00	51.07	28.30	39.86	48.27	60.35	91.38	53.21	−2.93
	SealKV K2V2	2.41	53.67	28.01	45.17	50.76	61.29	91.21	55.02	−1.11
	SealKV K4V2	3.41	53.31	**29.95**	45.26	53.65	63.53	**92.65**	**56.39**	+0.26
	SealKV K4V4	4.41	52.41	28.11	45.80	**54.65**	62.87	92.25	56.02	−0.12

B COMPLETE GPT-OSS RESULTS

Table 6 restates the GPT-OSS comparison of Section 4.4 with reported uncertainty, and Table 7 gives every BMS-augmented score with its change from the corresponding baseline.

Table 6: GPT-OSS accuracy (%) and KL divergence from BF16 with reported uncertainty, baselines without BMS; the compact form is Table 3. Bold: best low-bit score per task, excluding BF16 and FP8.

		GPT-OSS-20B					**GPT-OSS-120B**				
Method	Bits	AIME ↑	GPQA ↑	KLD ↓	16K ↑	32K ↑	AIME ↑	GPQA ↑	KLD ↓	16K ↑	32K ↑
BF16	16.00	$89.2_{\pm 1.4}$	$73.0_{\pm 1.1}$	0.000	$96.6_{\pm 0.7}$	$86.6_{\pm 0.9}$	$95.0_{\pm 2.6}$	$80.3_{\pm 1.9}$	0.000	$99.0_{\pm 0.2}$	$83.9_{\pm 2.1}$
FP8	8.00	$85.0_{\pm 2.0}$	$72.3_{\pm 1.1}$	0.068	$94.3_{\pm 1.6}$	$88.3_{\pm 1.0}$	$92.5_{\pm 2.4}$	$79.8_{\pm 2.1}$	0.075	$91.3_{\pm 0.5}$	$82.3_{\pm 1.8}$
SAW-INT4	4.50	$78.3_{\pm 2.0}$	$70.5_{\pm 2.1}$	0.291	$88.7_{\pm 2.2}$	$76.3_{\pm 1.6}$	$73.3_{\pm 3.6}$	$77.0_{\pm 1.9}$	0.286	$84.1_{\pm 0.6}$	$61.1_{\pm 2.4}$
TQ-4bit	4.25	$77.9_{\pm 2.0}$	$68.7_{\pm 2.2}$	0.336	$87.8_{\pm 0.9}$	$77.5_{\pm 0.8}$	$73.3_{\pm 3.4}$	$74.7_{\pm 2.1}$	0.338	$79.1_{\pm 2.1}$	$55.7_{\pm 1.1}$
NVFP4	4.50	$75.4_{\pm 1.9}$	$70.5_{\pm 2.2}$	0.226	$87.8_{\pm 1.8}$	$76.6_{\pm 2.3}$	$70.0_{\pm 3.3}$	$73.2_{\pm 2.0}$	0.234	$70.1_{\pm 2.1}$	$48.2_{\pm 3.4}$
SealKV	4.52	$\mathbf{90.8}_{\pm 2.6}$	$\mathbf{73.0}_{\pm 2.3}$	**0.047**	$\mathbf{95.0}_{\pm 2.2}$	$\mathbf{86.0}_{\pm 2.9}$	$\mathbf{94.2}_{\pm 2.1}$	$\mathbf{80.4}_{\pm 2.1}$	**0.088**	$\mathbf{98.0}_{\pm 1.4}$	$\mathbf{83.0}_{\pm 3.8}$

Table 7: BMS ablation on GPT-OSS. Each cell gives the score after adding BMS, followed by its change from the corresponding unmodified baseline in Table 6. BMS adds approximately 0.25 bits/value and improves every listed metric.

	GPT-OSS-20B				GPT-OSS-120B			
Benchmark	**BF16**	SAW-INT4	TQ-4bit	NVFP4	**BF16**	SAW-INT4	TQ-4bit	NVFP4
AIME-25 ↑	89.2	$85.8_{(+7.5)}$	$82.5_{(+4.6)}$	$84.2_{(+8.8)}$	95.0	$90.8_{(+17.5)}$	$90.0_{(+16.7)}$	$88.3_{(+18.3)}$
GPQA-D ↑	73.0	$72.5_{(+2.0)}$	$72.5_{(+3.8)}$	$72.7_{(+2.2)}$	80.3	$80.0_{(+3.0)}$	$80.0_{(+5.3)}$	$80.8_{(+7.6)}$
KLD ↓	0.000	$0.146_{(-0.145)}$	$0.159_{(-0.177)}$	$0.137_{(-0.089)}$	0.000	$0.135_{(-0.151)}$	$0.142_{(-0.196)}$	$0.131_{(-0.103)}$
NIAH 16K ↑	96.6	$95.2_{(+6.5)}$	$95.8_{(+8.0)}$	$94.6_{(+6.8)}$	99.0	$94.8_{(+10.7)}$	$93.7_{(+14.6)}$	$93.8_{(+23.7)}$
NIAH 32K ↑	86.6	$85.9_{(+9.6)}$	$85.6_{(+8.1)}$	$86.1_{(+9.5)}$	83.9	$81.3_{(+20.2)}$	$80.4_{(+24.7)}$	$80.4_{(+32.2)}$

C SERVING MEASUREMENTS AND EVALUATION PROTOCOL

Serving setup. Serving uses Qwen3-8B on one H100 80 GB with B identical prompts of 15,336, 31,248, or 63,144 tokens, so that the context after 616 generated tokens is 15,952, 31,864, or 63,760 tokens (16K, 32K, and 64K). Decode is isolated by the difference method: a 616-token and a 104-token generation of the same prompts share an identical prefill, so their time difference is exactly 512 decoding steps at batch B; each point is measured twice with alternating legs. SealKV runs as a vLLM 0.16 plugin with RHT, INT4 block means, protected BF16 regions, and online sealing, all inside the timed decode. SAW-INT4 runs in its authors' SGLang fork, OSCAR in its authors' SGLang tree, and TurboQuant in upstream vLLM 0.20.2, each at its released default configuration. Each ratio divides by BF16 measured in the same engine with its fastest attention backend (FA3 decode); SealKV and its BF16 reference are the same vLLM 0.16 build with the plugin enabled or disabled. Memory fractions are 0.93 unless noted: vLLM BF16 uses 0.97 at 16K and 32K; the OSCAR tree runs OSCAR at its authors' 0.80 (0.70 at 16K batches 56–104 and 0.93 at 32K batches 2–8) and its BF16 at 0.90–0.93; and the SAW fork uses 0.80 at 64K. Each (method, context, batch) point starts a fresh engine, and the batch grows until the KV pool can no longer hold B complete sequences or start-up runs out of memory; a curve keeps the best measurement at each batch, and its peak is the maximum over batches. Every 64K run in every engine enables YaRN with factor 4, since Qwen3-8B's rotary table covers 40,960 positions. Repetition spread is below 2% of the step time at most points, and 40 points measured on two identical machines agree within 0.968–1.022.

Kernel sweep protocol. Figure 8a,b and Tables 9 and 10 come from one script that, at each (context, batch) shape, allocates fresh random BF16 operands and times BF16 FA3 and then the SealKV kernels on the same H100 80 GB SXM (132 SMs, CUDA 12.8), with one query token per sequence, 32 query heads, 8 KV heads, and head dimension 128. After a 3 s idle pause, each kernel runs three warm-up launches, is captured five times into one CUDA graph, and is replayed once untimed; a point is the median over the next ten replays (50 launches), which matches vLLM's graph-replayed decode. SealKV runs on the tensors the serving plugin holds (BF16 sink and ring, packed residual codes, BF16 scales, and INT4 block-mean records) with its balanced schedule, merge kernel, and the launch configuration the plugin's start-up autotuning selects for the shape. FA3 is vLLM's Hopper build, called with contiguous BF16 K/V per sequence, a preallocated output, and automatic split-KV. The reference kernels are INT4 fused attention (64 B of codes plus a BF16 scale and zero per token, head, and tensor) and CPTC dequantize-only attention (64 B of trellis codes plus a BF16 scale), timed under the same protocol. Bandwidth divides the bytes a kernel reads by its time: $2BS \cdot H_{kv}D \cdot 2$ bytes for BF16, and codes, scales, mean records, sink, and ring for SealKV. At 1M context and batch 32, BF16 K/V exceeds HBM, so that group divides the measured SealKV time by four times FA3's measured 256K time. The reference lines are the two kernels at large batch: INT4 at $3.64\times$ is its mean over the largest batch that fits at each context from 4K to 1M, and CPTC at $2.95\times$ marks where it saturates; Table 10 lists both at every bar shape. FA3 and each SealKV width on real sealed codes match FP32 attention to relative error at most 2.4×10^{-3}. Repeating the sweep on a second machine reproduces the K4V4 grid with a median speedup ratio of 0.994.

Accuracy setup. The Qwen3 evaluation uses the released 4B, 8B, and 14B weights, temperature 0.6, top-$p = 0.95$, top-$k = 20$, and a 32,768-token generation cap; NIAH decodes greedily for 128 tokens. The BF16 and SealKV rows use eight seeded runs for each AIME benchmark, four

Table 8: Serving Qwen3-8B on one H100 with 512 generated tokens per request, each method in its own engine. Peak: maximum decode throughput over the batch sizes the engine holds, relative to the strongest BF16 of the same engine; vLLM 0.16 BF16 peaks at 985, 482, and 220 tokens/s with 26, 12, and 5 sequences at 16K, 32K, and 64K (SAW fork: 938, 447, 223; OSCAR tree: 984, 478, 222; vLLM 0.20.2 for TurboQuant: 988, 476, 220). Same batch: throughput relative to that BF16 at equal batch size B; the last column of each context is the largest batch at which every method was measured.

	Peak			Same batch							
				16K			32K			64K	
Method	16K	32K	64K	B=1	8	24	1	8	12	1	4
SAW-INT4	1.30	1.30	1.38	0.68	1.02	1.03	0.54	1.03	0.99	0.40	0.76
OSCAR	1.77	1.87	2.00	0.83	1.13	1.09	0.78	1.24	1.28	0.71	1.10
TurboQuant K3V3	0.22	0.22	0.24	0.50	0.27	0.21	0.32	0.22	0.21	0.23	0.22
SealKV K4V4	**2.35**	**2.47**	**2.81**	0.78	1.22	1.62	0.82	1.48	1.60	0.91	1.45
SealKV K4V2	**2.53**	**2.92**	**3.08**	0.77	1.24	1.66	0.82	1.50	1.63	0.90	1.49
SealKV K2V2	**2.65**	**3.10**	**3.44**	0.78	1.29	1.70	0.83	1.53	1.69	0.93	1.51

Table 9: K4V4 decode-attention speedup over BF16 FA3 by context and batch (top), and K4V2 / K2V2 at selected contexts (bottom), under the protocol of Appendix C. –: FA3 cannot run with BF16 K/V in 80 GB (out of memory at 68.7 GB; larger than HBM beyond).

Context	B=1	B=4	B=8	B=16	B=32	B=64	B=128	B=256
4K	0.22	1.01	1.32	1.70	2.11	2.31	2.49	2.53
8K	0.27	1.21	1.66	2.08	2.34	2.52	2.63	2.58
16K	0.52	1.50	1.97	2.33	2.53	2.66	2.74	2.60
32K	0.68	1.82	2.25	2.50	2.62	2.67	2.65	2.64
64K	0.92	2.09	2.45	2.61	2.66	2.66	2.65	–
128K	1.32	2.35	2.62	2.63	2.59	2.65	–	–
256K	1.64	2.53	2.67	2.66	2.62	–	–	–
512K	2.02	2.64	2.59	2.59	–	–	–	–
1M	2.31	2.61	2.60	–	–	–	–	–
K4V2 / K2V2 at the same shapes								
16K	0.49/0.50	1.48/1.53	2.04/2.09	2.42/2.46	2.66/2.68	2.74/2.80	2.86/2.88	2.80/2.87
64K	0.91/0.96	2.16/2.19	2.54/2.59	2.70/2.74	2.79/2.83	2.84/2.88	2.84/2.93	–
256K	1.63/1.69	2.63/2.64	2.74/2.78	2.82/2.87	2.81/2.87	–	–	–
1M	2.36/2.38	2.80/2.83	2.80/2.89	–	–	–	–	–

for GPQA-Diamond, HumanEval, and MATH-500, and one 100-sample NIAH run; KIVI uses four runs each for AIME-25, AIME-26, and GPQA-Diamond, two for HumanEval and MATH-500, and one NIAH run. SealKV's accuracy runs keep a 128-token BF16 sink and a 256–271-token BF16 window. The WikiText-2 comparison uses 65,504 tokens and computes KL divergence to the same model with BF16 KV caching. SWE-bench Verified uses 500 instances with Qwen3-Coder-30B-A3B, at most 100 steps or two hours per instance; τ^2-bench uses all 50 airline and 114 retail tasks with four trials each at temperature 0, a Qwen3-32B user simulator shared by every arm, and at most 300 steps or 3,600 s per simulation; simulations that hit the time limit count as failures. τ^2-bench paired tests are exact McNemar tests over tasks, each scored by the majority of its trials (ties as failures).

Baseline scope. The Qwen3 task comparison uses the precision budgets shown in Table 4, including 3.25 bits/value for the evaluated TurboQuant configuration and 3.00 for OSCAR at 8B. The GPT-OSS comparison uses a four-bit TurboQuant configuration at 4.25 bits/value; each BMS variant adds approximately 0.25 bits/value of mean metadata (0.27 at $D = 64$, $B = 16$). In the accuracy runs, SAW-INT4 is charged a 16-bit scale and zero per token-head vector, $4 + 32/D$ bits/value (4.25 at $D = 128$; 4.50 on GPT-OSS, $D = 64$); NVFP4 stores one FP8 scale per 16 values (4.50); and OSCAR at 8B stores INT2 codes with an FP32 scale and zero per 64-channel group, as in its authors' tree (3.00). Figure 1c places SAW-INT4 and OSCAR at their serving layouts: SAW-INT4's

Table 10: Numbers behind Figure 8a,b: FA3 time, speedup over FA3 of the three SealKV widths and of the two reference kernels at the same shape, and achieved bandwidth on each kernel's stored bytes. †: BF16 K/V does not fit in 80 GB; FA3 projected at its measured 3.09 TB/s.

Context × batch	FA3 μs	Speedup over FA3					Achieved TB/s			
		K4V4	K4V2	K2V2	INT4 attn.	CPTC deq.	FA3	K4V4	INT4	CPTC
(a) batch 32, context swept										
4K × 32	192	2.11	2.21	2.31	3.14	1.94	2.80	2.02	2.34	1.41
16K × 32	717	2.53	2.66	2.68	3.38	2.54	3.00	2.22	2.70	1.97
64K × 32	2797	2.66	2.79	2.83	3.50	2.78	3.07	2.29	2.85	2.21
256K × 32	11137	2.62	2.81	2.87	3.57	2.87	3.09	2.23	2.93	2.28
1M × 32	44546†	2.58	2.78	2.89	–	–	–	2.20	–	–
(b) 32K context, batch swept										
32K × 1	58	0.68	0.72	0.74	1.60	0.80	2.33	0.45	1.00	0.48
32K × 4	187	1.82	1.84	1.87	3.16	1.93	2.87	1.48	2.40	1.42
32K × 16	711	2.50	2.59	2.64	3.38	2.53	3.02	2.15	2.70	1.96
32K × 64	2845	2.67	2.84	2.87	3.55	2.84	3.02	2.29	2.86	2.21
32K × 256	11161	2.64	2.79	2.87	3.60	2.91	3.08	2.31	2.94	2.31

fork stores FP32 scales and zeros (4.50), and OSCAR's serving configuration uses 128-channel groups (2.50). KIVI uses its published configuration (Liu et al., 2024): two-bit asymmetric quantization with per-channel keys and per-token values, group size 32, and a 128–159-token full-precision residual; its BF16 scale and zero point per 32 values give 3.00 bits/value.

D GENERATION LENGTH ON THE AGENTIC BENCHMARKS

We record the number of output tokens each arm generates on the two agentic benchmarks of Section 4.3. Figure 9 plots the per-instance distributions and Table 11 summarizes them on paired samples: all 200 τ^2-bench airline simulations and 337 SWE-bench Verified instances (instances 150–499 with per-call traces for every arm, excluding the 13 on which a baseline agent hit the two-hour limit without writing a trace).

Table 11: Output tokens per instance on the paired sample; ratios are relative to BF16.

Benchmark	Arm	n	Mean	Median	p90	Mean/BF16	p90/BF16
SWE-bench Verified	BF16	337	8,298	6,789	11,994	1.00	1.00
	SealKV (K4V4)	337	9,342	6,994	15,673	1.13	1.31
	SAW-INT4	337	9,630	7,067	17,970	1.16	1.50
	NVFP4	337	9,574	6,874	22,882	1.15	1.91
τ^2-bench airline	BF16	200	2,310	1,224	6,075	1.00	1.00
	SealKV (K4V4)	200	2,575	1,292	6,375	1.11	1.05
	SAW-INT4	200	3,029	1,326	9,685	1.31	1.59
	NVFP4	200	2,585	1,258	8,169	1.12	1.34

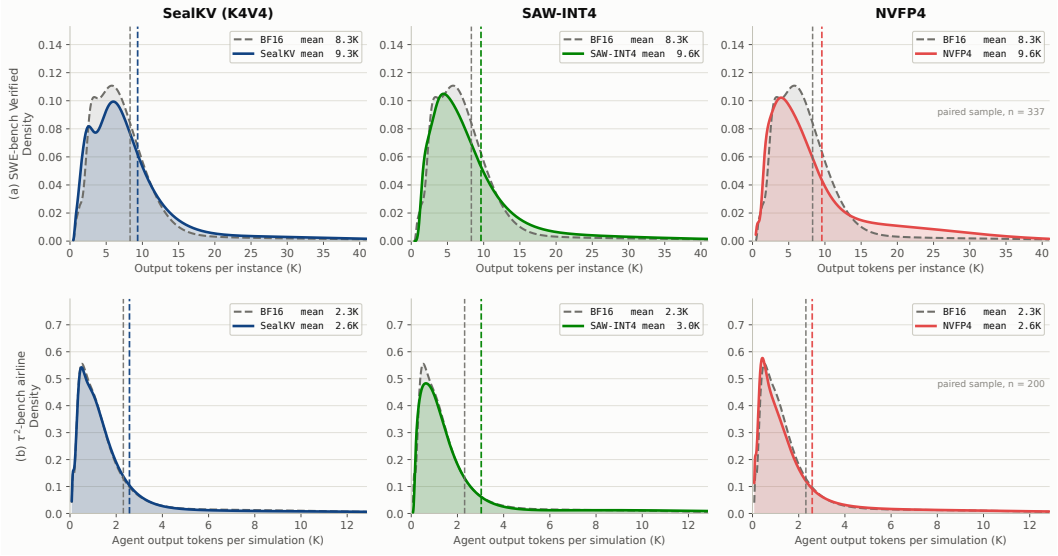


Figure 9: **Output-token distributions on the agentic benchmarks** (Qwen3-Coder-30B-A3B). Each panel shows one quantized arm (solid) against the BF16 reference (dashed); dashed vertical lines mark means. (a) SWE-bench Verified, output tokens per instance summed over agent steps (4,096-token per-turn cap). (b) τ^2-bench airline, agent output tokens per simulation (user-simulator tokens excluded). The horizontal axis is truncated at the 99th (a) and 96th (b) percentile.

E Ablations of precision allocation and preprocessing

We examine K/V bit allocation, the components of SealKV, and the precision of the stored block mean. All ablations use the reference quantizer on Qwen3-8B with the WikiText-2 protocol of Section 4.2: 65,504 scored positions per arm and teacher-forced KL divergence to BF16. Every arm keeps the 256-token BF16 window and seals blocks of $B = 16$ tokens; error bars in Figures 10 and 12 give the per-token standard error.

E.1 Allocating bits to keys and values

At the same stored budget, assigning more bits to keys yields lower KL divergence than assigning them to values (Figure 10): K3V2 gives 7.2 vs. 15.6×10^{-3} for K2V3, K4V2 gives 4.6 vs. 15.0×10^{-3} for K2V4, and K4V3 gives 2.3 vs. 3.8×10^{-3} for K3V4. At 3.41 bits/value, K4V2 (4.6×10^{-3}) and uniform K3V3 (5.0×10^{-3}) are close.

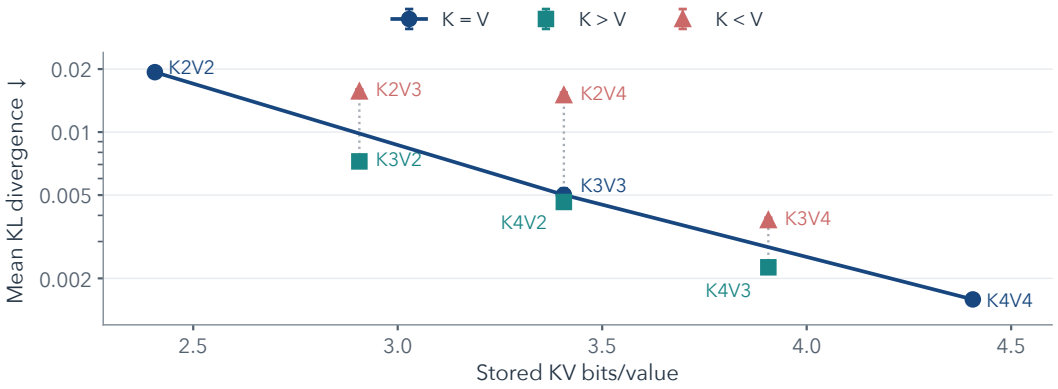


Figure 10: **Extra bits help keys more than values.** Dotted lines connect equal-budget allocations; lower KL divergence is better. Bits/value includes metadata; every point uses NVFP4 block means.

E.2 Contributions of individual components

Figure 11 cumulatively enables sink protection, BMS, RHT, and CPTC, normalizing KL divergence to the scalar INT baseline at each bit width. INT is symmetric round-to-nearest with one absmax scale per token and head, charged as BF16. INT and +Sink store $b + 0.125$ bits/value in the coded region; BMS adds 1.0, 0.5, or 0.28 bits/value with BF16, NVFP8, or NVFP4 means, and RHT and CPTC add none, so the +RHT and +CPTC bars are rate-matched. The complete codec applies BMS in the rotated basis. At four bits, protecting 128 sink tokens lowers divergence to 48.0% of baseline, BMS to 35.6–38.7%, RHT to 7.2–7.7%, and CPTC to 4.9–5.1%. At two bits, the complete recipe lowers divergence to 2.1–2.2% of baseline. With BF16 means, the complete recipe gives $46.6\times$ and $20.3\times$ lower KL divergence at two and four bits, respectively.

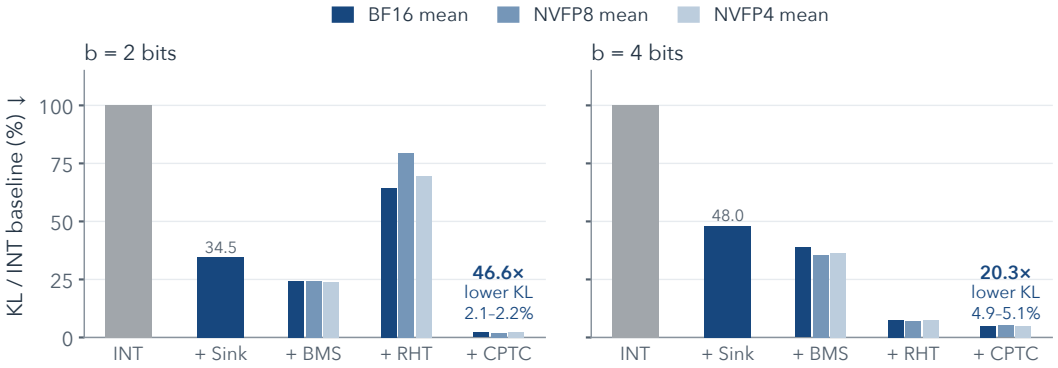


Figure 11: **CPTC completes the gains from preprocessing.** Cumulative component ablation, with KL divergence expressed as a percentage of the corresponding INT baseline. Sink protection retains 128 tokens.

Rotation and trellis coding. Table 12 holds BMS (BF16 means) and sink protection fixed and crosses RHT with the quantizer, so each column pair is rate-matched. With RHT, CPTC gives the lowest divergence at both widths: 1.53×10^{-3} at four bits and 1.92×10^{-2} at two bits. With the same preprocessing and rate, the non-uniform scalar codebook NF4 gives 2.07×10^{-3} at four bits; at two bits, calibrated OSCAR, with its own rotation and 64 sink tokens, gives 3.8×10^{-2} at 3.00 bits/value versus CPTC's 1.9×10^{-2} at 3.125. With RHT and sink protection fixed, BMS lowers divergence $2.4\times$ for CPTC at four bits and $90\times$ at two bits.

Table 12: KL divergence ($\times 10^{-3}$) with BMS (BF16 means) and 128 sink tokens held fixed.

	$b = 2$ (3.125 bits/value)		$b = 4$ (5.125 bits/value)	
Rotation	INT	CPTC	INT	CPTC
None	216.3	92.1	12.06	30.66
RHT	573.9	**19.2**	2.31	**1.53**

E.3 Precision of the block mean

Replacing BF16 means with NVFP8 or NVFP4 means shifts the storage–quality curve toward fewer bits/value with little change in KL divergence (Figure 12); at each payload setting ($b = 2, 4, 6$), all three formats stay extremely close to each other. The integrated backend's INT4 mean records give 1.51×10^{-3} at $b = 4$ (Appendix H).

F CPTC reconstruction details

CPTC (Anonymous, 2026) is submitted concurrently to ICLR 2027; its anonymized draft is attached as supplementary material. We restate CPTC's reconstruction and codebook construction here.

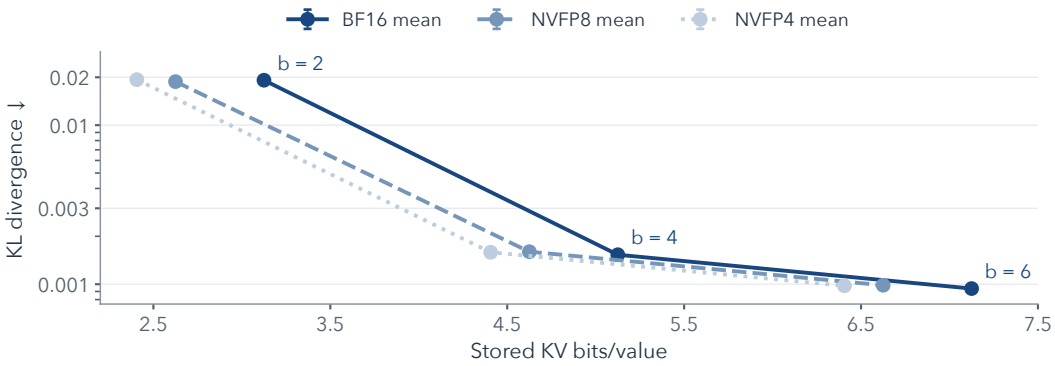


Figure 12: **Low-precision means retain nearly the same KL divergence.** BF16, NVFP8, and NVFP4 mean records at payload settings $b = 2, 4, 6$. The horizontal shift reflects metadata savings.

CPTC CODEBOOK CONSTRUCTION AND ENCODING

For a T-value trellis with b payload bits per value and window width $b \le L \le 16$, write the bT-bit code as $c[0], \ldots, c[bT-1]$ in least-significant-bit-first order. The decoder forms the circular window

$$w_d(c) = \sum_{i=0}^{L-1} 2^i \, c[(bd+i) \bmod bT], \qquad d = 0, \ldots, T-1. \tag{8}$$

Wrapping each window into the beginning of the same bitstream eliminates transmitted initial-state metadata (Figure 14). The reduced trellis has 2^{L-b} states and 2^b outgoing branches per state.

CPTC maps each window w to a reconstruction value through two fixed 16-bit constants a_0, a_1 (Anonymous, 2026):

$$x_0 = ((a_0 w) \bmod 2^{16}) \gg 10, \quad x_1 = ((a_1 w) \bmod 2^{16}) \gg 10, \quad h = x_0 + x_1 \in \{0, \ldots, 126\}. \tag{9}$$

For $h < 64$, reconstruction uses entry h of a 64-entry table; otherwise it uses the negated entry $126 - h$. The sign change is a sign-bit XOR in the E2M5 byte representation. The reconstructed value is then multiplied by the scale of its token/head vector.

The table is constructed once for each (L, a_0, a_1) by enumerating all 2^L windows and converting the resulting histogram of h into cumulative-probability cells. Symmetrizing the cell boundaries about 0.5 gives paired entries equal and opposite values. Each table entry is the conditional mean of the target distribution within its cell: for a standard normal Z, this is $\mathbb{E}[Z \mid \Phi(Z) \in [c_{\text{lo}}, c_{\text{hi}}]]$. The values are multiplied by two to target $\mathcal{N}(0, 4)$ and rounded to the nearest representable byte. The constants are selected by exhaustive search for low mean-squared error on the target density, independently of model data. For $L = 10$, they are $a_0 = \texttt{0x39C0}$ and $a_1 = \texttt{0x4E00}$, corresponding to 231 and 312 shifted left by $16 - L$ bits. The serving decoder precomputes the complete 1,024-entry mapping in BF16; the compact 64-byte construction and the expanded lookup table represent the same mapping.

During prefill, our fused Viterbi encoder scales, codes, and packs all sealed rows of a layer at once, producing codes bit-identical to those of CPTC's reference kernel. The rotation and codebook are fixed independently of the model, while means and scales are estimated from the cache being compressed, so SealKV requires neither an offline calibration corpus nor model-specific codebook fitting. The fused decode kernel consumes reconstructed values using BF16 tensor-core products with FP32 accumulation.

G SUPPORTING ILLUSTRATIONS AND REPRODUCTION SETTINGS

Effective-bit comparisons. Table 1 uses four payload bits per value and excludes scales and other metadata. Lloyd-Max denotes the optimal scalar quantizer for this source; QTIP uses its 1MAD/3INST construction. The QTIP and RPTC results are our reproductions, and CPTC at

$L = 10$ uses 10^6 Gaussian samples with $T = 128$. Scalar reconstruction uses a 16-value grid; QTIP uses arithmetic, RPTC at $L = 16$ uses a 2^{16}-entry table, and CPTC at $L = 16$ uses arithmetic plus a 64-byte table. The selected $L = 10$ CPTC decoder stores all 1,024 BF16 values in a 2 KiB lookup table. SealKV keeps $L = 10$ and the same 1,024-entry codebook at every payload width: the fused kernels implement $b \in \{2, 4\}$ (K4V4, K4V2, K2V2), and the reference quantizer also runs $b = 3$ and $b = 6$. For a fixed boundary the reduced trellis has 2^{L-b} states (256, 128, 64, and 16 at $b = 2, 3, 4, 6$) with 2^b branches each, so every coded value costs $2^L = 1{,}024$ transitions per Viterbi pass at every width.

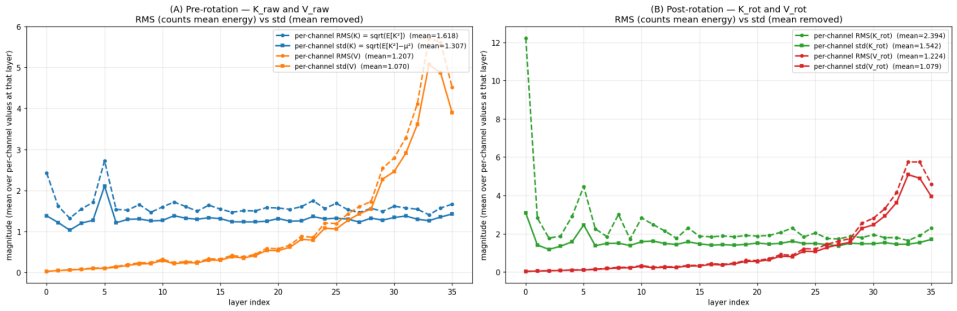


Figure 13: Per-channel RMS (circles) versus standard deviation (squares) per layer over all tokens of 24 Qwen3-8B AIME-25 traces; the gap is the evidence of a per-channel mean, whose energy is the difference of squared statistics (Equation 3). Before rotation, RMS/std $= 1.618/1.307$ (K) and $1.207/1.070$ (V); after rotation the key-side averages are $2.394/1.542$.

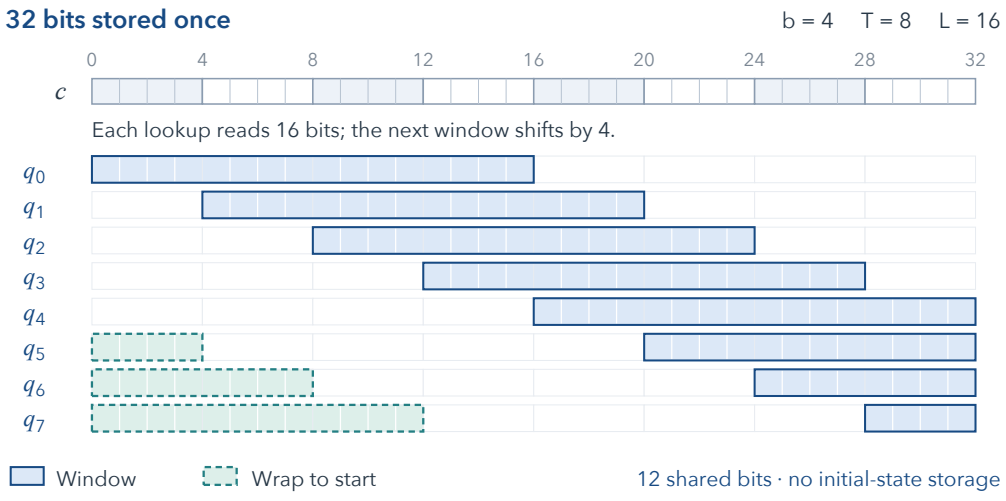


Figure 14: The bitshift trellis's circular sliding window (Mao & Gray, 2010; Anonymous, 2026), illustrated for b=4, T=8, L=16. Each quantized value q_i is reconstructed from a 16-bit window of the block's bitstream; the window advances by b bits per value, so consecutive values share $L - b = 12$ bits of state. The last $L/b - 1$ windows wrap tail-biting into the block's initial bits (dashed), so a T-value block costs exactly bT bits with no transmitted initial state.

Table 13: Query/key/value normalization in widely used open-weight LLMs. The normalization *type* also varies (per-head RMSNorm, parameter-free ℓ_2, latent normalization in MLA architectures). SealKV estimates its per-token scale, a standard deviation, from each mean-subtracted rotated vector, so it does not depend on this choice (Section 3.2).

Model	q	k	v	Model	q	k	v
Qwen3 (8B/32B/235B/Coder)	✓	✓	—	Llama 3.1 / 3.3	—	—	—
Gemma (2026)	✓	✓	✓	Phi-4	—	—	—
Llama 4 Scout (16E)	✓	✓	—	NVIDIA Nemotron	—	—	—
GLM-4.5 / 4.6 / 4.7	✓	✓	—	GPT-OSS (20B/120B)	—	—	—
MiniMax M3	✓	✓	—				

H HYPERPARAMETERS AND IMPLEMENTATION

Codec and supported shapes. The integrated backend uses a 16-token mean-sharing block, 128 protected prefix tokens, and a 256-row BF16 ring. Its CPTC configuration is $L = 10$ and $T = D = 128$ with $(b_K, b_V) \in \{(4,4),(4,2),(2,2)\}$, each setting having its own fused decode kernel, and block means use $\beta_\mu = 4$; the codebook contains 1,024 BF16 entries with 127 distinct values, mean zero, standard deviation 2.0008, and range ± 6.75. The sign vector for $R = H_D \operatorname{diag}(s)$ is generated with seed zero and has exactly $D/2$ signs of each value. The GPT-OSS experiments use $D = T = 64$ in the reference quantizer; their stored budget is 4.52 bits/value, Equation 5 with $b = 4$, $D = 64$, $B = 16$, $\beta_\mu = 4$, and a densely stored record ($\pi = 0$); the full-attention layers are sealed, and GPT-OSS's 128-token sliding-window layers stay in BF16. The kernel sweep and serving measurements of K4V4, K4V2, and K2V2 (Figures 1c and 8; Tables 8–10) use these fused kernels; uniform three- and six-bit codes and the K3V2, K2V3, K2V4, K4V3, and K3V4 allocations of Appendix E.1 run in the reference quantizer. The profiling KL sweep uses NVFP4 means, and the integrated backend uses INT4 means with BF16 scales and padded records. At $D = 128$, with its padding, the INT4 record costs $\beta_\mu + 16/D + \pi = 4.5$ bits per channel, the same as an NVFP4 record, so both report the same bits/value.

Mean records and residual scale. For each completed block and each role (K or V), the integrated codec computes the per-channel mean μ_j of block j with a fixed pairwise FP32 summation tree. It stores an INT4 offset-binary vector with one BF16 scale:

$$\begin{aligned} m_j &= \mathrm{BF16}(\|\mu_j\|_\infty/7), \\ u_{j,d} &= \mathrm{clip}\big(\mathrm{round}(\mu_{j,d}/\max(m_j, 10^{-30})) + 8, 0, 15\big), \\ \widehat{\mu}_{j,d} &= m_j(u_{j,d} - 8). \end{aligned} \tag{10}$$

A mean whose scale rounds to zero stores $m_j = 0$ and $u_j = 8$ and reconstructs to zero. Subtracting the stored $\widehat{\mu}_j$ ensures that encoding and reconstruction use identical offsets, so the mean-rounding remainder is included in the residual being coded. Recomputing a completed block's record is bitwise idempotent, so decode can rewrite records produced during prefill.

For a residual row $e = x' - \widehat{\mu}_j$ with channel mean $\bar{e} = D^{-1}\sum_d e_d$, the normalization uses the population standard deviation $\rho = \max(\sqrt{D^{-1}\sum_d (e_d - \bar{e})^2}, 10^{-12})$, i.e., $\epsilon = 10^{-12}$ in the per-token scale of Section 3.2. The encoder quantizes $2e/\rho$ in FP16 and stores the scale $\mathrm{BF16}(\rho/2)$, i.e., $\sigma_{\mathcal{C}} = 2$. The normalized row therefore has standard deviation two, the codebook's target, whatever the query/key/value normalization of the host model (Table 13).

At $D = 128$, the K and V mean vectors occupy 64 bytes each and their two BF16 scales occupy four bytes, so 132 bytes of content sit in a 144-byte shared record per 16-token block; the 12 bytes of padding are the $\pi = 0.375$ bits per channel of Equation 5. Each token and head additionally stores 64 bytes of K codes, 64 bytes of V codes, and two BF16 residual scales. The coded-region budget is consequently

$$r_{\mathrm{sealed}} = \frac{8\big(64 + 64 + 4 + 144/16\big)}{2 \cdot 128} = 4.40625 \text{ bits/value}. \tag{11}$$

Substituting $b = 4$, $D = 128$, $B = 16$, $\beta_\mu = 4$, and $\pi = 0.375$ into Equation 5 gives the same rate. This is 141 bytes per token/head for K and V together, compared with 512 bytes in BF16, a $3.63\times$ reduction; K4V2 stores 109 bytes per token/head (3.41 bits/value) and K2V2 77 bytes (2.41).

Frame boundaries and sealing schedule. Let N be the true context length, $S = 32\lceil N/32 \rceil$, and $M = S - 256$ for $N \geq 384$. The three disjoint reader regions are

$$\underbrace{[0, 128)}_{\text{BF16 prefix}}, \qquad \underbrace{[128, M)}_{\text{sealed interior}}, \qquad \underbrace{[M, N)}_{\text{BF16 tail}}. \tag{12}$$

Frame positions at least N are masked. The dense tail contains 225–256 tokens, and the ring row for absolute position t is $(t - 128) \bmod 256$. When $t = N - 226 \geq 128$, the decoder encodes absolute position t, which lies in a completed block still in the ring; its middle-store row is $N - 354$. The backend realizes the whole-block sealing of Algorithm 1 with an equivalent per-position schedule: it codes one position per step, $t = N - 226$, against its completed block's mean, whose record it recomputes bit for bit at each of the block's steps, so every stored code, scale, and record is identical to whole-block sealing. In the notation of Algorithm 1, the served seal distance is $\Delta = 226 = W - \tau + 2$ with stage $\tau = 32$, which seals every row of the stage-rounded middle $[128, M)$ for all N. A readiness invariant ensures that every row exposed to the sealed reader already has codes, scales, and its block-mean record. Contexts shorter than 384 tokens are handled as dense frames.

Encoder execution. Layer 0 launches the cross-layer sealing path after appending its newest rotated K/V rows and before its own attention, on the same stream and inside the captured CUDA graph. One Triton mean kernel gathers a completed 16-row block per layer, sequence, and head and emits the ready token's residual. One batched encoder call per payload width encodes the ready rows across layers, keys, and values, and scatter operations store the result.

H.1 Kernel implementation

We describe the K4V4 decode kernel for head dimension 128. The K4V2 and K2V2 kernels differ only in how they stage and window their two-bit code streams; they share its schedule, mean-record path, online softmax, and epilogue.

Every warp runs the same loop: it stages the codes of 32 tokens, looks up their reconstructions, multiplies them with `mma.sync`, and updates an online softmax. The kernel launches one block of 12 warps per SM (132 on H100) and double-buffers its 32-token stages.

Each warp owns two staging slots, each holding 32 tokens of packed K and V codes and their BF16 scales. The slots are filled with `cp.async` and XOR-swizzled so that every `ldmatrix` tile spans all 32 shared-memory banks.

The reconstruction table holds the 1,024 CPTC values, replicated once per lane in shared memory at $\mathrm{addr}(i, \lambda) = \mathrm{base} + 128i + 4\lambda$, so every lane reads from its own bank. Codes cross word and lane boundaries, so shuffles supply the neighboring symbols of each 10-bit window; for four-bit symbols $\xi_{t,d}$ of one token/head vector,

$$w_{t,d} = \xi_{t,d} + 16\xi_{t,(d+1) \bmod D} + 256\big(\xi_{t,(d+2) \bmod D} \bmod 4\big), \qquad \bar{x}_{t,d} = \mathcal{C}_{10}[w_{t,d}]. \tag{13}$$

A two-bit code stream (V in K4V2; K and V in K2V2) forms the same 10-bit window from five consecutive symbols, $w_{t,d} = \sum_{i=0}^{4} 4^i\, \xi_{t,(d+i) \bmod D}$, and indexes the same table. Each lookup is a shift, a mask, and a shared-memory load. The two looked-up BF16 values are packed into one `mma.sync` operand register, and no dequantized value is written to shared or global memory.

The kernel loads the INT4 mean records directly into registers with `ld.global.nc.v4` and prefetches the next stage's records into L2. A stored mean code u is read as the BF16 value $128 + u$ (bit pattern `0x4300` $\mid u$), so $\widehat{\mu} = m[(128 + u) - 136]$ and both mean terms run on the tensor cores. The key correction $m[\sum_d q'_d(128 + u_d) - 136 \sum_d q'_d]$ is computed for eight blocks per `mma` pass and folded into the per-token multiply-add. The value correction is one PV-shaped `mma` per block whose extra k-slots carry -136.

Sink and window blocks stream BF16 from global memory directly into `mma` fragments without staging. The online softmax runs in the log2 domain with `ex2.approx.ftz`. Sequence lengths

are read from device memory, so a captured CUDA graph replays correctly as contexts grow or become ragged.

The 32-token stages of all (batch, head) pairs are laid end to end and split into equal contiguous runs over the 132×12 warps. A run that crosses a (batch, head) boundary keeps a separate softmax state for each segment, and a small merge kernel combines the partial outputs and log-normalizers.

Profile. Under the protocol of Appendix C, at Qwen3-8B, batch 64, and 32K context, the K4V4 kernel reads 2.44 GB in 1,065 μs, 2.29 TB/s or 68% of HBM peak. It is issue-bound: NCU reports 72.5% issue-slot utilization, IPC 2.90 of 4, and 0.3% shared-memory bank conflicts, with about 1,314 instructions per 32-token stage per warp, of which 768 (58%) are the three-instruction lookup, 128 are packs, and 58 are `mma`. Consistently, its time scales with the SM clock while memory-bound FA3's does not: at a fixed 1,500 MHz, K4V4 is flat at 1.95–2.02 TB/s from 4K to 256K. The remaining gap to the HBM roof is a property of this instruction mix, recoverable only by decoding a value in fewer instructions.

H.2 Serving scope

The backend supports device-indexed slot mapping, CUDA graph replay, and continuous batching as requests join, finish, and reorder.