ABRIDGEKV: COMPRESSING LONG REASONING INTO THE KV CACHE OF LEARNED PAUSE TOKENS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long chain-of-thought decoding is memory-bound: the KV cache of the generated tokens grows linearly with the test-time compute being spent. Evicting cache entries does not fix this, because a model never trained on a cache with entries removed generates longer, often up to the length limit. We instead train the model to compress its own output. ABRIDGEKV writes each completed block of generated text into the KV entries of a few learned pause tokens and frees the block's cache, trained end to end with the next-token loss. Three novel choices make this work: the summary rows take positions inside the block they replace, the row budget is spread across layers by each layer's measured value, and the blocks are large (512 tokens). On 446 competition problems with 32K-token generations, keeping a quarter of the generated cache matches an identically trained full-KV model in accuracy and output length; the strongest published evictor at a matched budget writes far longer, and the closest trained compressor, retrained on the same data, loses 10 points. Served in vLLM, the cache is $3.5\times$ smaller per sequence and a saturated H100 finishes a problem $3.2\times$ faster than with full KV.

1 INTRODUCTION

Test-time scaling improves reasoning accuracy by generating longer chains of thought (CoT) (OpenAI, 2024; DeepSeek-AI, 2025; Muennighoff et al., 2025; Snell et al., 2024). On competition mathematics, even small models generate tens of thousands of tokens per response and frequently reach the maximum generation length. Decoding such long CoTs is memory-bound: the KV cache grows linearly with the number of generated tokens, and its size, not the model weights, bounds the batch size and hence throughput (Kwon et al., 2023). Unlike long-prompt settings, this cache is produced by the model's own decoding and grows with exactly the test-time compute being spent. We target the KV cache of the generated tokens.

KV eviction keeps a fixed number of cache entries and discards the rest, selected by recency or accumulated attention score (Zhang et al., 2023; Xiao et al., 2024b; Li et al., 2024); several recent methods specialize it to long reasoning (Cai et al., 2025; Song et al., 2025; Zhang et al., 2026; Chang et al., 2026). Two properties limit it. The model is never trained on a cache with entries removed, and the selection is made from past attention, whereas a CoT reuses intermediate results at unpredictable later points; an evicted entry leaves no representation behind. Empirically the failure appears as longer generations: at a quarter of the cache, simple policies reach the 32K limit on 38–58% of samples, and R-KV, the strongest published method here, generates 40–69% more tokens than the uncompressed model (Figure 4a), which offsets the per-token memory saving. This motivates training the model to reason over a compressed history.

Learned compression (Mu et al., 2023; Ge et al., 2024b; Zhang et al., 2024a) was developed for fixed inputs, such as long documents, compressed once before decoding. Compressing a CoT differs: the model must compress its own output online, dozens of times per response, and condition on the compressed states thousands of tokens later, so each summary has to be written without knowing which of its contents the later reasoning will need. MEMENTO (Kontonis et al., 2026), which motivated this work, segments the reasoning and conditions on text summaries of the segments. LightThinker (Zhang et al., 2025) compresses each thought, delimited by a paragraph break (about 50 tokens on our data), into 9 learned KV entries, evaluated with outputs of at most 10K tokens. At long generation lengths this compression unit is too small: a 30K-token response under-

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.

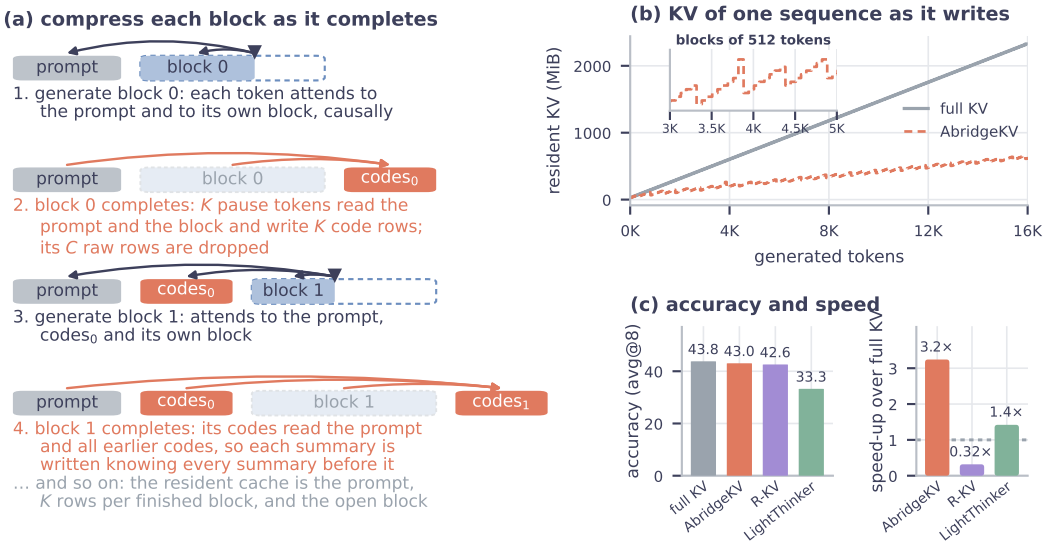


Figure 1: (a) ABRIDGEKV writes in blocks of C tokens; when a block completes, K pause tokens write its code rows and its raw rows are freed. (b) KV of one sequence in vLLM: it rises through each block and falls at each compression. (c) 446 competition problems, 32K cap: accuracy, and speed-up in time to finish over full KV (dotted) on a saturated H100.

goes several hundred lossy compressions, and only the current thought, a few dozen tokens, is kept uncompressed. Its compressed entries also take new positions after each thought, which we find costly in isolation. Retrained on our data and served in our engine, it is 10 points below the full-KV model.

ABRIDGEKV compresses the generated KV cache in fixed blocks of 512 tokens into learned KV entries (Figure 1a). When a block completes, a set of pause tokens attends to it and writes a summary into their own KV entries at every layer, which we call code rows; the block's KV is then freed, and all later tokens attend to the code rows instead. Training is end to end with the next-token loss, so the gradient of later tokens shapes each summary. Two design choices are central. First, code rows are assigned positions inside the block they replace rather than after it, as in prior learned summaries: relative distances between later queries and a summary match those of the original block, so pretrained attention patterns transfer, and positions never exceed the response length. Second, the row budget is allocated non-uniformly across layers: the value of code rows is concentrated in a few layers, so we measure each layer's sensitivity and reallocate rows at a fixed mean; the same mean spent uniformly loses significantly against full KV, and the reallocation recovers it. We call this the *per-layer* setting, in contrast to the *uniform* setting with 128 rows in every layer. With the block size, these choices support one principle: a compressed representation works best when it matches the tokens it replaces, in position, in the layers that use it, and in leaving recent context uncompressed. Fixed per-layer budgets map to cache groups in a paged engine, so the saving is physical memory in vLLM.

Our contributions are as follows.

1. We introduce ABRIDGEKV, in which a reasoning model **compresses its own chain of thought into the KV cache of learned pause tokens** as it writes, trained end to end and served as physical memory in a paged engine (Section 2).
2. We identify **three novel design choices**: in-block code positions (no prior learned summary uses them), a layer budget set by measured value (new to learned compression) and large blocks (the nearest prior method compresses per thought), and show that code rows carry their block's content many blocks later (Sections 4.4–5).
3. At a 32K cap, ABRIDGEKV **matches full KV with a** $4\times$ **smaller persistent cache and finishes a problem** $3.2\times$ **faster** in our tuned vLLM, where eviction writes far longer and LightThinker loses 10 points (Section 4).

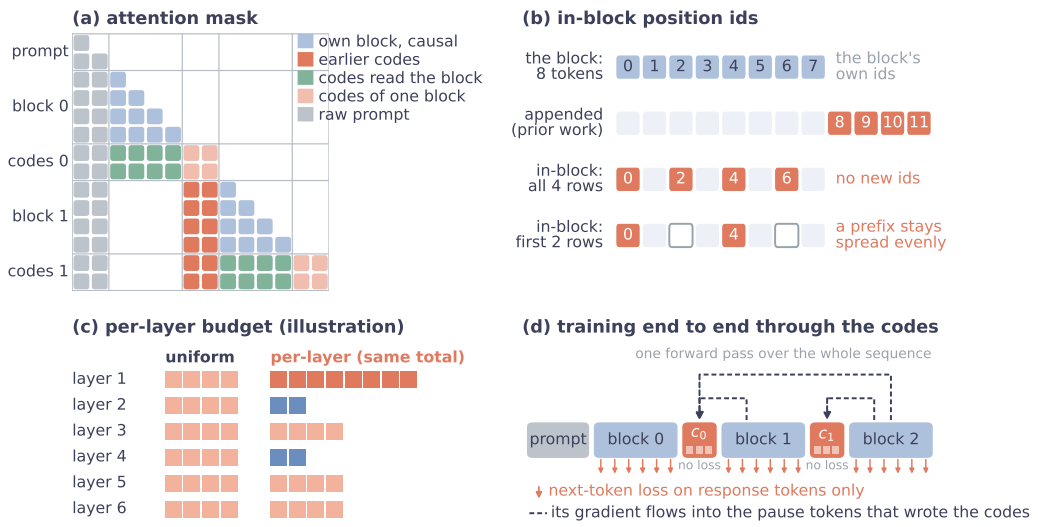


Figure 2: Method details. (a) Attention mask for two blocks (C=4, K=2, 2-token prompt). (b) Code row k takes position $\mathrm{bitrev}(k) \cdot C/K_{\max}$ inside the block it replaces (C=8, $K_{\max}$=4), not after it as in prior work; any prefix of the rows is an even grid. (c) Per-layer quotas keep the uniform mean. (d) Training is one forward pass under mask (a), with next-token loss on response tokens only; later blocks' loss reaches the pause tokens that wrote their codes.

2 Method

2.1 Block layout and attention mask

Layout. The prompt (P tokens) is kept raw and never compressed. The response is generated in blocks of C=512 tokens; block $b \in \{0, 1, \dots\}$ has content $x_{b,0}, \dots, x_{b,C-1}$ and $K_{\max}$=256 *code slots* $p_{b,0}, \dots, p_{b,K_{\max}-1}$, pause tokens with learned input embeddings. When block b completes, the $K_{\max}$ pause tokens are run through the model in one forward pass; their KV rows at every layer are the block's *codes*. The block's content KV is then dropped. Nothing else is added to the model: no adapters, no extra layers, no change to the rotary embedding.

Visibility. A content query $x_{b,t}$ attends to the prompt, to the codes of every earlier block $b' < b$, and to its own block causally, $x_{b,<t}$. A code query $p_{b,k}$ attends to the prompt, to the codes of earlier blocks, to all of its block $x_{b,0..C-1}$, and to the other codes of its block. Content of block b is never visible to any query outside block b (Figure 2a). This mask is what the trainer enforces on the full sequence and what the runtime enforces by eviction.

Positions. Content keeps its uncompressed numbering, and each code row sits *inside* the block it summarizes:

$$\mathrm{pos}(x_{b,t}) = P + bC + t, \qquad \mathrm{pos}(p_{b,k}) = P + bC + \mathrm{bitrev}_{K_{\max}}(k)\,\frac{C}{K_{\max}}, \tag{1}$$

where $\mathrm{bitrev}_{K_{\max}}(k)$ reverses the $\log_2 K_{\max}$ bits of k (for $K_{\max}$=8 the slot order is $0, 4, 2, 6, 1, 5, 3, 7$). Query and key use the same rotation, so this is a choice of position ids only. Three properties follow. *(i) No position inflation.* A response of T tokens occupies exactly T positions whatever $K_{\max}$ is, so a 32K-token response stays inside the 32,768 positions the base model was trained on. Giving the codes fresh positions after their block, as the learned-summary literature does, stretches the same response over $T(C{+}K)/C$ positions. *(ii) Nested, K-independent quotas.* Keeping the first q slots in bit-reversed order, $k < q$, keeps an even grid of stride C/q over the block (Figure 2b): every quota level sees an evenly spaced summary, quota q is a prefix of quota $2q$, and the layout of the kept rows does not depend on $K_{\max}$, so a per-(layer, head) budget is a mask on one trained model rather than a different model. *(iii) Stable relative geometry.* A later query sees

each code at roughly the distance the content it replaces would have had, rather than at a distance that grows with the number of intervening summaries. Ablating only this choice, holding the mask, budget, trainer and data fixed, costs 6.2 $[4.6, 7.8]$ avg@8 at 32K, and not through extrapolation (Section 4.4). One consequence is worth stating: because a code query shares its key's position, it attends to the part of its block after its grid point at negative relative offset, which the pretrained weights have never seen; fine-tuning learns this, and it is where the in-block layout pays for its properties.

Quotas. Each layer ℓ keeps only the first $q_\ell \leq K_{\max}$ code rows of every completed block and masks the rest; at serving time only those rows persist (Appendix F). Section 2.3 sets the quotas.

2.2 Training objective

The model is trained with the ordinary next-token loss on every response token, computed while its history is held as codes (Eq. 2); the prompt is never compressed. The whole response, code rows included, is one forward pass under the block mask, so the loss of later blocks flows back into the pause tokens that wrote the codes they read (Figure 2d). We add no auxiliary terms: distillation from a frozen full-context teacher and keeping a raw tail were tried and did not help (Appendix C).

$$\mathcal{L}_{\mathrm{CE}} = -\tfrac{1}{|R|} \sum_{t \in R} \log p_\theta\big(x_t \mid \text{prompt},\ \text{codes}_{<b(t)},\ x_{b(t),<t}\big) \tag{2}$$

2.3 Allocating the budget across layers

The budget is a per-layer quota $q_\ell \leq K_{\max}$: layer ℓ keeps q_ℓ of the $K_{\max}$=256 code rows of every completed block and masks the rest, the same quota for every KV head of the layer (so tensor-parallel serving stays balanced and the physical cache can drop the masked rows, Appendix F). *Uniform* allocation sets q_ℓ=128 everywhere. *Per-layer* (dynamic) allocation keeps the same mean of 128 rows but moves rows between layers, in three steps:

1. Train a model with the uniform-128 schedule.
2. Measure what each layer's code rows are worth: at evaluation time, without retraining, halve one layer's quota (128 $\rightarrow$ 64) and record the increase in held-out cross-entropy; repeat for each of the 36 layers (Figure 6a). Most layers cost nothing; a few cost a lot.
3. Rank layers by cost: the 6 most expensive get 256 rows, the 12 cheapest 64, the other 18 keep 128 ($6{\cdot}256 + 18{\cdot}128 + 12{\cdot}64 = 36{\cdot}128$), and train a new model on this schedule.

The resulting schedule (Figure 6) raises layers 0, 15, 18, 20, 21 and 24 and cuts layers 1–6, 8, 10, 30–32 and 35. Section 5 explains what the measurement shows, why we use three levels and cut only a third of the layers, and why the quota is set per layer rather than per head.

3 Experimental setup

Every arm fine-tunes Qwen3-4B-Base on the same 2B response tokens of math SFT data (400K prompt/response pairs derived from OpenMathReasoning) in the same canonical order, with C=512, $K_{\max}$=256 slots and a mean budget of 128 code rows per block unless stated; each arm is served with the schedule it was trained with. Arms are named by their block-to-code ratio (this budget is 4:1; 2:1 and 8:1 keep 256 and 64 rows) and their allocation, uniform or per-layer; *appended positions* is the ablation whose code rows take new positions after the block. Full hyperparameters, hardware and throughput are in Appendix C. The reference is the same base model fine-tuned on the same tokens with full KV, the ceiling a compressed model can match but not beat; every arm is reported as a gap to it.

Evaluation. A 446-problem competition suite (twelve competitions from AIME, HMMT, BRUMO, CMIMC, SMT and AMC, 2022–2026; Appendix D), 8 samples per problem at T=0.6, top-p 0.95, top-k 20, seed 1234, and a 32K cap (31,744 generated tokens); a sample is correct if its last boxed answer matches or math-verify judges the full text equivalent. Every arm is scored on the same problems and seed, so differences are paired per problem, with a 2000-resample per-problem bootstrap. Pooling makes these comparisons resolvable: on one 30-problem set the 95% interval on

Table 1: Main comparison at the 32K cap. SFT arms: Qwen3-4B-Base, 2B response tokens, same data order, C=512. Suite: avg@8 over 446 problems, 95% bootstrap interval; Δ: paired difference against vanilla. KV/512: persistent KV rows per 512 generated tokens per KV head; R-KV's budget is constant, so its value is measured (Appendix E). MATH-500: avg@4, strict (eviction rows: HF, 1–3 samples per problem). trunc.: % of suite samples reaching the cap. † AIME 2025 and HMMT Feb 2025 only (60 problems), paired on those. Shaded: ours; bold: best below References.

Arm	KV/512	CE	MATH-500	Suite avg@8	Δ vs. vanilla	trunc.%
References						
Base model (no SFT)	512	–	69.6	13.5 [11.0, 16.2]	-30.4 [-33.4, -27.0]	5.6
Vanilla SFT (full KV)	512	0.4162	88.1	43.8 [39.8, 47.6]	–	3.4
Ours: dynamic per-layer allocation						
ABRIDGEKV (per-layer)	128	**0.4215**	87.5	**43.0** [39.1, 46.9]	**-0.8** [-2.3, +0.8]	**3.2**
Ablations: one design choice removed						
uniform allocation	128	0.4227	**88.3**	41.5 [37.6, 45.3]	-2.3 [-3.8, -0.9]	3.4
appended positions	128	0.4297	85.8	35.3 [31.6, 39.0]	-8.5 [-10.3, -6.8]	3.6
Prior learned compression, in our training and serving stack						
LightThinker (thought-level)	97	0.4391	83.4	33.3 [29.6, 37.0]	-10.5 [-12.5, -8.7]	5.4
Training-free eviction (vanilla weights)						
sink + recency	128	–	84.0	22.7 [14.6, 32.3]†	-7.1 [-10.4, -3.8]†	37.9
random	128	–	76.1	18.1 [11.0, 26.2]†	-11.7 [-16.5, -7.3]†	52.9
attention score (H2O)	128	–	59.1	10.0 [5.0, 15.8]†	-19.8 [-26.2, -13.3]†	57.7
R-KV, budget 2048	74	–	83.9	35.6 [31.7, 39.5]	-8.2 [-10.2, -6.3]	55.8
R-KV, budget 4096	144	–	86.5	42.6 [38.5, 46.6]	-1.2 [-2.7, +0.3]	31.7

a paired gap is about ± 5 points, and a full-KV re-run differing only in its data seed flips sign between AIME years; on the pooled suite the half-width is 1.5 (Appendix D.1). MATH-500 is avg@4 at a 16K cap; budget is persistent KV rows per 512 generated tokens per KV head ("KV/512" in Table 1).

Serving. Speed is measured in vLLM on one H100 with our patch for compressed decoding. A completed block's code rows are computed in the next engine step alongside other requests' decode tokens and its raw rows are released; a per-layer schedule is stored as one KV cache group per quota level, so rows a layer does not keep occupy no memory. Every arm runs under CUDA graphs and vLLM's asynchronous scheduling, except R-KV, whose port applies each compaction before the next step, and LightThinker (Appendix G). Time to finish is measured by replaying 512 requests at each arm's own suite lengths, run to completion at the concurrency cap that finishes fastest. The served model is the trained one: vLLM greedy tokens match the reference decoder, and trainer and incremental generator agree on per-token CE to 4×10^{-4} (Appendix J). Implementation and per-arm measurements are in Appendix F.

4 RESULTS

4.1 ACCURACY AND GENERATION LENGTH

At a quarter of the persistent cache, ABRIDGEKV reaches 43.0 avg@8 against 43.8 for full KV (Table 1, Figure 3a), a paired gap of -0.8 $[-2.3, +0.8]$ that lies inside the spread of a full-KV re-run differing only in its data seed (1 s.d. 0.8 points, Appendix C). On MATH-500, whose solutions are short, the two are 87.5 and 88.1.

Generation length. Matching accuracy could hide a model that writes more to make up for what it has lost. ABRIDGEKV does not: its length distribution is statistically indistinguishable from full KV's, as close as a full-KV re-run that differs only in its data seed (Kolmogorov–Smirnov distance 0.024 against 0.017; 0.032 is significant at 5%), and the same problems run long under both (per-problem correlation 0.93 against the re-run's 0.92; Figure 4a). R-KV at 4:1 is 0.33 away, 32% of its samples at the cap.

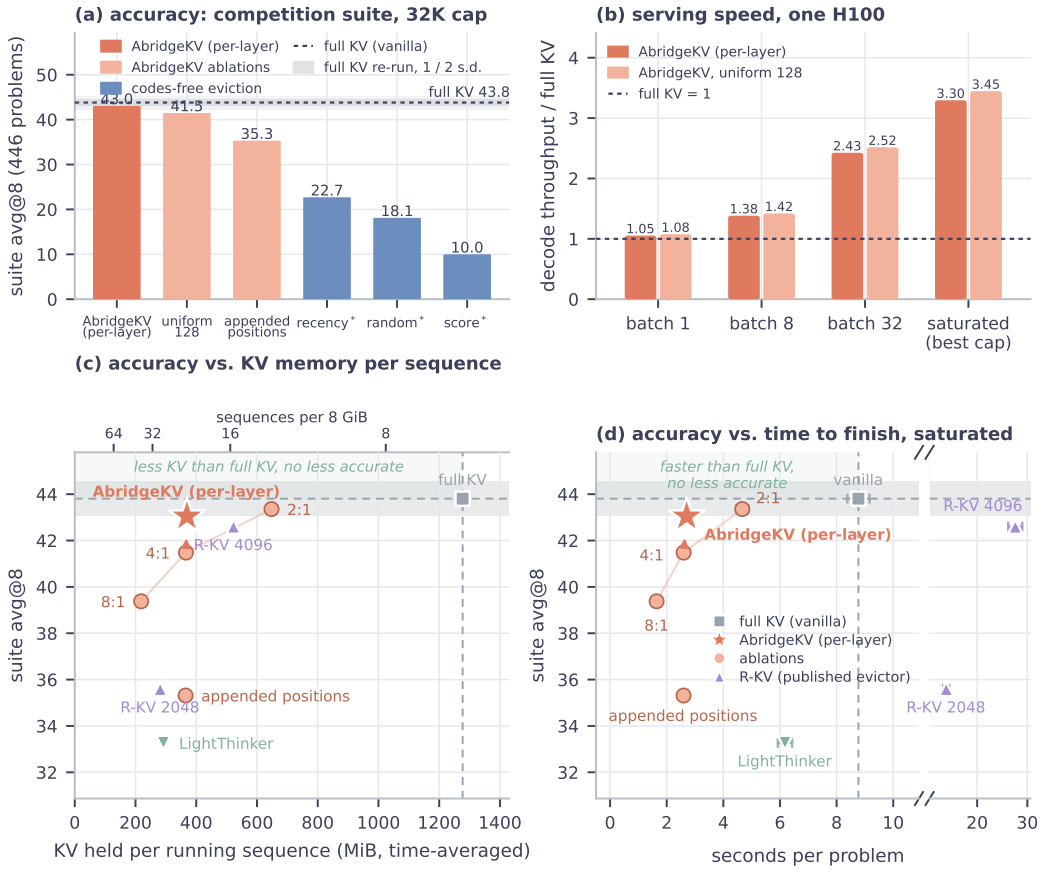


Figure 3: Main results. Orange circles are our arms and the orange star the per-layer model; grey squares full KV, purple triangles R-KV, green LightThinker; grey band: 1 s.d. of a full-KV re-run. (a) Suite avg@8, 32K cap (*60-problem subset). (b) Decode throughput over full KV, one H100, 16K tokens. (c) Accuracy against KV held per running sequence, time-averaged over each arm's generations; the arrow is the per-layer schedule at the uniform one's memory. (d) Accuracy against time to finish a problem: 512 suite-length requests run to completion at the fastest concurrency cap; shaded: at least as accurate as full KV and faster; the axis is broken to reach R-KV, whose horizontal bar extends to the time it would need if its scoring cost nothing. Other horizontal bars: 95% intervals on mean generated length.

4.2 Serving efficiency

AbridgeKV's resident KV per sequence is $3.54\times$ smaller than full KV's (the ceiling is $4\times$; the prompt and the open block stay raw), and because the per-layer schedule is a handful of cache groups on the stock paged path, it decodes as fast as the uniform setting: $1.05\times$, $1.38\times$ and $2.43\times$ full KV's throughput at batch 1, 8 and 32 (Figure 3b). Over a whole generation, a running sequence of AbridgeKV holds 369 MiB of KV on average against 1,277 MiB for full KV, the same as the uniform setting at 1.6 points higher accuracy, so an 8 GiB cache holds 22 such sequences instead of 6 (Figure 3c). Figure 3d then charges each arm for what it generates: saturated, AbridgeKV finishes a problem in $0.31\times$ full KV's time, the same as the uniform setting ($0.30\times$). It is the only compressed arm in the region that is at least as accurate as full KV and faster; R-KV (Section 4.3) sits to the right of full KV because it writes more and pays a compaction every 64 decode steps.

4.3 Comparison with eviction

Training-free eviction at the same budget does not come close, which is expected for two reasons. Eviction selects entries by the attention they have received so far, but a CoT reuses intermediate re-

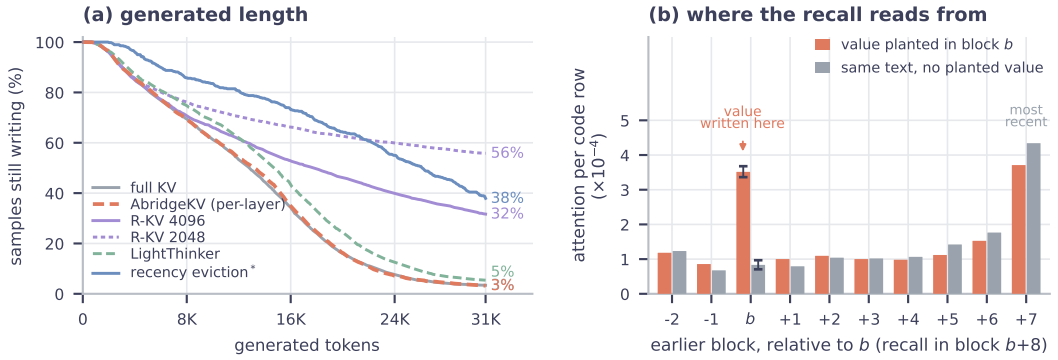


Figure 4: (a) Share of samples still writing at each length, and truncated at the cap (*60-problem subset). (b) A value planted in block b ("$Q_7 = 7391$") is recalled 8 compressions later: attention of the recalled digits to each earlier block's code rows, per row, over heads and layers (93 chains; 95% interval at b), with and without the planted value.

sults at later points that past attention does not predict; and an evicted entry leaves no representation, whereas a learned code row retains a summary of every block in a form the model is trained to read. When a needed result has been evicted, the model tends to recompute it, and generations lengthen.

Keeping 128 of each completed block's 512 rows of the full-KV model, by recency, at random, or by accumulated attention score, loses 7–20 points on their 60-problem subset, and the truncation column of Table 1 shows the mechanism: 38–58% of samples run to the 32K cap. R-KV (Cai et al., 2025), the published evictor designed for long reasoning, which we port to our serving stack and run on the same checkpoint, problems, sampling and grader, is the strongest of the family. We run it in its servable form, one kept set shared by all heads as in its authors' own SGLang port, since a paged cache cannot keep different tokens per head (Appendix E). With residency matched in its favour it is within noise of AbridgeKV at 4:1 (-0.4 $[-2.1, +1.2]$) and loses at 8:1 (-3.8 $[-5.5, -2.2]$ against our 8:1 arm). It changes the output in the same way: 32% and 56% of its samples reach the cap and it writes 40–69% more tokens than full KV (Figure 4a), so the memory it saves per token is spent on extra tokens and it takes longer than full KV to finish a problem (Figure 3d). The full comparison, including the port's parity check, is in Appendix E.

4.4 Ablations

The ablation rows of Table 1 remove one design choice at a time, holding trainer, data, order, block size and mean budget fixed; Appendix H gives the full block-size and budget sweep.

Code positions. Giving each code row a fresh position after its block, the convention of learned summaries (Zhang et al., 2024a; Chevalier et al., 2023; Mu et al., 2023; Zhang et al., 2025), costs 6.2 $[4.6, 7.8]$ avg@8 against the in-block layout, the largest effect we measure. It is not position inflation: appended rows stretch a 32K response past the base model's trained range only after 26,214 generated tokens, which 6.3% of samples reach, and restricting the comparison to the 256 problems where no sample of either arm gets there leaves the gap unchanged at 6.2 $[4.1, 8.3]$. An in-block row keeps the distance to later queries that its content would have had (Appendix I).

Per-layer allocation. Spending the same mean of 128 rows uniformly across layers loses 2.3 $[0.9, 3.8]$ against full KV; allocating it by each layer's measured value (Section 2.3) recovers 1.6 $[0.1, 3.0]$ of that, paired, and brings the arm inside full KV's interval.

Block size at a fixed ratio. The block size trades the raw window a sequence keeps exact, and so its memory, against the number of lossy compressions its history passes through: at 4:1, a late query at our mean length reads through 51 compression events at C=256 and 13 at C=1024. The suite is monotone in block size, 40.7, 41.5 and 42.5 for C=256, 512 and 1024 (Figure 5a), for 11% more KV per running sequence across the range (342 to 380 MiB) and no measurable change in

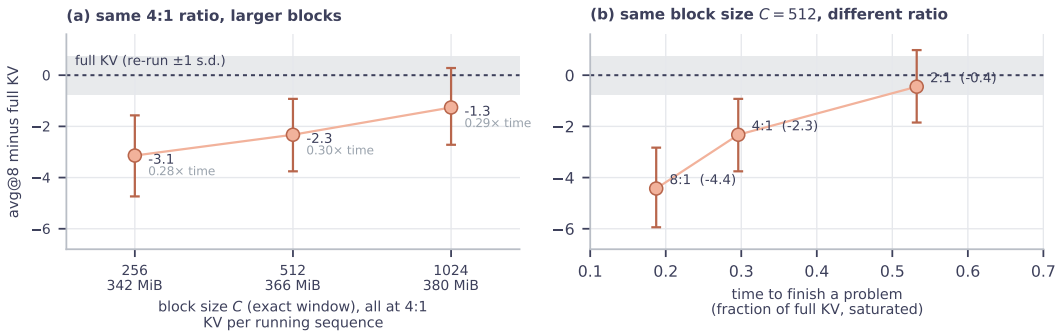


Figure 5: Compression shape; paired gap to full KV with 95% intervals (grey: 1 s.d. of a full-KV re-run). (a) Same 4:1 ratio, larger blocks: a little more KV per sequence (second tick row), fewer lossy compressions, higher accuracy, the same time to finish (grey). (b) Same block, higher ratio: faster, less accurate.

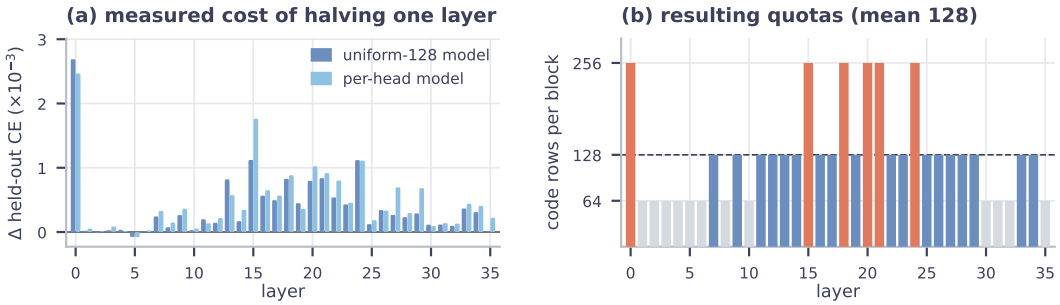


Figure 6: From measurement to quotas. (a) Held-out CE cost of halving one layer's code rows at evaluation time, on two trained models. (b) The trained schedule: the six most expensive layers get 256 rows, the twelve cheapest 64, at the same mean.

time to finish (0.28, 0.30 and 0.29× full KV's), and only the small-block end is separated from full KV (-3.1 $[-4.7, -1.6]$ at C=256, -1.3 $[-2.7, +0.3]$ at C=1024). Thought-level compression sits far to the small side of this range. LightThinker's algorithm, trained on our data and served in our engine, writes each thought (53 tokens on average in its generations, ended by a paragraph break) into 10 rows including its continue token, which is 5.3:1, and scores 33.3 on the suite, -10.5 $[-12.5, -8.7]$ against full KV (Table 1; Appendix G); our 8:1 arm, which keeps fewer rows, loses 4.4. It is also slower, 0.70× full KV's time to finish against our 0.31× (Figure 3d; an upper bound, Appendix G), since its compression step runs ten times as often. Training cost does not go up with block size over this range (Appendix C.1).

Compression ratio at a fixed block size. At C=512, 2:1 is lossless (-0.4 $[-1.8, +1.0]$) and finishes a problem in 0.53× full KV's time; 4:1 costs 2.3 points at 0.30× and 8:1 costs 4.4 at 0.19× (Figure 5b). The per-layer schedule moves the 4:1 point to parity at the same speed.

5 Analysis

5.1 Choosing the quotas

Question. Allocating rows by layer turns a significant loss into parity (Section 4.4). We want to know whether the gain comes from *where* the rows go, which would make the measurement behind the schedule essential, or merely from making the budget uneven.

Procedure. On the trained uniform model we halve one layer's code rows at evaluation time and record the increase in held-out cross-entropy, for each of the 36 layers (Figure 6a). The most expensive layers are raised, the cheapest cut, at the same mean, and the model is retrained on that schedule

(Figure 6b). For contrast we train a schedule of the same mean chosen without the trained model, from the heads whose full-KV attention is hardest to approximate with few keys; it raises the early layers.

Result. The measured cost sits in a few layers, the first layer and a band in the middle of the network, while most early and late layers can lose half their rows at almost no cost. Moving rows by this measurement reaches parity with full KV; the schedule chosen from the full-KV model's attention, which raises exactly the early layers the measurement finds cheap, is no better than uniform (Appendix B, Table 2). The placement is what matters, and it has to be read from the trained compressed model: a proxy from the uncompressed model points the wrong way. Two refinements do not pay. Single-layer costs are not additive (halving every layer costs twice their sum), so we cut only a third of the layers and use three levels rather than fitting small differences; and a per-head schedule matches per-layer accuracy while costing memory or speed to serve (Appendix B).

5.2 Information flow through the code rows

Question. Once a block is compressed its raw KV is gone, so anything later reasoning takes from that block must pass through its code rows. We want to see this directly: whether a specific fact written in a block is stored in that block's code rows and retrieved from them many blocks later.

Procedure. We insert one sentence defining a random four-digit value ("Denote this intermediate result as $Q_7 = 7391$.") into block b of the per-layer model's own AIME solutions, ask for it k blocks later ("Recall that $Q_7 =$"), and record whether it is reproduced and where its tokens attend.

Result. The value is recalled about three times in four (full KV: almost always), with no decline from one to eight compressions, and never when block b's code rows are removed. The recalling tokens attend to block b's code rows far more than to any other block's and far more than the same positions do when no value was planted (Figure 4b): the information travels by the most direct route, from the block into its own code rows and from there to the token that needs it. This routing happens in the 256-row layers, the ones the measurement of Section 5.1 raised, and in the 128-row layers. Failed recalls attend to the right rows as strongly as successful ones, so what limits recall is how much the code rows hold, not whether the model finds them. The model's own reasoning shows the same: numbers reused from a block depend on its code rows alone (Appendix B).

6 Discussion

Scope. Mathematical reasoning is the common test bed of the closest related work (Kontonis et al., 2026; Zhang et al., 2025; Cai et al., 2025), and it is ours. It is a demanding test of long generation: solutions run to tens of thousands of tokens, correctness is verifiable, and our suite pools twelve competitions over five years, 446 problems, so the claims above are resolved at a precision single benchmarks do not reach. Nothing in the recipe is specific to the domain or the model: the trainer, the layout and the allocation procedure take any decoder with rotary positions, the allocation is re-measured on each trained model rather than transferred, and the serving path is the stock paged engine with one cache group per quota. We plan to extend it to larger models and to other domains, such as code generation, and we expect it to carry over; larger models may also be more compressible, since their KV states carry more redundancy per token.

Block size. Accuracy is still rising at $C{=}1024$, so larger blocks may be better still; we did not train them, although they cost little more to train (Appendix C.1). The trade-off is memory: a larger block keeps more raw rows resident per sequence, so at a fixed cache it means fewer sequences in flight. We expect the practical choice to be the largest block the deployment's cache can carry.

Limitations. The method is trained rather than applied: every result starts from a base model with our own SFT, and adding it to a released, RL-tuned reasoning checkpoint without losing that tuning is untested. The compression ratio is fixed per block, so a block that carries little and one that carries a lot get the same rows. And the quota schedule comes from single-layer measurements that we assumed to be independent (Section 5.1); a method that measured how cuts interact could find a better one.

REPRODUCIBILITY STATEMENT

Every arm is trained from Qwen3-4B-Base on the same 2B response tokens in the same order (the canonical data stream), with the hyperparameters, hardware and throughput of Appendix C; the 446-problem suite, its sources and the sampling protocol are in Section 3 and Appendix D; the serving implementation, the benchmark protocol (pinned CPU cores, CUDA graphs on every arm, the shared concurrency-cap grid) and the per-arm measurements are in Appendix F; and the parity checks that tie the served model to the trained one are in Appendix J. We will release the training code, the vLLM patch, the evaluation and benchmark scripts, and the trained checkpoints.

AI USE STATEMENT

In this work, we used generative AI tools (a coding agent) for implementing methods, including the training, evaluation, serving and benchmark code; for running and monitoring experiments; for drafting and editing text; for creating figures and tables from the logged results; and for searching and summarizing the literature. We have not used generative AI tools for research ideas or for interpreting results, and we have not used them for generating synthetic datasets: our SFT data are derived from OpenMathReasoning, a public dataset of model-written solutions that we did not generate. Developing theoretical models, formulating mathematical claims, providing or writing proofs, translation, and qualitative data analysis are not applicable to this work. Every reported number was produced by code we ran and was checked against the underlying logs. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant J. Nair, Ilya Soloveychik, and Purushotham Kamath. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. In *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

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

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan-Kelley. Reducing transformer key-value cache size with cross-layer attention. *arXiv preprint arXiv:2405.12981*, 2024. NeurIPS 2024.

Aydar Bulatov, Yury Kuratov, and Mikhail Burtsev. Recurrent memory transformer. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

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

Zefan Cai, Wen Xiao, Hanshi Sun, Cheng Luo, Yikai Zhang, Ke Wan, Yucheng Li, Yeyang Zhou, Li-Wen Chang, Jiuxiang Gu, Zhen Dong, Anima Anandkumar, Abedelkadir Asi, and Junjie Hu. R-kv: Redundancy-aware kv cache compression for reasoning models. In *Advances in Neural Information Processing Systems*, 2025.

Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Chong-Yan Chen, Yu-Fang Hu, Pei-Shuo Wang, Ning-Chi Huang, Luis Ceze, Mohamed S. Abdelfattah, and Kai-Chiang Wu. Palu: Kv-cache compression with low-rank projection. In *International Conference on Learning Representations*, 2025.

Ting-Yun Chang, Harvey Yiyun Fu, Deqing Fu, Chenghao Yang, Jesse Thomason, and Robin Jia. Value-aware stochastic kv cache eviction for reasoning models. *arXiv preprint arXiv:2606.03928*, 2026.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

Jeffrey Cheng and Benjamin Van Durme. Compressed chain of thought: Efficient reasoning through dense representations. *arXiv preprint arXiv:2412.13171*, 2024.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. In *Proceedings of EMNLP*, pp. 3829–3846, 2023.

DeepSeek-AI. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

DeepSeek-AI. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025. doi: 10.1038/s41586-025-09422-z.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S. Kevin Zhou. Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference. In *Advances in Neural Information Processing Systems*, 2025.

Tianyu Fu, Haofeng Huang, Xuefei Ning, Genghan Zhang, Boju Chen, Tianqi Wu, Hongyi Wang, Zixiao Huang, Shiyao Li, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang. Mixture of attention spans: Optimizing llm inference efficiency with heterogeneous sliding-window lengths. In *Conference on Language Modeling (COLM)*, 2025a.

Yu Fu, Zefan Cai, Abedelkadir Asi, Wayne Xiong, Yue Dong, and Wen Xiao. Not all heads matter: A head-level kv cache compression method with integrated retrieval and reasoning. In *International Conference on Learning Representations (ICLR)*, 2025b.

Yizhao Gao, Shuming Guo, Shijie Cao, Yuqing Xia, Yu Cheng, Lei Wang, Lingxiao Ma, Yutao Sun, Tianzhu Ye, Li Dong, Hayden Kwok-Hay So, Yu Hua, Ting Cao, Fan Yang, and Mao Yang. Seerattention-r: Sparse attention adaptation for long reasoning. *arXiv preprint arXiv:2506.08889*, 2025.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive kv cache compression for llms. In *International Conference on Learning Representations (ICLR)*, 2024a.

Tao Ge, Jing Hu, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model. In *International Conference on Learning Representations (ICLR)*, 2024b.

Gemma Team. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*, 2024.

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. *arXiv preprint arXiv:2401.18079*, 2024. NeurIPS 2024.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of EMNLP*, pp. 13358–13376, 2023.

Hongye Jin, Xiaotian Han, Jingfeng Yang, Zhimeng Jiang, Zirui Liu, Chia-Yuan Chang, Huiyuan Chen, and Xia Hu. Llm maybe longlm: Self-extend llm context window without tuning. In *International Conference on Machine Learning (ICML)*, 2024.

Jang-Hyun Kim, Junyoung Yeom, Sangdoo Yun, and Hyun Oh Song. Compressed context memory for online language model interaction. In *International Conference on Learning Representations (ICLR)*, 2024.

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

Vasilis Kontonis, Yuchen Zeng, Shivam Garg, Lingjiao Chen, Hao Tang, Ziyan Wang, Ahmed Awadallah, Eric Horvitz, John Langford, and Dimitris Papailiopoulos. MEMENTO: Teaching LLMs to manage their own context. *arXiv preprint arXiv:2604.09852*, 2026.

Anna Kuzina, Maciej Pioro, Paul N. Whatmough, and Babak Ehteshami Bejnordi. Kava: Latent reasoning via compressed kv-cache distillation. In *International Conference on Learning Representations (ICLR)*, 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 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. In *Advances in Neural Information Processing Systems*, 2024.

Xin Liu, Runsong Zhao, Xinyu Liu, Junhao Ruan, Pengcheng Huang, Shichao Dong, Chunyang Xiao, Chenglong Wang, Changliang Li, Jingbo Zhu, and Tong Xiao. Hybridthinker: Efficient chain-of-thought reasoning via compressed memory and transient thought steps. *arXiv preprint arXiv:2606.03768*, 2026.

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

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. *arXiv preprint arXiv:2402.02750*, 2024. ICML 2024.

Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, Zhiqi Huang, Huan Yuan, Suting Xu, Xinran Xu, Guokun Lai, Yanru Chen, Huabin Zheng, Junjie Yan, Jianlin Su, Yuxin Wu, Neo Y. Zhang, Zhilin Yang, Xinyu Zhou, Mingxing Zhang, and Jiezhong Qiu. Moba: Mixture of block attention for long-context llms. In *Advances in Neural Information Processing Systems*, 2025.

Amirkeivan Mohtashami and Martin Jaggi. Landmark attention: Random-access infinite context length for transformers. In *Advances in Neural Information Processing Systems*, 2023.

Jesse Mu, Xiang Lisa Li, and Noah Goodman. Learning to compress prompts with gist tokens. In *Advances in Neural Information Processing Systems*, 2023.

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 EMNLP*, 2025.

Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143*, 2024.

Piotr Nawrot, Adrian Łańcucki, Marcin Chochowski, David Tarjan, and Edoardo M. Ponti. Dynamic memory compression: Retrofitting llms for accelerated inference. *arXiv preprint arXiv:2403.09636*, 2024. ICML 2024.

OpenAI. Learning to reason with LLMs. `https://openai.com/index/learning-to-reason-with-llms/`, 2024.

Matanel Oren, Michael Hassid, Nir Yarden, Yossi Adi, and Roy Schwartz. Transformers are multi-state rnns. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 18724–18741, 2024.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of ACL*, pp. 963–981, 2024.

Jianhui Pang, Fanghua Ye, Derek F. Wong, Xin He, Wanshun Chen, and Longyue Wang. Anchor-based large language models. *arXiv preprint arXiv:2402.07616*, 2024. Findings of ACL 2024.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*, 2023.

Guanghui Qin, Corby Rosset, Ethan C. Chau, Nikhil Rao, and Benjamin Van Durme. Dodo: Dynamic contextual compression for decoder-only lms. In *Proceedings of ACL*, 2024.

Ziran Qin, Yuchen Cao, Mingbao Lin, Wen Hu, Shixuan Fan, Ke Cheng, Weiyao Lin, and Jianguo Li. Cake: Cascading and adaptive kv cache eviction with layer preferences. In *International Conference on Learning Representations (ICLR)*, 2025.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations (ICLR)*, 2020.

Akshat Ramachandran, Marina Neseem, Charbel Sakr, Rangharajan Venkatesan, Brucek Khailany, and Tushar Krishna. Thinkv: Thought-adaptive kv cache compression for efficient reasoning models. In *International Conference on Learning Representations (ICLR)*, 2026.

Isaac Rehg. Kv-compress: Paged kv-cache compression with variable compression rates per attention head. *arXiv preprint arXiv:2410.00161*, 2024.

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

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Jiwon Song, Dongwon Jo, Yulhwa Kim, and Jae-Joon Kim. Reasoning path compression: Compressing generation trajectories for efficient llm reasoning. *arXiv preprint arXiv:2505.13866*, 2025. NeurIPS 2025.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

Yang Sui, Yu-Neng Chuang, Guanchu Wang, Jiamu Zhang, Tianyi Zhang, Jiayi Yuan, Hongyi Liu, Andrew Wen, Shaochen Zhong, Na Zou, Hanjie Chen, and Xia Hu. Stop overthinking: A survey on efficient reasoning for large language models. *Transactions on Machine Learning Research*, 2025.

Wenhui Tan, Jiaze Li, Jianzhong Ju, Zhenbo Luo, Ruihua Song, and Jian Luan. Think silently, think fast: Dynamic latent compression of llm reasoning chains. In *Advances in Neural Information Processing Systems*, 2025.

Hanlin Tang, Yang Lin, Jing Lin, Qingsen Han, Shikuan Hong, Yiwu Yao, and Gongyi Wang. Razorattention: Efficient kv cache compression through retrieval heads. In *International Conference on Learning Representations (ICLR)*, 2025.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. In *International Conference on Machine Learning (ICML)*, 2024.

Zhongwei Wan, Xinjian Wu, Yu Zhang, Yi Xin, Chaofan Tao, Zhihong Zhu, Xin Wang, Siqi Luo, Jing Xiong, Longyue Wang, and Mi Zhang. D2o: Dynamic discriminative operations for efficient long-context inference of large language models. In *International Conference on Learning Representations (ICLR)*, 2025.

Heng Wang, Jielin Qiu, Wenting Zhao, Cheng Qian, Liangwei Yang, Jiawei Han, Heng Ji, Silvio Savarese, Shelby Heinecke, and Huan Wang. Random attention: Rethinking kv cache eviction for efficient reasoning. *arXiv preprint arXiv:2609.03430*, 2026.

Zheng Wang, Boxiao Jin, Zhongzhi Yu, and Minjia Zhang. Model tells you where to merge: Adaptive kv cache merging for llms on long-context tasks. *arXiv preprint arXiv:2407.08454*, 2024a.

Zihao Wang, Bin Cui, and Shaoduo Gan. Squeezeattention: 2d management of kv-cache in llm inference via layer-wise optimal budget. *arXiv preprint arXiv:2404.04793*, 2024b.

Jialong Wu, Zhenglin Wang, Linhai Zhang, Yilong Lai, Yulan He, and Deyu Zhou. Scope: Optimizing key-value cache compression in long-context generation. In *Proceedings of ACL*, 2025a.

Wenhao Wu, Yizhong Wang, Guangxuan Xiao, Hao Peng, and Yao Fu. Retrieval head mechanistically explains long-context factuality. In *International Conference on Learning Representations (ICLR)*, 2025b.

Heming Xia, Chak Tou Leong, Wenjie Wang, Yongqi Li, and Wenjie Li. Tokenskip: Controllable chain-of-thought compression in llms. In *Proceedings of EMNLP*, 2025.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. Infllm: Training-free long-context extrapolation for llms with an efficient context memory. In *Advances in Neural Information Processing Systems*, 2024a.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations (ICLR)*, 2024b.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: Efficient long-context llm inference with retrieval and streaming heads. In *International Conference on Learning Representations (ICLR)*, 2025.

Silei Xu, Wenhao Xie, Lingxiao Zhao, and Pengcheng He. Chain of draft: Thinking faster by writing less. *arXiv preprint arXiv:2502.18600*, 2025.

Dongjie Yang, Xiaodong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. Pyramidinfer: Pyramid kv cache compression for high-throughput llm inference. In *Findings of ACL*, 2024.

Xubing Ye, Yukang Gan, Xiaoke Huang, Yixiao Ge, and Yansong Tang. Voco-llama: Towards vision compression with large language models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 29836–29846, 2025.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*, 2025. ACL 2025, Best Paper Award.

Haoyue Zhang, Hualei Zhang, Xiaosong Ma, Jie Zhang, and Song Guo. Lazyeviction: Lagged kv eviction with attention pattern observation for efficient long reasoning. In *Proceedings of ACL*, 2026.

Jintian Zhang, Yuqi Zhu, Mengshu Sun, Yujie Luo, Shuofei Qiao, Lun Du, Da Zheng, Huajun Chen, and Ningyu Zhang. Lightthinker: Thinking step-by-step compression. In *Proceedings of EMNLP*, 2025.

Peitian Zhang, Zheng Liu, Shitao Xiao, Ninglu Shao, Qiwei Ye, and Zhicheng Dou. Long context compression with activation beacon. *arXiv preprint arXiv:2401.03462*, 2024a. ICLR 2025.

Yuxin Zhang, Yuxuan Du, Gen Luo, Yunshan Zhong, Zhenyu Zhang, Shiwei Liu, and Rongrong Ji. Cam: Cache merging for memory-efficient llms inference. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *PMLR*, pp. 58840–58850, 2024b. URL https://proceedings.mlr.press/v235/zhang24n.html.

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. H_2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2023.

Dawei Zhu, Nan Yang, Liang Wang, Yifan Song, Wenhao Wu, Furu Wei, and Sujian Li. Pose: Efficient context window extension of llms via positional skip-wise training. *arXiv preprint arXiv:2309.10400*, 2023.

A Related work

We sort prior work by what it compresses and whether the compression is learned. Every method that learns a compressed form of the model's own output was validated on generations one to two orders of magnitude shorter than ours, and every method validated at our length selects among existing rows rather than learning new ones.

Learned compression of the model's own output. Few papers train a model to compress what it is generating. MEMENTO (Kontonis et al., 2026) trains 8B–32B models to segment their reasoning into blocks, write a textual summary (a memento) of each block and reason forward attending only to the mementos, with about $2.5\times$ lower peak KV; it finds that information from a block also survives in the memento's own KV states, computed while the block was still visible. Its summary is text the model writes; ours is a set of learned KV rows, so the model is taught to summarize without ever emitting text. Its models also lose accuracy against their full-context counterparts on competition mathematics before reinforcement learning, whereas our recipe matches its full-KV counterpart with supervised training alone (Section 4.1). LightThinker (Zhang et al., 2025) is the nearest learned-row method: at each thought boundary the model writes the thought into a fixed set of cache tokens (9 for Qwen) under a special mask and drops the thought's raw KV, evaluated with outputs of at most 10K tokens on GSM8K, MMLU, GPQA and BBH. HybridThinker (Liu et al., 2026) keeps the last few steps raw and trains with a mix of visible and memory-only attention, at about 2.5K-token outputs. Both fix the summary in absolute size for a variable-length thought, so the ratio is set by the model's own segmentation; we fix the fraction per block and vary it by layer. Dynamic Memory Compression (Nawrot et al., 2024) learns, per head and step, whether to append a new KV pair or merge it into the last slot, and is the clearest precedent for learned heterogeneous ratios during decoding; it merges raw rows rather than writing new ones and was evaluated with short outputs. AnLLM (Pang et al., 2024) compresses each sentence into an anchor token and CCM (Kim et al., 2024) compresses a conversation turn by turn. Latent-reasoning methods replace the chain itself with vectors (Hao et al., 2024; Cheng & Van Durme, 2024; Tan et al., 2025; Kuzina et al., 2026); they give up the readable trace and are evaluated at GSM8K scale. None of these reports accuracy against a full-KV model on generations of our length, which is the comparison our results are about.

Masked-attention compression tokens for inputs. Our code rows are an instance of a device introduced for inputs: place a few learned tokens after a span, hide the span from everything that follows, and train end to end so its information must route through those tokens. Gist tokens (Mu et al., 2023) do this for instruction prompts, ICAE (Ge et al., 2024b) as an in-context autoencoder, AutoCompressors (Chevalier et al., 2023) as per-segment soft prompts, Activation Beacon (Zhang et al., 2024a) on the per-layer KV of each chunk with a ratio sampled during training, and VoCo-LLaMA (Ye et al., 2025) across modalities. Activation Beacon is the closest architecture: it also keeps compressed KV and proceeds in segments. What separates our setting is that these methods compress a fixed input once, before generation, and are evaluated with short outputs. We compress the generated stream once per completed block, tens of times inside one response, and each later summary is written by a model whose context is already made of summaries. They place the summary after its span (Section 4.4); we place it inside.

Training-free eviction during decoding. Generic evictors keep recent tokens and heavy hitters by accumulated attention (Zhang et al., 2023; Liu et al., 2023; Oren et al., 2024; Xiao et al., 2024b; Ge et al., 2024a; Adnan et al., 2024), and merging variants fold evicted rows into kept ones (Zhang et al., 2024b; Wang et al., 2024a). A reasoning-specific line targets long chains of thought: R-KV scores tokens by importance and redundancy (Cai et al., 2025), Reasoning Path Compression compresses

periodically from a window of recent queries (Song et al., 2025), and LazyEviction and VaSE evict on Qwen3-4B with outputs of up to 16K and about 18K tokens on average (Zhang et al., 2026; Chang et al., 2026); SCOPE separates prefill from decoding compression (Wu et al., 2025a) and ThinKV assigns precision by thought type (Ramachandran et al., 2026). Random eviction within the reasoning phase has been found to match the scored methods on the same models and benchmarks (Wang et al., 2026), which suggests that the scores are not the binding lever. These methods are our baselines (Section 4.3, Appendix E). They choose among rows that exist; our model writes new rows, trained for the queries that will read them.

Compression of the prompt. Most KV compression shrinks a long input once, after prefill: SnapKV scores the prompt from an observation window (Li et al., 2024), with layer and head budgets in PyramidKV, Ada-KV and HeadKV (Cai et al., 2024; Feng et al., 2025; Fu et al., 2025b), and KVzip scores entries by how well they let the model reconstruct the context and evicts query-agnostically (Kim et al., 2025). Text-level compressors shorten the prompt string instead (Jiang et al., 2023; Pan et al., 2024). None of these touches the generated stream, which is where the cache of a long reasoning response lives.

Sparse reads over a full cache. Another line keeps every row and reduces what each step reads: Quest (Tang et al., 2024), InfLLM (Xiao et al., 2024a), MoBA (Lu et al., 2025), and, trained for reasoning, NSA, evaluated on AIME at 8K–16K generation limits (Yuan et al., 2025), and SeerAttention-R, which learns a block-level gate over a full cache at up to 32K tokens (Gao et al., 2025). NSA's compression branch maps each block of keys to one compressed key, the closest mechanism to a block code, but the raw cache stays resident for its selection branch, so KV memory is not reduced. These methods compose with ours: they cut reads, we cut what is stored.

Allocating the budget across layers and heads. Heterogeneous budgets have been set per head from attention statistics (Ge et al., 2024a; Feng et al., 2025), retrieval-head probes (Fu et al., 2025b; Tang et al., 2025) building on Wu et al. (2025b), learned gates (Xiao et al., 2025) or search over window lengths (Fu et al., 2025a), and per layer from fixed pyramids (Cai et al., 2024; Yang et al., 2024) or measured statistics (Wang et al., 2024b; Wan et al., 2025; Qin et al., 2025). KV-Compress serves per-head rates inside paged attention (Rehg, 2024). Reported gains over uniform are modest outside very small budgets, and no decode-time reasoning compressor ablates the allocation. We measure each layer's value on the trained model itself, by cutting its rows, and find a real per-layer gain ($+1.6$ avg@8); refining further to per-head quotas did not improve on it (Section 5.1).

Positions for summary rows. RoPE (Su et al., 2021) makes attention depend on relative offset, which lets a code row take any position. Position interpolation and its successors (Chen et al., 2023; Peng et al., 2023; Zhu et al., 2023) decouple position ids from token count, and Self-Extend lets many tokens share one coarse position (Jin et al., 2024). Compressors use the default: gist tokens are ordinary tokens in the sequence (Mu et al., 2023), accumulated Activation Beacons occupy consecutive positions in the compacted stream (Zhang et al., 2024a), and evictors leave kept rows at their original positions (Li et al., 2024; Zhang et al., 2023). Our in-block, bit-reversed grid is, to our knowledge, new, and the ablation of Section 4.4 shows that the choice is worth 6.2 points and is not an extrapolation effect.

Orthogonal reductions. Quantization (Liu et al., 2024; Hooper et al., 2024), low-rank and latent KV (DeepSeek-AI, 2024; Chang et al., 2025), and sharing across heads or layers (Shazeer, 2019; Ainslie et al., 2023; Brandon et al., 2024) shrink each row, the row width, or the number of layers that store KV; Qwen3-4B already uses grouped-query attention, and each of these stacks on our row reduction. Bounded-memory architectures (sliding-window and hybrid layers (Beltagy et al., 2020; Gemma Team, 2025), recurrent and compressive memory (Rae et al., 2020; Bulatov et al., 2022; Munkhdalai et al., 2024)) change the model rather than compress its cache. Finally, methods that shorten the chain of thought (Xia et al., 2025; Xu et al., 2025; Sui et al., 2025) reduce the number of generated tokens rather than what each keeps resident, and also compose with ours.

Table 2: Budget allocation at a fixed mean of 128 code rows per block (all other settings identical). Held-out CE on 192 held-out SFT pairs; MATH-500 avg@4; suite avg@8 at 32K as in Table 1. The per-head row lets a head keep up to all 512 rows, with a floor of 32, nested inside the per-layer schedule; paired against the per-layer headline it is -0.6 $[-2.1, +0.9]$. The last row sets per-head quotas from how concentrated each head's attention is in the full-KV model, which raises the early layers; paired against the per-layer model it is -1.9 $[-3.3, -0.5]$.

Schedule	held-out CE	MATH-500	Suite avg@8 [95% CI]
uniform (every head 128)	0.4227	88.3	41.5 [37.6, 45.3]
per layer, measured on the trained model (ABRIDGEKV)	0.4215	87.5	43.0 [39.1, 46.9]
per head, measured, $K_{\max}$=512 (nested)	0.4209	–	42.4 [38.5, 46.3]
per head, from full-KV attention statistics	0.4231	–	41.1 [37.2, 45.0]

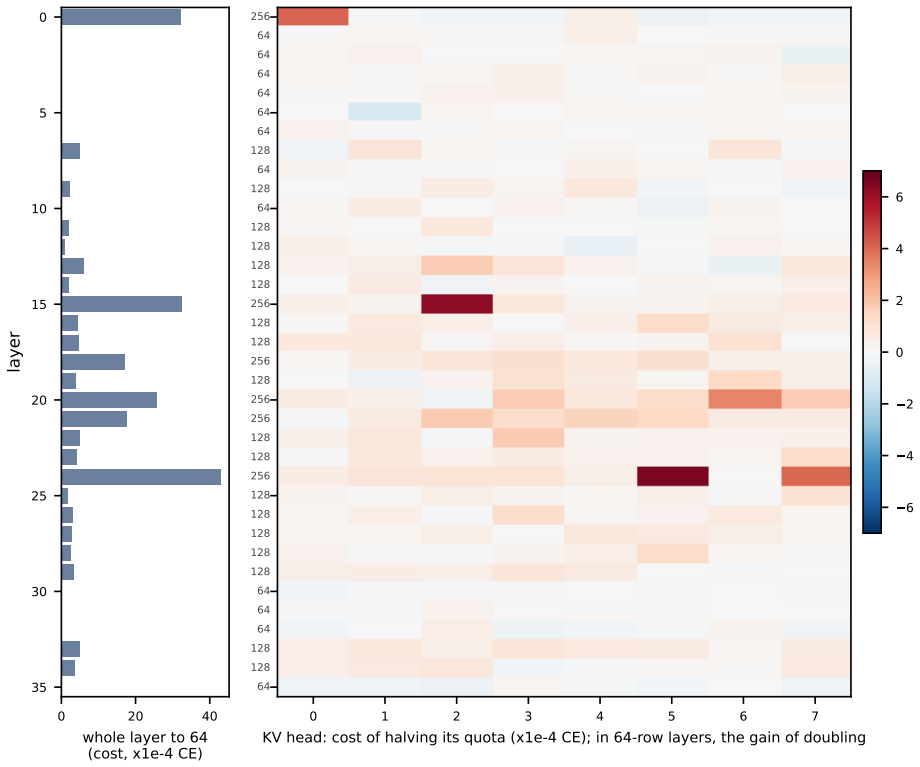


Figure 7: Where a layer's code rows are worth something, head by head. On the trained per-layer model, each (layer, KV head) cell shows the held-out CE cost of halving that one head's quota (in 64-row layers, the gain of doubling it), on the full 192-pair held-out set; the bars at left repeat the whole-layer cost of Figure 6. Noise floor: 0.4×10^{-4} (s.d. of head cells in layers whose whole-layer cost is nil).

B CHOOSING THE QUOTAS AND INFORMATION FLOW: DETAILS

Per head versus per layer. Figure 7 repeats the truncation measurement of Section 2.3 head by head on the per-layer model. In the layers cut to 64 rows, doubling any single head gains nothing; in the layers at 128, no head is expensive; in the six layers raised to 256, the value sits in one or two heads. So the per-layer schedule already captures nearly all of the value that can be moved at this mean, and a per-head schedule trained from scratch (Table 2) is within noise of it. It would also cost to serve: a paged engine keeps one row count per layer, so ragged per-head quotas either pad every head to the layer's largest ($1.53\times$ the memory of the per-layer schedule) or need a packed side cache, which adds about 29 ms of fixed overhead per decode step. We therefore allocate per layer.

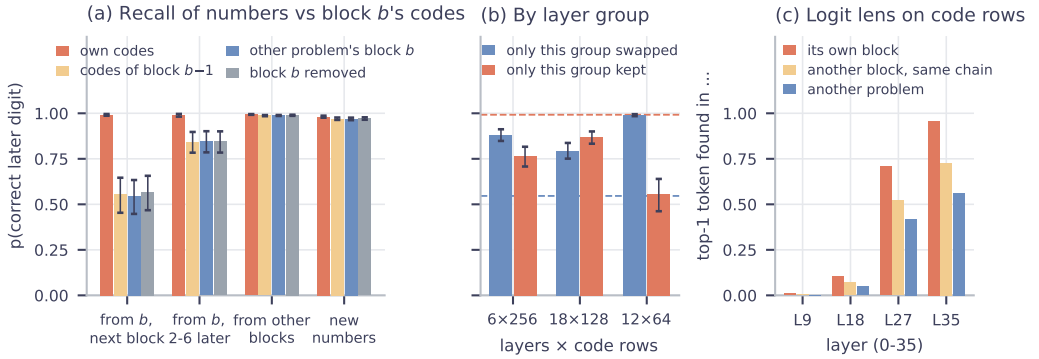


Figure 8: What the code rows carry (115 AIME chains, 895 intervened blocks). (a) Probability of a number's later digits, by where it was first written, with block b's code rows intact, replaced, or removed (95% intervals over chains). (b) Numbers from b reused in the next block, with one layer group swapped or only one kept; dashed: all intact, all swapped. (c) How often a code row's logit-lens top token occurs in its own block, another block, or another problem.

Code-row probe: further results. Figure 8 adds three views to Section 5.2. *Layers.* Swapping block b's code rows in only the 12 layers cut to 64 rows leaves next-block recall at 0.99, and keeping only those layers intact is as bad as swapping every layer (0.55); the 256- and 128-row layers share the content (kept alone, 0.77 and 0.87). *Logit lens.* Projected onto the vocabulary after layer 27, whose output later layers store, a code row's most likely token occurs in its own block for 71% of blocks, against 52% for another block of the same chain and 42% for another problem's (96, 73 and 56% after the last layer). *Attention.* During the code step, digits that are reused later get no more attention than other digits (0.69 against 0.85 of uniform); newlines get the most ($2.2\times$). Numbers reused two to six blocks later are recalled at 0.85 even without block b's rows, partly because context lets the model re-derive them.

C Training Details

Table 3: Training and evaluation setup shared by every arm.

Base model	Qwen3-4B-Base
SFT data	OpenMathReasoning-derived math SFT, 400K prompt/response pairs, canonical 8x1x32 order
Response tokens per arm	2B (about 936 steps at effective batch 256)
Block size C / code slots $K_{\max}$ / mean quota	512 / 256 / 128
Optimizer, schedule	AdamW, lr 1e-4 WSD
Generation cap at eval	32768 tokens
Hardware	one node per run, either $4\times$ NVIDIA GB300 or $8\times$ NVIDIA H100-80GB; FlexAttention, bf16 autocast. The canonical data order fixes the per-step example set independently of world size, so an arm is the same run on either machine (Appendix J)

The learning rate warms up linearly over the first 3% of the 2B response tokens, holds at 10^{-4} until 80% of the budget, and decays linearly to 10^{-5} over the last 20% (a warmup–stable–decay schedule); AdamW with $\beta = (0.9, 0.95)$, weight decay 0.1 and gradient clipping at 1.0; effective batch 256 sequences (about 2.1M response tokens per step), 936 steps. The data order is fixed per step independently of the number of GPUs (`DATA_STREAM=canonical:8x1x32`), and the held-out set is the first 192 pairs.

Additions that did not help. Two additions were trained under the same protocol, each on the uniform-128 setting, and then combined with the per-layer schedule. *Raw tail.* The last 16 to-

kens of every completed block stay uncompressed for good and remain visible to every later query, so the most recent text before each code step is exact; the code rows summarise the other 496 positions on the same in-block grid, and the block keeps 144 rows instead of 128. *Distillation.* The frozen full-KV model trained on the same data order serves as teacher, and the loss adds $\lambda T^2\,\mathrm{KL}(\text{teacher}\,\|\,\text{student})$ at $\lambda{=}1$, $T{=}1$ on the response positions, so the compressed model is pulled toward the full-KV model's next-token distribution rather than only toward the data. Neither improved on the uniform setting it was added to (41.6 and 39.9 against 41.5), and the three-way combination with the per-layer schedule scored 41.1, significantly below the per-layer schedule alone (-1.9 $[-3.4, -0.4]$ paired).

Seed-to-seed noise. Retraining the full-KV model on a different data seed shifts the suite score by 0.8 points (1 s.d. of the paired difference), the grey band of the figures; a gap inside it cannot be told from seed noise.

C.1 Training cost of the block size

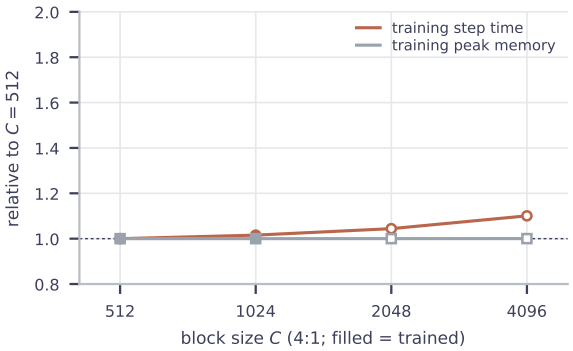


Figure 9: Training cost of the block size at 4:1, relative to $C{=}512$: step time and peak memory for one forward and backward pass over a 16K-token sequence on one H100 (filled: block sizes we trained). $C{=}256$, not shown, takes $1.9\times$ as long, from partly filled attention tiles and twice as many blocks.

We time one forward and backward pass of the trainer (FlexAttention, bf16 autocast, activation checkpointing) over a 16K-token response at 4:1 for C from 512 to 4096 (Figure 9). Peak memory is the same at every block size, because with checkpointing it is set by the sequence length; step time rises by 2% at $C{=}1024$ and 10% at $C{=}4096$, from the attention inside each open block.

D Competition suite

D.1 Pooling across competitions

Most long-generation compression papers report on one or two 30-problem sets, usually AIME. On our own paired data a single 30-problem set puts the 95% interval on a gap between two arms at roughly ±5 points, even with eight samples per problem and the per-problem pairing those published tables mostly lack: a vanilla replicate differing only in its data seed comes out significantly worse than vanilla on AIME 2024 and better on AIME 2025, and the uniform arm's gap changes sign between AIME 2024 and HMMT Feb 2025. Pooling 446 problems brings the half-width to 1.5 points, which is what makes the uniform-versus-dynamic difference resolvable at all. Cheaper metrics do not substitute for it either: across the $4\times$ range of budget at $C{=}512$, held-out CE moves by 0.010 while the suite moves by 4.0 points (Table 7), and MATH-500 by 1.6.

E R-KV comparison

R-KV (Cai et al., 2025) scores each generated token by attention importance minus key redundancy and keeps the highest-scoring rows up to a fixed budget. We port it to the same vLLM version

Table 4: The competition suite of Section 3: 446 problems from twelve competitions. "post-cutoff" sets were released after the base model's data cutoff.

Set	source	problems	era
brumo_2025	MathArena/brumo_2025	30	post-cutoff
cmimc_2025	MathArena/cmimc_2025	40	post-cutoff
smt_2025	MathArena/smt_2025	53	post-cutoff
aime_2026	MathArena/aime_2026	30	post-cutoff
hmmt_feb_2023	MathArena/hmmt_feb_2023	30	pre-2025
aime_2023	MathArena/aime_2023_I+II	30	pre-2025
aime_2022	AI-MO/aimo-validation-aime (url 2022)	30	pre-2025
amc_2022_2023	AI-MO/aimo-validation-amc	83	pre-2025
aime_2024	MathArena/aime_2024_I+II	30	pre-2025
aime_2025	MathArena/aime_2025	30	post-cutoff
hmmt_feb_2024	MathArena/hmmt_feb_2024	30	pre-2025
hmmt_feb_2025	MathArena/hmmt_feb_2025	30	post-cutoff

Table 5: R-KV against the arm it is paired with at each budget, on the same 446 problems, the same sampling and the same grader. KV/512 = persistent rows per 512 generated tokens per KV head; R-KV's is its *measured* mean residency, so at both settings it holds about 12% more cache than the arm beside it. trunc. = % of samples that reach the 32K cap. Δ is the paired per-problem difference against that partner arm, bootstrapped on the 446 common problems. Accuracy separates them only at the tighter budget; truncation and generated length separate them at both.

Arm	KV/512	Suite avg@8	Δ vs. partner	trunc. %	gen. tokens
vanilla SFT (full KV)	512	43.8 [39.8, 47.6]	–	3.4	13,162
ABRIDGEKV, per-layer 4:1	128	43.0 [39.1, 46.9]	–	3.2	13,257
R-KV, budget 4096	144	42.6 [38.5, 46.6]	-0.4 [-2.1, +1.2]	31.7	18,428
ABRIDGEKV, uniform 8:1	64	39.4 [35.5, 43.2]	–	3.6	13,673
R-KV, budget 2048	74	35.6 [31.7, 39.5]	-3.8 [-5.5, -2.2]	55.8	22,230

we serve with and run it on the full-KV checkpoint with the same problems, sampling and grader. Table 5 collects the measurements.

Budget matching. R-KV's budget is a constant number of rows; ours is a rate. We match them by measured residency: over the suite's samples a budget of 4096 leaves about 12% more generated rows resident than our 4:1 arm, and 2048 about 12% more than our 8:1 arm, so each R-KV setting is given slightly more cache than the arm it is compared with.

Served form. R-KV's reference code selects tokens per KV head, which a paged cache cannot honour, since dropping a token frees it for every head. R-KV's own SGLang and vLLM ports reduce the score to one decision per token (mean over heads, sum over layers), and we follow that reduction. We also enable its opt-in flag that returns evicted pages to the allocator; without it the cache occupancy equals full KV's and eviction saves no memory. Eviction rewrites the cache in place, so prefix caching and asynchronous scheduling are off for R-KV, as in its release.

Port and verification. We verified the scoring against the released implementation before running anything. A CPU test drives our `vllm/rkv/algo.py` and `R-KV/HuggingFace/rkv/compression/r1_kv.py` on the same inputs and compares the attention term, the redundancy term and the selected keys and values: 45 of 45 configurations agree bit for bit, over fp32, bf16 and fp16, three GQA shapes, cache lengths 64 to 1024 and budgets 32 and 128. The remaining hyperparameters are those of R-KV's own evaluation command: importance weight λ=0.1, an observation window of 8 tokens and a pooling kernel of 7 (the retain ratio is inert with the default retain direction). The cadence is the one place we chose a setting: R-KV compresses every 128 decoding steps and we ran every 64, which is more expensive per token and keeps the resident cache closer to the budget; we chose the more frequent setting. The redundancy term builds a key-by-key matrix per layer, and the port silently falls back to full KV for

a request whose matrix exceeds a size cap; we size the cap from the budget, trace every compaction, and accept a run only if no request was skipped. Both reported runs compacted every request.

F SERVING DETAILS

Implementation. The patch adds compressed decoding to vLLM's v1 engine. When a request completes a block of C tokens, the scheduler appends its K pause tokens; their KV at every layer is written into the block's persistent slots, and the block's C raw rows are freed. The pause tokens run in the same forward pass as other requests' decode tokens (merged code steps). This removes a separate engine step and the rebuild of the persistent batch that followed it, which took 12–40% of wall time before merging; merged steps are captured as piecewise CUDA graphs up to 2048 tokens, the same capture range for every arm, baselines included, and merging is batch-invariant (identical greedy tokens on 64 of 64 prompts for every arm). A per-layer schedule is served as one KV cache group per quota level, so a layer with quota q stores q rows per block and nothing else; uniform arms use a single group. Each engine is pinned to its own GPU and CPU cores.

Time to finish. Each arm's time to finish is taken at the best cap of a doubling grid, with both neighbouring caps measured, with two exceptions. ABRIDGEKV at 256 concurrent sequences runs out of GPU memory: at 128 its cache is already 92% full, and the activations of 256 sequences do not fit beside it, so 128 is the largest cap it can run on one H100. Its time to finish is therefore bounded by memory rather than by a measured optimum, and a GPU with more memory could run more of its sequences at once; the uniform setting, which does run at 256, is slower there than at 128 (2.73 against 2.60 s), so we expect that further gain to be small. R-KV at budget 4096 is flat between 128 and 256 (27.6 s at both), three times full KV's time, so a larger cap cannot change its position.

G LIGHTTHINKER IN OUR STACK

LightThinker (Zhang et al., 2025) is the closest prior method: it compresses a model's own reasoning online into learned KV entries and discards the compressed span. We compare against its mechanism trained on our data and served in our engine rather than against its released checkpoints, which are 7B/8B models trained on different data with 4K-token sequences and evaluated greedily at 10K tokens, so their numbers would confound base model, data and decoding with the compression method.

The arm. A thought ends at a paragraph break; the model then writes 9 cache tokens at appended positions under LightThinker's attention mask and continues from them, and the thought's raw KV is dropped at inference. Data, recipe and the 2B-token budget are those of every other arm; the served model matches the training-time forward pass (argmax agreement 0.996). Its generations end a thought every 53 tokens on average (counted in the engine over 7.7M generated tokens), and each thought leaves 9 cache rows and the continue token, so the arm keeps 97 rows per 512 generated tokens, between our 8:1 and 4:1 arms.

Result. It scores 33.3 on the suite, -9.8 $[-11.5, -8.0]$ paired against ABRIDGEKV, and 2.0 points below our appended-position ablation, which shares its positions but compresses 512-token blocks. The compression unit explains most of the gap: at a fixed ratio, the block size sets how many lossy compressions a late query reads through and how much recent text stays exact (Table 6), and thought-level compression sits far below the smallest block we train. It also writes longer, 14.9K tokens on average against 13.2K for full KV, and truncates 5.4% of samples against 3.4% (Figure 4a). Training is slower as well: the augmented sequences carry 25% more tokens, the thought-boundary mask is irregular, and the run fit only micro-batches of 2; per GPU it processed original tokens at 71% of the rate of our appended-position arm on the same H100 nodes (run logs, not a controlled measurement).

Serving measurement. Its time to finish (Figure 3d) is an upper bound. Its compaction is triggered by a sampled token, which our asynchronous scheduler cannot anticipate, so it ran with synchronous scheduling at about 65% GPU utilisation while every other arm ran asynchronously; against the synchronous full-KV cell it is $0.66\times$. It also needs real prompts and sampling (T=0.6) to end

thoughts at all (greedy decoding of the dummy prompt never emitted the boundary token), whereas the other arms replay their lengths greedily; the sampler's cost is not separated. Its best concurrency cap is 128, with both neighbours measured (64: 7.23, 128: 6.18, 256: 6.35 s per problem).

Table 6: Compression events a late query reads through, for our mean generation (13.2K tokens) and at the 32K cap, and the exact window (the open block, $C/2$ tokens in expectation).

block C	events (mean gen.)	events (at cap)	exact window
1024	13	31	512
512 (ours)	26	62	256
256	51	124	128
≈53 (LightThinker, measured)	249	618	26

H Block size and code budget sweep

Within one block size, halving the code budget costs far more on the suite than held-out CE suggests. At C=512, going from K=128 (4:1) to K=64 (8:1) moves held-out CE by 0.005 but the pooled suite by 2.1 points, 41.5 to 39.4: paired against vanilla that is -4.4 $[-5.9, -2.8]$ at 8:1 against -2.3 $[-3.8, -0.9]$ at 4:1, and MATH-500 moves only 88.3 to 86.7. Doubling the budget the other way, K=256 (2:1) moves held-out CE by a further 0.004 and the suite by 1.9 points to 43.4, which is -0.4 $[-1.8, +1.0]$ against vanilla: lossless within the interval, at half the compression. Across the whole $4\times$ range of budget held-out CE moves 0.010 while the suite moves 4.0 points. The remaining rows vary the block size at a fixed ratio (Figure 5); held-out CE is not comparable across block sizes, since the held-out set is defined by blocks per example.

I Code positions in prior learned summaries

No learned summary we know of places its rows inside the span they replace: accumulated Activation Beacons occupy consecutive positions in the compacted stream (Zhang et al., 2024a), AutoCompressors give summary vectors no position (OPT) or ordinary sequential RoPE (Llama-2) (Chevalier et al., 2023), gist tokens are ordinary tokens (Mu et al., 2023), and LightThinker (Zhang et al., 2025), the nearest published relative, writes its gist rows at the thought boundary without discussing their positions. The closest are methods that reuse an *existing* token's position (sentence anchors (Pang et al., 2024), selected token states (Qin et al., 2024)), defer position to retrieval time (Mohtashami & Jaggi, 2023), or apply an intra-block encoding inside the compressor without giving the compressed key an in-block position (Yuan et al., 2025). The appended arm is therefore our own model with the prior convention swapped in, not a reproduction of any one method: it shares our trainer, our fixed 512-token blocks, our budget and our data order, and it differs from LightThinker in segmentation (fixed blocks against thought boundaries), in emitting no trigger token, and in the shape of the budget. On shape: we fix the *fraction* (128 rows per 512-token block, 4:1, then varied per layer), whereas LightThinker fixes the *absolute* size (9 cache tokens per thought, 7 for Llama) of a variable-length unit, so its ratio is whatever its segmentation produces — about 1.7:1 for a short step and 20:1 for a long one, 5:1 to 7:1 on average by task. Our ablation therefore sits in a comparable average budget regime, which is what makes the layout the operative difference, but it is not that method's operating point. The closest published configuration is LightThinker's own token-level variant, which compresses every 6 generated tokens into 2 cache tokens at appended positions and which its authors report 5–6 points below their thought-level scheme on GSM8K-class tasks; our ablation is that variant's design at $512 \rightarrow 128$. No published learned-summary method has been evaluated at this generation length — LightThinker caps its outputs at 10K tokens, against our 32K cap and 13K average (Appendix A) — which is why a matched-training ablation, not a reimplementation, is the comparison we can defend.

J Parity checks

Every accuracy in the paper is measured on the served model, so the serving path must compute the same function as the trainer. We check this at three levels (Table 8). The trainer's single forward

Table 7: Block size and code budget sweep (uniform quotas, bitrev positions, $K_{\max}$=256, same 2B tokens). Held-out CE is comparable only within one block size (the held-out set is defined by blocks per example). MATH-500 avg@4; suite avg@8 at 32K as in Table 1.

C	mean K	ratio	held-out CE	MATH-500	Suite avg@8 [95% CI]
512	128	4:1	0.4227	88.3	41.5 [37.6, 45.3]
512	64	8:1	0.4281	86.7	39.4 [35.5, 43.2]
512	256	2:1	0.4184	87.6	43.4 [39.4, 47.2]
256	64	4:1	0.4246	86.9	40.7 [36.7, 44.5]
1024	256	4:1	0.4260	87.2	42.5 [38.6, 46.3]

pass under the block mask and the incremental generator, which compresses block by block as at inference, agree on per-token cross-entropy to within 4.5×10^{-4} on average; the same comparison with a deliberately wrong schedule differs by two orders of magnitude, so the check is sensitive to the kind of error it guards against. The vLLM engine reproduces the reference decoder's greedy tokens across a block boundary, and on the per-layer model with physical cache groups it agrees on the top-1 token at 99.5% of positions across two boundaries. The R-KV port reproduces the released scoring bit for bit (Appendix E). Together these rule out the two failure modes that would invalidate the comparison: a served model that is not the trained one, and a baseline that silently runs at full KV.

Table 8: Train/serve parity checks.

Check	Setting	Result
Trainer vs. incremental generator (per-token CE)	vph checkpoint, 3 held-out sequences, trained schedule	mean $\lvert\Delta\rvert$ 3.5e-4 to 4.5e-4, max 1.3e-2 to 2.3e-2
Same, deliberately wrong (uniform) schedule	control	mean $\lvert\Delta\rvert$ 1.6e-2 to 2.2e-2, max 4.0
HF reference vs. vLLM packed KV (greedy tokens)	uniform128, hetero_proxy, vph; 510 prompt + 16 generated (one block boundary)	identical greedy tokens
Multi-boundary HF vs. vLLM parity	per-layer headline, physical cache groups; 510-token prompt, 1,200 teacher-forced tokens (two block boundaries)	top-1 agreement 0.995, mean $\lvert\Delta \log p\rvert$ 0.006