ROUTECACHE: EFFICIENT LONG-CONTEXT ATTENTION VIA LEARNABLE RETRIEVAL AND TEMPORAL CACHING

Anonymous authors
Paper under double-blind review

ABSTRACT

Sparse attention can reduce long-context inference costs, but learning discrete token selection from the language modeling objective remains challenging. We introduce RouteCache, which combines end-to-end trained hash-based retrieval with an LRU working cache for efficient decoding. Our key insight is that attention-score gradients provide task-driven supervision for retrieval scores, enabling training beyond layer-wise imitation of dense attention. Starting from oracle initialization, we introduce closed-loop gradient feedback: gated hash scores influence attention during training and receive feedback from the resulting language modeling loss. This jointly optimizes hash networks across layers while keeping the pretrained backbone frozen. At inference, the gates are removed and the hash networks select tokens for sparse attention across all layers. To translate sparse retrieval into practical acceleration, the working cache reuses recently retrieved KV pairs, while fused kernels and asynchronous execution reduce data movement and scheduling overhead. Across Qwen3 models from 4B to 30B parameters, including MoE and reasoning variants, RouteCache achieves quality generally close to dense attention on perplexity, LongBench V2, and RULER evaluations. At 256K context, latency benchmarks show up to 4.9–7$\times$ speedups over StaticCache, depending on the retrieval and cache budgets. Code is available at `https://anonymous.4open.science/r/routecache-0921`.

1 INTRODUCTION

Long-context understanding is increasingly critical for LLM-based agent systems (Yao et al., 2023; Wong et al., 2026), which must maintain extensive interaction histories and process large codebases or document collections within a single session. Yet the quadratic complexity of attention fundamentally limits this capability (Yang et al., 2025b; DeepSeek-AI et al., 2025). Recent advances in Mixture-of-Experts reduce MLP costs, making attention efficiency increasingly important for long-context inference (DeepSeek-AI et al., 2025; Yuan et al., 2025; MiniMax et al., 2025).

Sparse attention offers a way to reduce this cost by selecting a small subset of contextual tokens. The central question is how to learn these selections so that they support the model's final predictions, rather than merely reproduce dense attention patterns at individual layers. We address this with RouteCache, combining end-to-end trained hash-based retrieval with a working memory cache for efficient execution.

Our sparse retrieval is realized by trainable hash functions that project queries and keys into compact binary spaces, replacing costly inner products with Hamming distance comparisons for efficient Top-K selection (Figure 1). To evaluate the effectiveness of Top-K routing, we freeze all pre-trained LLM parameters and optimize only the introduced hash networks and gating parameters.

Existing learned sparse attention methods often supervise each layer's routing with dense attention targets (Li et al., 2025; DeepSeek-AI et al., 2025). Such supervision rewards local agreement with an oracle, rather than directly optimizing how retrieval decisions across layers jointly affect the final prediction. Moving beyond this layer-wise objective is difficult because discrete Top-K selection blocks direct backpropagation through token selection. Our key insight is that attention-

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.

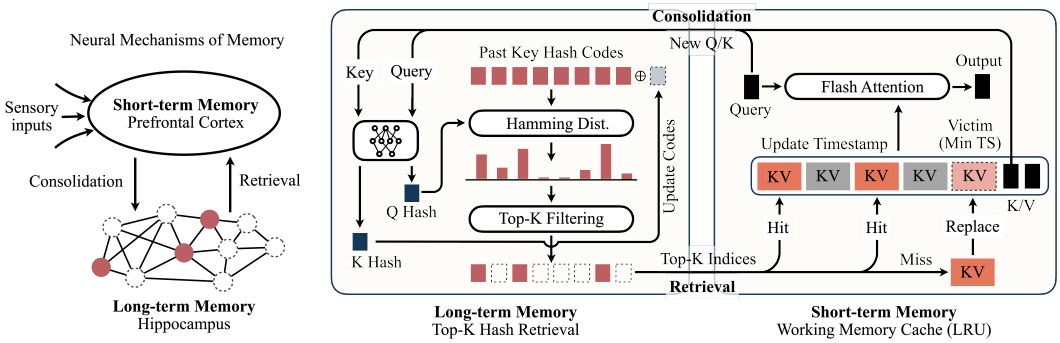


Figure 1: **The RouteCache Architecture.** (*Left*) Analogy to neural memory: the prefrontal cortex maintains short-term working memory while the hippocampus mediates long-term storage and retrieval. (*Right*) RouteCache mirrors this division by combining hash-based global retrieval for sparse historical access with an LRU-managed cache to exploit temporal locality.

score gradients already indicate how token scores should change to reduce the language modeling loss, providing a task-driven training signal for the hash networks (Figure 2, ①).

We introduce closed-loop training through a gated additive score formulation: hash scores influence predictions and receive feedback from the resulting language modeling loss, while retaining the original attention-score gradient path (Figure 2, ③). Building on oracle-based initialization, this supports joint optimization of retrieval across layers under the model's end task. Our experiments empirically evaluate this training formulation against an open-loop gradient-transfer baseline.

Previous sparse attention methods often retain selected dense layers to help preserve accuracy (Li et al., 2025; Chen et al., 2024). With the proposed training formulation, RouteCache applies sparse retrieval to all attention layers and achieves quality generally close to dense attention in our evaluations. This formulation may also inform future training methods for other discrete or heterogeneous attention architectures.

Sparse retrieval alone does not guarantee practical speedups: gathering scattered KV pairs can remain costly even when attention computation is substantially reduced. To support efficient execution of the learned retrieval, we use an LRU working memory cache to reuse KV pairs accessed across nearby decoding steps. Cache hit rates in Figure 3 (*middle & right*) provide empirical support for this reuse. Fused kernels and asynchronous scheduling further limit retrieval and cache-management overhead, translating attention sparsity into practical decoding acceleration.

We evaluate scalability and generalization across Qwen3 (4B/8B/14B) (Yang et al., 2025a), Qwen3-30B-MoE (Yang et al., 2025a), and Qwen3-4B-Thinking-2507 (Yang et al., 2025a). On Perplexity benchmarks, LongBench V2 (Bai et al., 2024), and RULER (Hsieh et al., 2024), RouteCache closely approaches dense attention accuracy even when decoding 16K tokens under a 256K context, with efficiency evaluated across a batch-size/context-length matrix, yielding up to 4.9-7× speedup over StaticCache in per-layer decoding latency.

In summary, our contributions are as follows:

- We introduce closed-loop gradient feedback for end-to-end hash retrieval training, jointly optimizing routing across layers under the language modeling objective rather than only matching layer-wise oracle targets.
- We develop RouteCache, combining learned token-level retrieval with an LRU working cache. By reusing cached KV pairs and fetching only cache misses from HBM, this design reduces repeated data transfers while preserving fine-grained token selection.
- We design parallel, timestamp-based cache-management kernels with lightweight bookkeeping in shared memory. Kernel fusion and asynchronous scheduling further reduce launch overhead and overlap retrieval and cache updates with K/V projections.

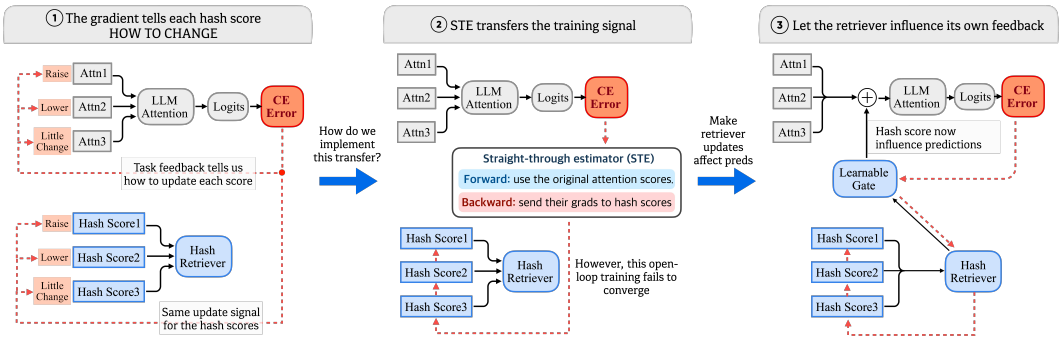


Figure 2: **RouteCache training.** (1) Attention-score gradients indicate how each score should change to reduce the language modeling loss. (2) A straight-through estimator transfers these gradients to hash scores while keeping the forward attention unchanged. (3) Adding gated hash scores to attention logits lets retrieval updates affect predictions and subsequent gradients, closing the training loop.

2 Related Work

2.1 Sparse Attention and KV Cache Compression

KV eviction methods such as H_2O (Zhang et al., 2023) and SnapKV (Li et al., 2024) reduce storage by retaining selected tokens, but cannot revisit discarded KV pairs. Dynamic sparse attention instead retrieves from the full historical cache, using techniques such as hash-based search (Chen et al., 2024; Li et al., 2025). RouteCache follows this retrieval paradigm: its working cache reduces repeated KV fetches without removing tokens from historical storage. Unlike storage-reduction methods, this design retains the full-history KV storage requirement and targets decoding computation and data movement.

Our main distinction lies in how retrieval is trained. Methods such as DeepSeek-V3.2 and SeerAttention (DeepSeek-AI et al., 2025; Gao et al., 2024) supervise routing with dense attention targets, encouraging agreement within each layer. RouteCache uses oracle supervision for initialization, then jointly optimizes hash networks across layers using the language modeling loss. Its closed-loop formulation lets hash-score updates affect predictions and subsequent gradients, moving beyond layer-wise imitation toward task-driven retrieval while keeping the backbone frozen. This allows retrieval preferences to adapt to their downstream effects, rather than treating each layer's oracle-selected tokens as fixed training targets.

We also favor token-level selection over block retrieval (Tang et al., 2024; Xiao et al., 2024), which improves contiguous access but can fetch many low-attention neighbors. In our analysis, covering the same top-2% token set with blocks of 64 requires approximately $27\times$ the attention FLOPs (Appendix D). Evaluations on Qasper, QMSum, and RepoBench also favor token-level selection (Appendix A). Pooling keys into block representatives can further obscure token-level attention patterns, making isolated relevant tokens harder to distinguish (Appendix D). RouteCache combines this fine-grained selection with temporal caching to reduce repeated random-access costs.

2.2 Temporal Locality in Attention

Temporal locality refers to the repeated access to contextual tokens across nearby decoding steps. H_2O (Zhang et al., 2023) identifies attention *heavy hitters*, a small subset of tokens that accounts for a large share of attention, and uses accumulated attention scores to retain these tokens alongside recent ones. StreamingLLM (Xiao et al., 2023) identifies *attention sinks*: initial tokens that attract attention even when they have limited semantic relevance. These findings suggest persistent targets for attention across decoding steps. Recent-token windows also overlap substantially between adjacent steps, while coherent generation may repeatedly refer to the same entities or evidence. Together, these patterns help explain the temporal locality observed in our retrieval sets (Figure 3), although sparsity alone does not guarantee such overlap (see Appendix C for further discussion).

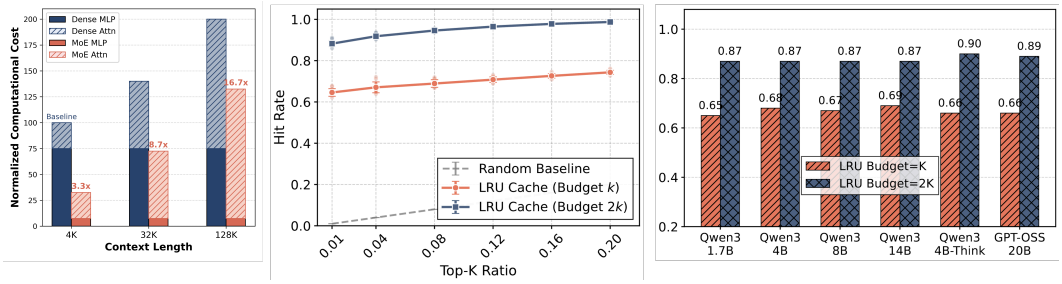


Figure 3: **Attention cost and cache reuse analysis.** (*Left*) Attention and MLP costs in the evaluated models. (*Middle*) LRU cache hit rate across Top-K ratios, with cache capacities of K and $2K$. (*Right*) Hit rates across models, reaching 87–90% with the $2K$ cache. Hit rate measures reuse from the cache (Appendix C).

3 Differentiable Sparse Retrieval

3.1 Hash-Based Retrieval with Working Cache

Following recent advancements in hash-based retrieval (Chen et al., 2024; Li et al., 2025), RouteCache compresses queries and keys into a binary space to accelerate Top-K selection via Hamming distance. At decoding step t, we project the query q_t and key k_t into binary vectors $\hat{q}_t, \hat{k}_t \in \{-1, 1\}^{d_{\text{hash}}}$ using a two-layer MLP followed by a binary activation function. We denote the original attention-head dimension by d_{head} and the hash-code dimension by d_{hash}:

$$\hat{q}_t = \text{sign}(\text{MLP}_q(q_t)), \quad \hat{k}_t = \text{sign}(\text{MLP}_k(k_t)). \tag{1}$$

This replaces costly inner products with highly efficient bitwise operations. The index set $\mathcal{T}_t$ of the Top-K historical keys is obtained by minimizing the Hamming distance D_H:

$$\mathcal{T}_t = \underset{i<t}{\text{Top-K}} \left(-D_H(\hat{q}_t, \hat{k}_i)\right). \tag{2}$$

Once $\mathcal{T}_t$ is identified, we compare it against the current LRU working memory $\mathcal{C}_{t-1}$. Cache misses are fetched from global memory to replace evicted tokens. The updated cache $\mathcal{C}_t$ contains the current historical retrievals together with retained tokens from earlier steps. The current token's KV pair is supplied directly, bypassing retrieval, so attention operates over $\mathcal{C}_t \cup \{t\}$:

$$o_t = \text{softmax}\left(\frac{q_t K_{\mathcal{C}_t \cup \{t\}}^\top}{\sqrt{d_{\text{head}}}}\right) V_{\mathcal{C}_t \cup \{t\}}. \tag{3}$$

This pipeline (Figure 1) effectively bridges global sparse retrieval with local, dense caching.

3.2 End-to-End Hash Routing via Gradient Feedback

Our goal is to optimize retrieval across layers for the final language modeling objective, rather than only reproduce each layer's dense attention targets. Discrete Top-K selection makes direct backpropagation through token selection difficult. We address this with attention-score gradient feedback, using oracle supervision as initialization. End-to-end training uses full causal attention over all tokens allowed by the causal mask ($j \leq i$), without Top-K truncation or LRU-based selection. It updates the hash networks and gating parameters while keeping the pretrained LLM frozen; sparse retrieval is applied only at inference.

Oracle Initialization. Following Li et al. (2025), we initialize each layer's hash networks by ranking dense-attention Top-K keys above non-Top-K keys. This provides a retrieval starting point, after which we optimize the networks jointly using the language modeling loss. We refer to initialization and end-to-end optimization as Stage 1 and Stage 2, respectively, in the experiments. The ranking objective and binary-code training details are provided in Appendix H.

Attention Gradients as Routing Signals. Attention-score gradients describe how changing token scores affects the language modeling loss and can supervise hash retrieval (Figure 2, ①). Let

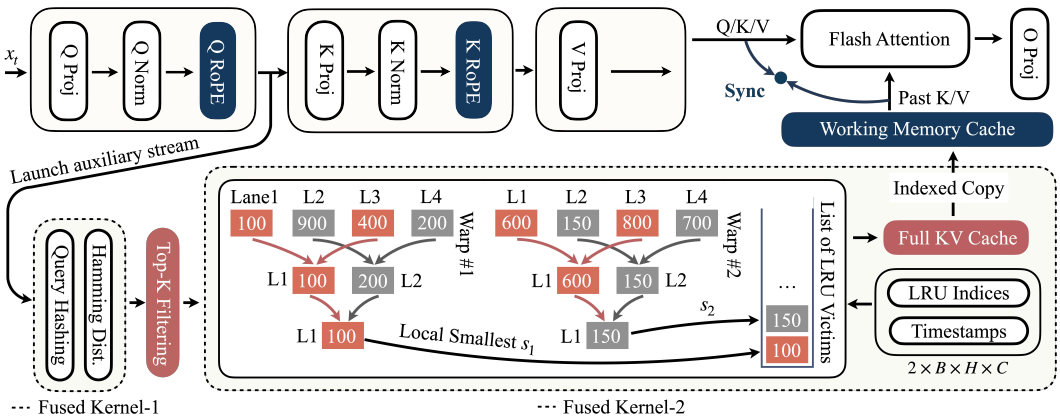


Figure 4: **RouteCache inference.** Attention combines the current token's KV pair with an LRU working cache containing retrieved and retained historical tokens. The schematic shows the main stream and the auxiliary retrieval path, which overlaps hash retrieval and working-cache updates with K/V projections and synchronizes before attention. A separate auxiliary stream updates the historical KV cache and key hashes after attention. This update path is not shown.

$s = q_i^\top k_j$ and $\hat{s} = \hat{q}_i^\top \hat{k}_j$. We express forward score coupling and backward gradient routing as

$$s' = as + (1-a)\,\mathrm{sg}(s) + b\hat{s} + (c-b)\,\mathrm{sg}(\hat{s}), \tag{4}$$

where sg denotes stop-gradient. For coefficients independent of the layer input x, the forward value and local backward rule are

$$\mathrm{forward}(s') = s + c\hat{s}, \qquad \frac{\partial s'}{\partial x} = a\frac{\partial s}{\partial x} + b\frac{\partial \hat{s}}{\partial x}. \tag{5}$$

Thus c sets the forward hash contribution, while a and b set the two gradient coefficients.

Closing the Feedback Loop. Open-loop gradient transfer uses $(a, b, c) = (0, 1, 0)$, whereas our method uses $(1, g, g)$ with a head-wise learnable gain g:

$$s'_{\mathrm{open}} = \mathrm{sg}(s - \hat{s}) + \hat{s}, \tag{6}$$

$$s'_{\mathrm{closed}} = s + g\hat{s}. \tag{7}$$

Open-loop training leaves forward scores unchanged and transfers score gradients only through the hash branch. Our formulation retains the original score's gradient path and lets hash updates affect predictions and subsequent gradients, closing the feedback loop (Figure 2, ②–③). It jointly optimizes hash networks across layers under the language modeling objective.

At inference, we remove the additive term: trained hashes select historical tokens, and attention over the LRU working cache and current token uses the original scaled QK scores.

Efficient Implementation. To avoid materializing the $O(N^2)$ attention and hash-score matrices, we express Eq. 7 as a single dot product via feature concatenation:

$$s'_{i,j} = [q_i, g \cdot \hat{q}_i]^\top \left[k_j, \hat{k}_j\right]. \tag{8}$$

This form allows FlashAttention (Dao, 2023) to compute the combined scores on the fly without explicitly storing either score matrix.

4 Efficient Sparse Attention via Temporal Caching

Building on the temporal locality discussed above, our LRU working cache reduces the KV-fetching costs of sparse retrieval (Yuan et al., 2025; Tang et al., 2024).

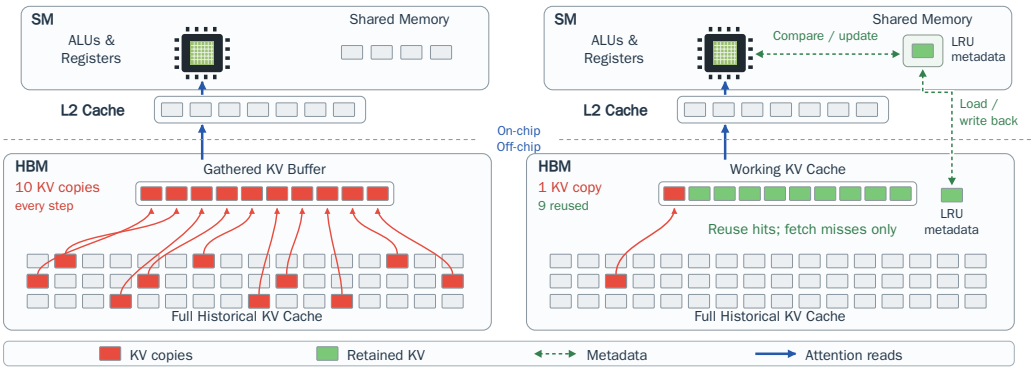


Figure 5: **Trading lightweight cache management for fewer KV copies.** (*Left*) Naive gathering copies all retrieved KV pairs. (*Right*) In the illustrated 10-token retrieval, RouteCache reuses nine cached pairs and fetches only one miss. LRU metadata is processed in shared memory, while both historical and working KV caches remain in HBM.

4.1 Lightweight LRU Cache Management

The Gathering Bottleneck. Token-level retrieval gathers scattered KV pairs from a historical cache that can far exceed L2 capacity. Limited L2 reuse and uncoalesced accesses make these gathers costly HBM transfers, so reducing the attention token count need not yield proportional latency savings. Our baseline with synchronous KV gathering is even slower than dense attention (Appendix F).

Lightweight LRU Management. We maintain a fixed-capacity working cache $\mathcal{C}$ in contiguous HBM storage. Rather than gathering every retrieved KV pair again, we compare the current retrieval set $\mathcal{T}_t$ against $\mathcal{C}_{t-1}$ and fetch only misses $\mathcal{M}_t$. Missing entries replace evicted victims $\mathcal{V}_t$ within the capacity limit L:

$$\mathcal{C}_t = (\mathcal{C}_{t-1} \cup \mathcal{M}_t) \setminus \mathcal{V}_t. \tag{9}$$

The manager operates on compact token indices and timestamps rather than full KV vectors. We stage this metadata in shared memory and perform hit tracking and victim selection on chip, using warp shuffles and shared-memory reductions instead of linked-list pointer chasing. Each hit updates its slot's timestamp $\tau_i \leftarrow t$, and parallel reductions select the $|\mathcal{M}_t|$ oldest eligible entries for replacement (Figure 4, *bottom*). Only compact metadata needs to be loaded and written back for bookkeeping; KV copies are restricted to misses. This trades repeated bulk gathering for lightweight on-chip cache management while keeping persistent KV buffers in HBM.

Reducing Historical KV Gathering. Figure 5 illustrates how lightweight on-chip bookkeeping avoids repeated KV copies. With the 87–90% working-cache hit rates in Figure 3, only 10–13% of the retrieved KV pairs require fetching from historical storage. This reduces historical KV gathering volume by approximately 8–10×, approaching an order of magnitude. The reduction applies to repeated gathering, not all HBM traffic: attention still reads the working cache. The corresponding latency improvements are quantified in Appendix F.

4.2 Kernel Fusion and Asynchronous Execution

Although caching reduces repeated KV transfers, intermediate tensor writes and kernel launches still contribute to decoding latency. Executing retrieval and projection operations sequentially also leaves opportunities for overlap unexploited. To reduce these costs and improve hardware utilization, we combine two system-level optimizations:

Kernel Fusion. Query hashing (hash-network projection, binarization, and bit-packing) and Hamming-distance computation are fused into one kernel, followed by separate Top-K selection and LRU cache-update kernels (Figure 4).

Asynchronous Execution. We use two auxiliary streams in addition to the main stream. The retrieval stream performs hash search and working-cache updates alongside K/V projections; the main stream waits for retrieval before FlashAttention. After attention, the update stream writes new KV pairs and

Table 1: **LongBench V2 results.** Ctx: context length limit. Max New: maximum generated tokens. Think: whether thinking mode is enabled. Top-K: historical tokens selected per decoding step. LRU: working-cache capacity, including the current Top-K set and retained tokens from earlier steps.

							LongBench V2 Accuracy ↑					
Model	**Ctx.**	**Max New**	**Think**	**Method**	**Top-K**	**LRU**	Overall	Easy	Hard	Short	Medium	Long
Qwen3-4B	40K	1K	✗	Full Attn	–	–	28.6	32.3	26.4	36.1	23.7	25.9
				MagicPIG	–	–	14.5	12.5	15.8	32.2	7.0	0.0
				Spotlight	2K	–	27.0	31.2	24.4	30.0	**23.3**	**29.6**
				RouteCache	1K	2K	**28.2**	**33.3**	**25.1**	**34.4**	**23.3**	27.8
Qwen3-8B	40K	1K	✗	Full Attn	–	–	32.8	38.5	29.3	39.4	25.6	36.1
				MagicPIG	–	–	18.5	18.2	18.6	**40.0**	8.8	1.9
				Spotlight	2K	–	31.8	37.5	**28.3**	37.2	**26.0**	**34.3**
				RouteCache	1K	2K	**32.6**	**39.6**	**28.3**	39.4	**26.0**	**34.3**
Qwen3-14B	40K	4K	✗	Full Attn	–	–	33.6	37.0	31.5	41.7	30.7	25.9
				MagicPIG	–	–	19.3	21.4	18.0	38.3	8.8	8.3
				Spotlight	2K	–	30.0	34.4	27.3	40.6	24.2	23.1
				RouteCache	1K	2K	**33.4**	**38.0**	**30.5**	**41.7**	**28.8**	**28.7**
30B-A3B	40K	4K	✗	Full Attn	–	–	33.7	40.0	29.9	40.6	28.9	33.3
				MagicPIG	–	–	25.2	30.7	21.9	41.1	16.7	15.7
				Spotlight	2K	–	30.4	38.5	25.0	41.1	22.8	25.0
				RouteCache	1K	2K	**33.0**	**43.0**	**26.8**	**41.1**	**26.4**	**31.9**
4B-Thinking	256K	16K	✓	Full Attn	–	–	39.0	43.2	36.3	51.7	30.7	34.3
				MagicPIG	–	–	24.9	22.9	26.0	23.3	25.1	26.9
				Spotlight	8K	–	35.0	39.1	32.5	41.7	32.1	29.6
				RouteCache	4K	8K	**38.6**	**44.8**	**34.7**	**48.3**	**34.0**	**31.5**

Table 2: **RULER results 1.** Accuracy (%) on the three hardest RULER subsets across context lengths 4K–32K.

					NIAH Multikey-3 ↑				QA-2 ↑				VT ↑			
Model	**Max New**	**Method**	**Top-K**	**LRU**	4K	8K	16K	32K	4K	8K	16K	32K	4K	8K	16K	32K
Qwen3-4B	1K	Full Attn	–	–	100.0	100.0	100.0	97.0	72.4	69.8	66.4	59.2	91.2	91.8	97.1	95.6
		Spotlight	2K	–	99.8	99.6	98.4	96.2	71.8	68.6	65.4	58.8	88.6	87.4	93.8	95.1
		RouteCache	1K	2K	**100.0**	**100.0**	**100.0**	**98.0**	**73.2**	**70.4**	**67.6**	**60.2**	**89.4**	**88.0**	**94.5**	**96.3**
Qwen3-8B	1K	Full Attn	–	–	100.0	98.0	95.0	95.0	81.0	75.8	74.4	74.6	99.6	99.6	98.8	99.4
		Spotlight	2K	–	99.6	98.8	97.4	96.0	79.4	76.2	71.8	71.0	99.3	99.2	98.9	98.4
		RouteCache	1K	2K	**100.0**	**100.0**	**100.0**	**99.0**	**80.2**	**77.0**	**72.2**	**71.6**	**99.5**	**99.4**	**99.1**	**98.6**
Qwen3-14B	4K	Full Attn	–	–	100.0	95.2	96.4	99.2	82.2	85.4	83.0	82.0	99.6	99.7	99.3	99.1
		Spotlight	2K	–	88.6	82.4	78.2	75.0	72.8	70.4	66.6	62.8	96.1	97.3	96.8	95.4
		RouteCache	1K	2K	**100.0**	**97.8**	**91.4**	**97.0**	**83.0**	**85.4**	**82.6**	**82.0**	**98.5**	**99.8**	**99.7**	**99.2**
30B-A3B	1K	Full Attn	–	–	92.8	93.0	93.8	98.8	83.0	79.4	73.8	67.8	100.0	100.0	100.0	100.0
		Spotlight	2K	–	84.2	82.0	78.8	71.4	80.6	77.4	72.2	67.0	**100.0**	**100.0**	99.8	99.6
		RouteCache	1K	2K	**92.4**	**96.8**	**94.2**	**97.4**	**84.8**	**83.2**	**76.4**	**69.2**	**100.0**	**100.0**	**100.0**	**100.0**

key hashes to historical storage. Each auxiliary worker is captured as a function-level CUDA graph containing multiple kernel calls, reducing host launch overhead. This is distinct from whole-model graph capture. Asynchronous scheduling reduces the retrieval work exposed on the attention critical path, with the benefit depending on the available overlap (Figures 4 and 9).

5 Experiments

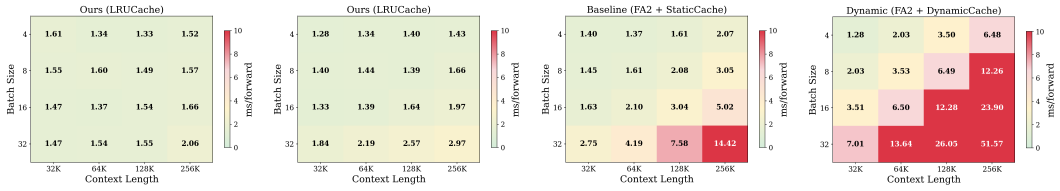
5.1 Experimental Setup

Models. We evaluate Qwen3 (Yang et al., 2025a) dense models (4B/8B/14B), an MoE model (30B-A3B), and a reasoning model (4B-Thinking). For all models, we set $d_{\text{hash}} = d_{\text{head}}$. The pretrained backbones remain frozen. Training updates only the hash networks and the gating parameters introduced for end-to-end optimization.

Training. For the main experiments, we train on a GitHub code corpus, concatenated into 256K-token sequences for 4B-Thinking and 40K-token sequences for the other models. Oracle initialization (Stage 1) uses top-192 attention targets for 4096 steps, taking 10–48 hours on 8 GPUs. Closed-loop training (Stage 2) adds 768–1024 steps and takes a further 1–3 days. Appendix E reports additional calibration-domain, cross-family, and storage results.

Table 3: **RULER results 2.** Accuracy (%) of 4B-Thinking on context lengths 16K–256K with thinking enabled and a maximum generation length of 16K tokens.

			NIAH Multikey-3 ↑					QA-2 ↑					VT ↑				
Method	**Top-K**	**LRU**	16K	32K	64K	128K	256K	16K	32K	64K	128K	256K	16K	32K	64K	128K	256K
Full Attn	–	–	100.0	100.0	98.0	82.0	5.0	94.0	93.0	84.0	70.0	46.0	100.0	100.0	100.0	98.4	85.4
Spotlight	8K	–	**100.0**	**100.0**	96.0	66.0	1.0	93.0	90.0	82.0	51.0	31.0	**100.0**	**100.0**	**100.0**	90.4	72.0
RouteCache	4K	8K	**100.0**	**100.0**	**98.0**	**78.0**	**4.0**	**96.0**	**91.0**	**85.0**	**63.0**	**42.0**	**100.0**	**100.0**	**100.0**	**100.0**	**87.2**



(a) Ours (1K / 2K) (b) Ours (4K / 8K) (c) StaticCache (d) DynamicCache

Figure 6: **Decoding latency (ms).** Each panel covers a 4×4 workload matrix: batch sizes 4, 8, 16, and 32, and context lengths 32K, 64K, 128K, and 256K. Panels (a) and (b) use Top-K/LRU budgets of 1K/2K and 4K/8K. Panels (c) and (d) show dense baselines.

Baselines. We compare against MagicPIG (Chen et al., 2024) and Spotlight (Li et al., 2025), two hash-based sparse attention methods. We use Spotlight's official training configuration and MagicPIG's default inference settings. Unlike these baselines, which retain selected dense layers, RouteCache applies sparse retrieval to all attention layers.

Temporal Retention Evaluation. We compare perplexity between eager attention over the current Top-K set and our kernel over the larger LRU working cache. Evaluations on book (Rae et al., 2019), math (Azerbayev et al., 2023), and code data also test transfer beyond the code-only training corpus.

Downstream Evaluation. We evaluate RouteCache with its accelerated kernel on LongBench V2 (Bai et al., 2024) and three RULER (Hsieh et al., 2024) subsets: NIAH Multikey-3, QA-2, and VT. For 4B-Thinking, we sample 100 queries per length for NIAH and QA-2 and use the full VT set. Maximum generation lengths are 16K tokens for 4B-Thinking with thinking enabled and 1K–4K for the other models. All decoding is greedy (temperature $= 0$) with fixed checkpoints and inputs; variability across training or calibration seeds is not evaluated.

Efficiency Evaluation. We measure per-layer latency of a single forward pass for Qwen3-4B-Thinking on a single H200 GPU, comparing Top-K/LRU budgets of 1K/2K and 4K/8K against the optimized StaticCache and DynamicCache implementations in Transformers 5.4.0. StaticCache preallocates KV storage to avoid repeated cache growth, providing a dense-attention baseline. All methods process identical ground-truth tokens from the first PG-19 (Rae et al., 2019) sample. Whole-model CUDA graph capture is disabled. RouteCache captures its auxiliary retrieval and update workers to reduce launch overhead.

Table 4: **Effect of temporal retention on perplexity.** Full Attn: dense attention. Eager: attention over the current retrieval set. LRU kernel: attention over the working cache, including current retrievals and retained historical tokens.

					Avg. PPL ↓		
Model	**Ctx.**	**Method**	**Top-K**	**LRU**	Book	Math	Code
Qwen3-14B	40K	Full Attn	–	–	7.85	1.93	2.47
		Eager	1K	–	7.72	1.92	2.39
		LRU kernel	1K	2K	7.70	1.93	2.40
30B-A3B	40K	Full Attn	–	–	8.20	1.94	2.42
		Eager	1K	–	7.92	1.93	2.25
		LRU kernel	1K	2K	7.90	1.93	2.32
4B-Thinking	256K	Full Attn	–	–	11.12	1.72	2.97
		Eager	2K	–	10.44	1.71	2.91
		LRU kernel	2K	4K	10.38	1.70	2.88

5.2 Quality Validation

Temporal Retention. The LRU kernel achieves perplexity comparable to or lower than full attention across the evaluated domains (Table 4). Relative to eager Top-K attention, retaining historical tokens improves perplexity in most settings, with small regressions in some math and code evaluations.

LongBench V2. Overall accuracy stays within 0.7 percentage points of full attention across all five models (Table 1). With a 256K context limit and a 16K generation budget, 4B-Thinking scores 38.6 versus 39.0 for full attention. Its average output length is 7,056 tokens (Table 5), showing that near-dense quality extends to long reasoning outputs.

RULER. RouteCache tracks full attention across the three subsets, though gaps remain at longer contexts (Tables 2 and 3). For 4B-Thinking at 128K, QA-2 accuracy falls from 70% to 63%, while VT improves from 98.4% to 100%. At 256K, both dense and sparse NIAH accuracy are low (5% and 4%). RouteCache outperforms Spotlight in most settings despite using no dense layers. We omit MagicPIG from RULER because of its high latency at 256K and poor LongBench V2 results.

5.3 Efficiency Evaluation

Results. Latency grows more slowly than the dense baselines as workload increases (Figure 6). At batch size 32 and 256K context, the 1K/2K configuration reaches 2.06 ms per layer, yielding $7\times$ and $25\times$ speedups over StaticCache and DynamicCache. The 4K/8K configuration used in downstream evaluation reaches 2.97 ms, or $4.9\times$ over StaticCache. Gains are smaller at low loads, where retrieval overhead can outweigh savings.

Table 5: **Generation length.** Mean output tokens with RouteCache on LongBench V2 (503 samples).

Model	Avg. Tokens
Qwen3-14B	2946
Qwen3-30B-A3B	1273
Qwen3-4B-Thinking	7056

5.4 Ablation Studies

Training Stability. Figure 7 compares two instances of Eq. 4 on Qwen3-4B: open-loop $(0, 1, 0)$ and closed-loop $(1, g, g)$. Both curves report *Top-K inference* LM loss relative to the frozen dense model. Open-loop residuals rise after 100 steps, while our formulation maintains stable inference loss and reaches negative residuals. This supports the complete training configuration empirically; forward coupling and backward gradient routing change together.

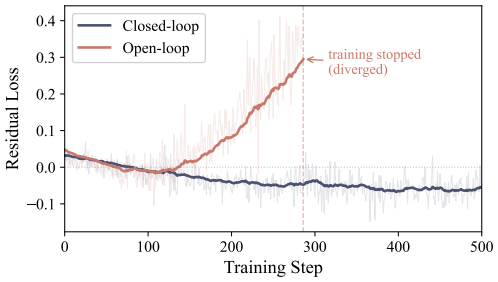


Figure 7: **Training stability.** Open-loop gradient transfer vs. closed-loop gated addition. Both curves show *Top-K inference* LM loss minus the dense model's loss.

Table 6: **End-to-end training gains.** RULER NIAH Multikey-3 accuracy (%) after oracle initialization (Stage 1) and subsequent closed-loop training (+Stage 2).

Model	Training	4K	8K	16K	32K
Qwen3-4B	Full Attn	100.0	100.0	100.0	97.0
	Stage 1 only	100.0	100.0	97.0	92.0
	+Stage 2	100.0	100.0	100.0	98.0
Qwen3-8B	Full Attn	100.0	98.0	95.0	95.0
	Stage 1 only	100.0	97.0	93.0	91.0
	+Stage 2	100.0	100.0	100.0	99.0

End-to-End Training Gains. Adding closed-loop training to oracle initialization improves or preserves NIAH Multikey-3 accuracy across the tested lengths (Table 6). At 32K, accuracy rises from 92% to 98% on Qwen3-4B and from 91% to 99% on Qwen3-8B, demonstrating gains beyond layer-wise oracle supervision.

System Optimizations. See Appendix F for ablations of LRU caching and asynchronous execution, and Appendix G for CUDA graphs and kernel fusion.

6 Conclusion and Limitations

RouteCache combines closed-loop hash training with temporal caching, achieving near-dense quality on Qwen3 and up to 4.9–$7\times$ per-layer decoding speedups over StaticCache at 256K, depending on retrieval/cache budgets. Calibration-domain and cross-family coverage remains limited, and cache efficiency depends on temporal locality. Serving-engine integration and sparse prefill remain future work, alongside broader evaluation across model families and diverse application domains.

AI USE STATEMENT

We used generative AI tools to polish language and formatting, check grammar, logical consistency, citations, and typographical errors, and assist with figure-generation scripts. The authors take responsibility for the final manuscript and artifacts.

ETHICS STATEMENT

We do not identify ethical concerns specific to this work on efficient LLM inference.

REPRODUCIBILITY STATEMENT

We provide code for reproducing the experiments, including complete training pipelines for all evaluated models, evaluation scripts for all reported benchmarks, and all acceleration kernels.

REFERENCES

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, et al. Llemma: An open language model for mathematics. *arXiv preprint arXiv:2310.10631*, 2023.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, et al. LongBench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. *arXiv preprint arXiv:2412.15204*, 2024.

Ralph Allan Bradley and Milton E. Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.

Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, et al. MagicPIG: LSH sampling for efficient LLM generation. *arXiv preprint arXiv:2410.16179*, 2024.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Pradeep Dasigi, Kyle Lo, Iz Beltagy, Arman Cohan, Noah A. Smith, et al. A dataset of information-seeking questions and answers anchored in research papers. *arXiv preprint arXiv:2105.03011*, 2021.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, et al. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Yizhao Gao, Zhichen Zeng, Dayou Du, Shijie Cao, Hayden Kwok-Hay So, et al. SeerAttention: Learning intrinsic sparse attention in your LLMs. *arXiv preprint arXiv:2410.13276*, 2024.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, et al. RULER: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.

Wenhao Li, Yuxin Zhang, Gen Luo, Haiyuan Wan, et al. Spotlight Attention: Towards efficient LLM generation via non-linear hashing-based KV cache retrieval. *arXiv preprint arXiv:2508.19740*, 2025.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, et al. SnapKV: LLM knows what you are looking for before generation. *arXiv preprint arXiv:2404.14469*, 2024.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023.

MiniMax, Aonian Li, Bangwei Gong, Bo Yang, Boji Shan, et al. MiniMax-01: Scaling foundation models with lightning attention. *arXiv preprint arXiv:2501.08313*, 2025.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. *arXiv preprint arXiv:1911.05507*, 2019.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, et al. RoFormer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

Mingjie Sun, Xinlei Chen, J. Zico Kolter, and Zhuang Liu. Massive activations in large language models. *arXiv preprint arXiv:2402.17762*, 2024.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, et al. Quest: Query-aware sparsity for efficient long-context LLM inference. *arXiv preprint arXiv:2406.10774*, 2024.

Sherman Wong, Zhenting Qi, Zhaodong Wang, Nathan Hu, Samuel Lin, et al. Confucius code agent: Scalable agent scaffolding for real-world codebases. *arXiv preprint arXiv:2512.10398*, 2026.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, et al. InfLLM: Training-free long-context extrapolation for LLMs with an efficient context memory. *arXiv preprint arXiv:2402.04617*, 2024.

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

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving Mamba2 with delta rule. In *ICLR*, 2025b.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, et al. ReAct: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *ACL*, 2025.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, et al. H2O: Heavy-hitter oracle for efficient generative inference of large language models. *arXiv preprint arXiv:2306.14048*, 2023.

Ming Zhong, Da Yin, Tao Yu, Ahmad Zaidi, Mutethia Mutuma, et al. QMSum: A new benchmark for query-based multi-domain meeting summarization. *arXiv preprint arXiv:2104.05938*, 2021.

A Block-level vs. Token-level Retrieval

Figure 8 compares token-level and block-level retrieval. On Qasper (Dasigi et al., 2021), QMSum (Zhong et al., 2021), and RepoBench (Liu et al., 2023), token-level retrieval generally achieves higher scores than block-level retrieval across the evaluated model sizes.

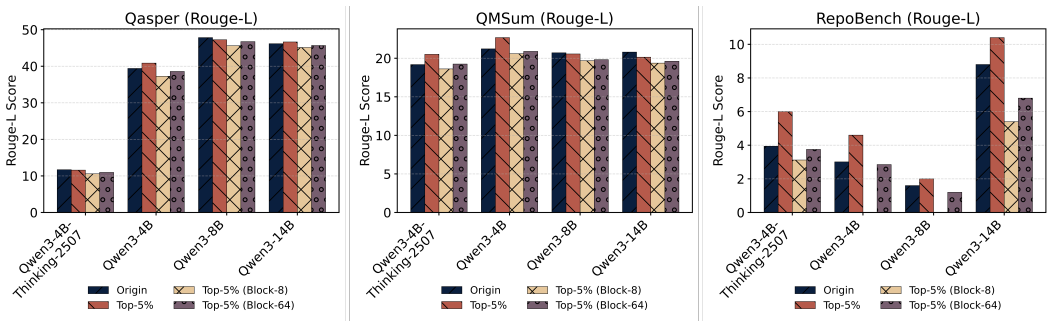


Figure 8: **Token-level vs. block-level retrieval.** Across the evaluated benchmarks and model sizes, token-level retrieval generally achieves higher scores than block-level retrieval.

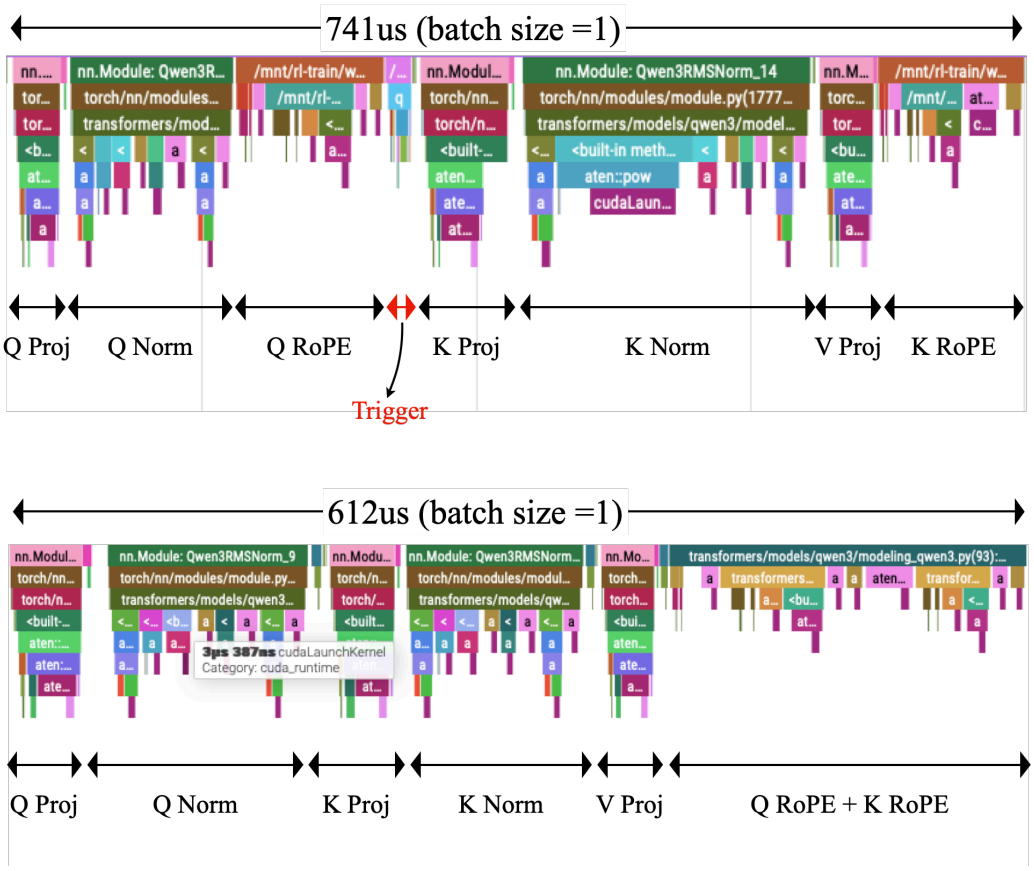


Figure 9: **Hiding query retrieval latency on H200.** (*Upper*) Auxiliary-stream CUDA graph replay overlaps retrieval and cache management with K/V projections. With full overlap, the exposed overhead is primarily the graph replay launch cost. (*Lower*) Dense attention without retrieval.

B Hiding Query Retrieval Latency with Asynchronous Execution

Figure 9 shows how asynchronous execution hides query retrieval and cache-management latency behind K/V projections during decoding. We launch these operations through CUDA graph replay on an auxiliary stream. When their execution is fully overlapped with K/V projections, the exposed overhead is primarily the graph replay launch cost, rather than the retrieval computation itself.

C Sources of Temporal Locality

Cache hit rate. Let $\mathcal{T}_t$ be the retrieved Top-K set at decoding step t, and let $\mathcal{C}_{t-1}$ be the cache contents before processing that step. The cache hit rate measures the fraction of retrieved tokens whose KV pairs are already cached:

$$H_t = \frac{|\mathcal{T}_t \cap \mathcal{C}_{t-1}|}{|\mathcal{T}_t|}.$$

Figure 3 reports this metric for different cache capacities and models. Only the remaining fraction $1 - H_t$ requires fetching historical KV pairs, reducing repeated transfers. This fraction describes retrieval misses, not total memory traffic or latency.

Temporal locality requires persistence in *which* tokens receive attention, not merely a sparse attention distribution at each step. Several observed attention patterns can contribute to this persistence.

Heavy hitters and attention sinks. H_2O (Zhang et al., 2023) finds that a small set of heavy-hitter tokens accounts for a large share of attention and that their emergence correlates with frequent token

co-occurrence. Its retention policy combines accumulated attention scores with recency. Attention sinks provide another source of repeated access: initial tokens can attract attention regardless of their semantic importance (Xiao et al., 2023). Massive activations have also been linked to attention concentrating on the corresponding tokens (Sun et al., 2024). These observations support the presence of recurrent attention targets, although high accumulated attention does not imply membership in the Top-K at every step.

Overlapping recent context. For a fixed window of W recent tokens, consecutive decoding steps share $W - 1$ tokens once the window is full. To the extent that attention favors recent context, this overlap creates opportunities for KV reuse. This explanation does not require recent tokens to dominate every head or layer.

These mechanisms offer an interpretation of our empirical locality measurements rather than a guarantee for arbitrary inputs. RouteCache accommodates changes in the retrieved set by fetching cache misses from the full historical KV cache, while reusing the KV pairs already present in working memory.

D Block-level Retrieval Overhead and Pooling Analysis

Covering the same top-2% token set with 64-token blocks requires approximately 27× the attention FLOPs of token-level selection in our analysis (Figure 10, first two panels). The extra computation comes from including low-attention neighbors in each retrieved block.

The remaining five panels compare the attention matrix A, variants without RoPE (Su et al., 2021) or semantic content, and mean- and max-pooled A. In this example, pooling weakens vertical attention patterns while diagonal structure remains prominent, suggesting that these operations obscure token-level distinctions.

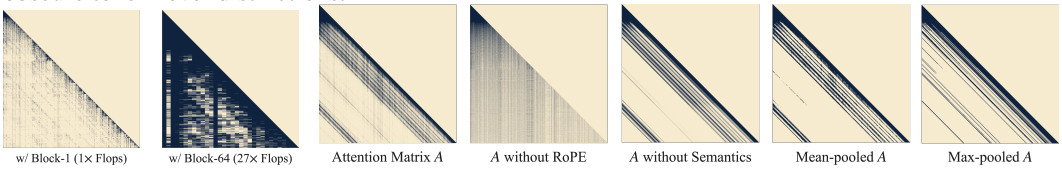


Figure 10: **Retrieval granularity and pooling effects.** From left to right: token-level retrieval (1× FLOPs), 64-token block retrieval (27× FLOPs), attention matrix A, A without RoPE, A without semantic content, mean-pooled A, and max-pooled A. Pooling weakens vertical patterns while retaining prominent diagonals in this example.

E Extra Experiments

Calibration domain and model family. Table 7 extends the main evaluation beyond GitHub calibration and Qwen3 backbones. On Qwen3-4B-Thinking, C4 calibration yields the same LongBench V2 score as code calibration (38.6). On Llama-3.1-8B, RouteCache scores 31.2 versus 30.4 for full attention. These single-checkpoint results support transfer to another calibration domain and model family.

Table 7: **Additional LongBench V2 results (%).** Both models use Top-K/LRU budgets of 4K/8K. Thinking is enabled only for Qwen3-4B-Thinking.

Model	Full Attn	Ours (GitHub)	Ours (C4)
Qwen3-4B-Thinking	39.0	38.6	38.6
Llama-3.1-8B	30.4	31.2	–

Parameter and storage overhead. The query/key hash MLPs contain $2(Q_H + K_H)d^2$ parameters per layer for the evaluated $d = 128$ configuration, where Q_H and K_H are the query and KV head

counts. Table 8 summarizes their overhead; training-only gates are excluded because they are removed at inference.

Table 8: **Hash-network parameter overhead.** Percentages are relative to total backbone parameters; storage assumes bf16.

Backbone	Hash params (M)	Backbone (%)	Storage (MiB)
Qwen3-4B / 4B-Thinking	47.19	1.17	90
Qwen3-8B	47.19	0.58	90
Qwen3-14B	62.91	0.43	120
Qwen3-30B-A3B	56.62	0.19	108

Per historical token and KV head, the 128-bit hash code occupies 16 bytes and the Top-K workspace adds 4 bytes, together contributing 3.91% relative to the 512-byte bf16 KV pair. The working cache and its metadata add a separate capacity-dependent cost. For Qwen3-8B at batch 1, 128K context, and a 2K working cache, tensor-size accounting gives approximately 1.08 GiB of extra storage over 18 GiB of dense KV (6.0%), including hash parameters, codes, workspace, and LRU buffers. These are analytical storage estimates, not measured peak GPU memory. RouteCache retains the full historical KV cache and optimizes access and computation rather than compressing its persistent capacity.

F Ablation of LRU Caching and Asynchronous Execution

We evaluate Qwen3-4B-Thinking-2507 on a single H200 at batch size 32 and 256K context, with Top-K/LRU budgets of 4K/8K. Table 9 reports decoding latency.

We compare direct KV gathering, which fetches all selected KV pairs from HBM at each step, with LRU caching, which fetches only cache misses. Each is tested with synchronous and asynchronous execution. All sparse variants use the same hash networks, with dense StaticCache as the reference.

Table 9: **Decoding latency with KV reuse and asynchronous execution.** Async indicates whether retrieval and cache operations overlap with K/V projections.

Retrieval Path	Async	Latency (ms) ↓
StaticCache (Full Attn)	–	14.42
Direct KV gathering	✗	16.59
Direct KV gathering	✓	7.81
LRU cache (Ours)	✗	4.52
LRU cache (Ours)	✓	**2.97**

Synchronous KV gathering is slower than dense attention (16.59 vs. 14.42 ms), showing that sparse selection alone need not improve latency. LRU reuse reduces latency to 4.52 ms, a $3.7\times$ speedup, by avoiding repeated historical KV transfers. Asynchronous execution further lowers it to 2.97 ms ($1.5\times$) by overlapping retrieval and cache management with K/V projections. Even with asynchronous execution, LRU caching remains $2.6\times$ faster than direct gathering. The two optimizations thus provide complementary benefits, with reuse savings depending on the cache hit rate (Appendix C).

G Ablation of Function-Level CUDA Graphs and Kernel Fusion

Under the setup in Appendix F, we separately disable function-level CUDA graphs and kernel fusion (Section 4.2). Whole-model graph capture remains disabled in all variants. Our function-level graphs capture only the auxiliary retrieval and update workers, each containing multiple kernel calls. All variants retain LRU caching and asynchronous execution.

Table 10: **Decoding latency with function-level CUDA graphs and kernel fusion.** Each ablation removes one optimization from the full configuration.

Configuration	Latency (ms) ↓
RouteCache (function-level graphs + kernel fusion)	**2.97**
w/o function-level graphs	3.63
w/o kernel fusion	18.89

H Oracle Initialization and Binary-Code Training

Following Li et al. (2025), Stage 1 initializes the hash networks independently at each layer using Top-K indices from dense oracle attention. Let $\mathcal{P}$ contain the oracle-selected keys and $\mathcal{N}$ the non-selected keys. We use a Bradley-Terry ranking objective (Bradley & Terry, 1952) to encourage smaller Hamming distances for selected keys:

$$\mathcal{L}_{\text{init}} = -\sum_{i\in\mathcal{P}}\sum_{j\in\mathcal{N}} \log\sigma\big(\beta\cdot(D_H(\hat{q},\hat{k}_j) - D_H(\hat{q},\hat{k}_i))\big), \tag{10}$$

where D_H is the Hamming distance, σ is the sigmoid function, and β controls the ranking-logit scale. We cache the frozen LLM's hidden states for independent layer-wise optimization.

Unlike the soft-sign training surrogate used by Li et al. (2025), our straight-through estimator emits hard-sign binary codes in the forward pass and uses tanh surrogate derivatives in the backward pass. This keeps the forward code representation binary during both training and inference. Oracle initialization supplies the starting point for the subsequent joint, language-modeling-based optimization; it is not the final training objective.