COARSE SEMANTIC BLOCKS FOR EFFICIENT TRAINING-FREE SPARSE ATTENTION

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce a training-free sparse attention method that builds a prompt-specific key index to accelerate both prefill and decoding. The method clusters the keys into large semantic blocks that group tokens from noncontiguous positions. Each query position selects blocks by scoring their centroids and attends to the original keys and values within them. Queries that select the same block are processed together, so prefill stays fast even though each query chooses its own blocks. Large blocks make centroid scoring and this grouped processing cheap, so each query can attend to more tokens in the same time. Larger blocks with a larger token budget can thus be both faster and more accurate on long-context tasks than smaller blocks with a smaller budget. We evaluate two hybrid LLMs, Qwen3.5-4B and 35B-A3B, on one NVIDIA H200. Including the cost of building and using the index, whole-model wall-clock speedups over dense attention are 2.3–2.4$\times$ for prefill and 1.7–2.0$\times$ for decoding at 512K tokens, and 2.9–3.0$\times$ for prefill at 1M. The 35B-A3B model stays close to dense attention on five RULER tasks through 512K, on LongBench-v2 overall, and on MRCR through 256K. The 4B model loses accuracy mainly on multi-key retrieval, yet outperforms dense Qwen3.5-2B, whose latency is similar at 512K, on all three benchmarks.

1 INTRODUCTION

Long contexts make attention expensive during both prefill and decoding. Sparse attention restricts each query to a subset of tokens, but it is faster only if choosing the tokens and preparing their keys and values costs less than the attention computation it saves. Methods with learned indexers, such as DSA (DeepSeek-AI, 2025), MSA (Lai et al., 2026), and QSA (Qiu et al., 2026), train the indexer and the model so that a small token budget suffices. In a large-scale study of training-free sparse attention, choosing tokens for each query during prefill remained impractical, because estimating importance was costly and kernels could not turn such irregular choices into real speedups (Nawrot et al., 2026). Existing methods therefore select contiguous blocks (Tang et al., 2024), whose tokens are often unrelated to one another, or share one selection across neighboring queries (Jiang et al., 2024; Lai et al., 2025; Jones et al., 2026; Fan et al., 2026a). Methods that cluster similar keys build their index alongside a dense prefill and use it for decoding alone (Liu et al., 2025; Zhu et al., 2026).

We show that, without training, a sparse prefill in which each query selects its own blocks can still run much faster than dense attention. Our method clusters the keys of each input prompt into large semantic blocks, groups of tokens with similar keys regardless of their positions in the sequence. Each query scores the cluster centroids and attends to the original keys and values of the blocks it selects. Unlike contiguous blocks, a semantic block gathers tokens that a query is likely to need together, and unlike earlier clustering methods, the same index sparsifies prefill as well as decoding.

Scoring one centroid per block makes selection cheaper than scoring every key individually. During prefill, queries that select the same block are processed together for that block, as in the KV-outer execution of MoBA and MSA (Lu et al., 2025; Lai et al., 2026). This reuses KV across queries while each query position still has its own selection. If queries are instead processed one tile at a time, each tile reads the union of its queries' blocks, which in our 512K measurement is slower than dense attention. Together, centroid scoring and KV-outer execution address the two obstacles noted above. Larger blocks make both steps cheaper but token selection less precise, and the time they save allows a larger budget. On both Qwen models at 512K, clusters of about 2,048 tokens are faster and more

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.

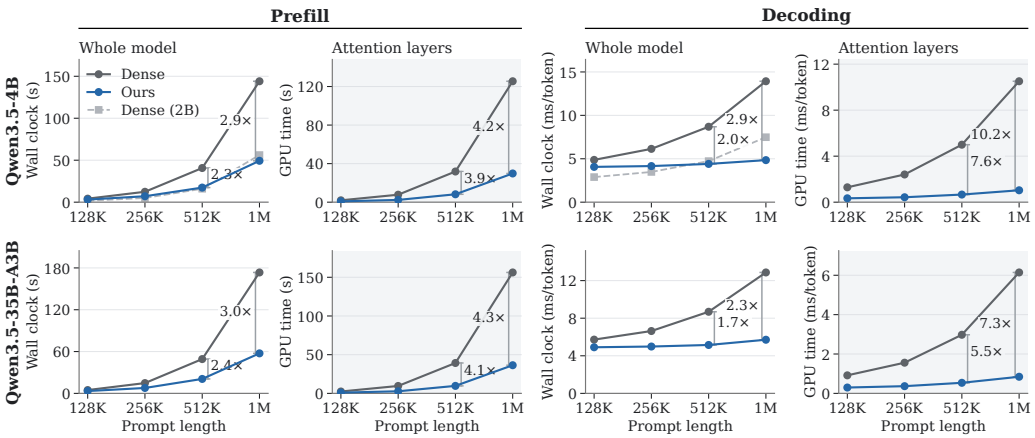

Figure 1: Prefill and decode latency on one H200 with a token budget of $L/16$ for a prompt of L tokens. Whole-model latency is measured by wall clock, and full-attention layers (shaded) by GPU time. The rows are Qwen3.5-4B (top) and 35B-A3B (bottom). Speedups are over dense attention (Tables 13 and 28).

accurate than clusters of 512 tokens given a quarter of the budget, so the time saved by larger blocks is better spent on a larger budget than on finer selection (Section 5.3).

On two hybrid LLMs, Qwen3.5-4B and 35B-A3B, whole-model prefill is 2.3–2.4× faster and decoding 1.7–2.0× faster than dense attention at 512K tokens, and prefill is 2.9–3.0× faster at 1M, with index construction included and dense attention run on the same inference stack (Figure 1). Near the end of a 1M-token prefill, our attention layers run 10.7× faster than dense attention, the same order as the 7.6× reported for QSA, whose sparse attention is trained with the model (Section 2). The 35B-A3B model stays close to dense attention on five RULER tasks (Hsieh et al., 2024) through 512K, on LongBench-v2 (Bai et al., 2025) overall, and on MRCR (OpenAI, 2025) through 256K, and the 4B model, which loses accuracy mainly on multi-key retrieval, still outperforms dense Qwen3.5-2B, whose latency is similar at 512K (Table 1). Hybrid models are increasingly common among top-performing LLMs (MiniMax, 2025; Qwen Team, 2026), yet few studies measure training-free sparse attention on them, and UniPrefill measures these methods only up to 128K tokens, where they barely shorten the prefill (Fan et al., 2026b). In all-attention models most heads attend locally (Zhou et al., 2026), whereas the few attention layers of a hybrid model are its only exact path to distant tokens and only part of its cost, so results on all-attention models or on attention layers alone do not carry over directly (Section 2).

2 Related Work

Many sparse attention methods first score individual tokens or blocks of consecutive tokens, then compute attention over the selected keys and values. DSA (DeepSeek-AI, 2025) and MSA (Lai et al., 2026) use learned indexers to score every visible token, with MSA pooling the resulting scores to select blocks. QSA reduces the number of candidates by pooling indexer keys into 4-token blocks before scoring (Qiu et al., 2026). These systems train both the indexer and the language model, so the model adapts to sparse attention and performs well with budgets as small as 2,048 tokens. On a model of the same family, QSA reports that its attention module, including the indexer, is 7.6× faster than dense attention on the last 16K tokens of a 1M-token prefill with a budget of 2,048 tokens, but not the speed of the whole model. Our method instead leaves the model unchanged and handles a budget 32 times larger. Measured on the last chunk as QSA is, here the last 64K tokens of a 1M-token prefill of Qwen3.5-35B-A3B, where every query sees almost the whole prompt and the sparse gain is largest, our attention layers are 10.7× faster and the whole model 3.9× faster (Figure 15).

Contiguous blocks can also be selected using summaries of the model's existing keys, as in Quest (Tang et al., 2024) and MoBA (Lu et al., 2025). MoBA computes attention block by block, together for all queries that selected a block, as MSA also does (Lai et al., 2026). FlashMoBA speeds

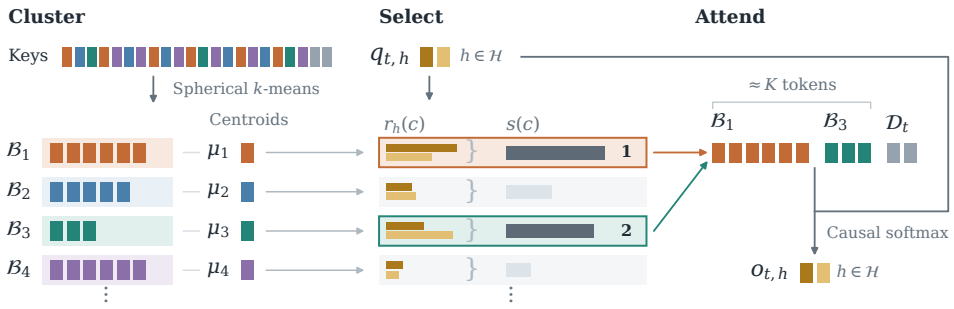

Figure 2: Keys are clustered separately for each layer and KV head. The query heads that share a KV head score the centroids separately (r_h), and the sum of their scores (s) selects the clusters. Each head attends to the original KV of the selected clusters, together with recent tokens $\mathcal{D}_t$, under a causal mask. Only positions at or before the query contribute to the attention sum.

up this computation for small blocks (Xiao et al., 2025). Training-free prefill methods make selection cheaper with structured patterns and block-level estimates, as in MInference (Jiang et al., 2024). FlexPrefill's query-aware pattern scores blocks using pooled queries and keys (Lai et al., 2025), while QuoKA uses representative queries to select a shared KV subset for a chunk (Jones et al., 2026). FlashPrefill and FlashPrefill V2 score mean-pooled key blocks for each tile of queries and keep the blocks above a threshold relative to the maximum score, the latter compensating the rest with their mean keys and values (Fan et al., 2026c;a). A large-scale comparison of training-free methods estimates their cost from FLOPs and memory traffic rather than measured time (Nawrot et al., 2026).

Clustering groups tokens by key similarity rather than sequence position. SAAP fits a key partition on a corpus and trains a query classifier to select clusters while keeping the language model frozen (Mazaré et al., 2025). Squeezed Attention (Hooper et al., 2025), ClusterKV (Liu et al., 2025), and Tactic (Zhu et al., 2026) use query–centroid scores without a learned classifier and attend to the selected original keys and values. Squeezed Attention indexes a reusable context offline, whereas ClusterKV and Tactic build prompt-specific clusters during dense prefill for subsequent sparse decoding. Our method also builds its clusters for each prompt without training, but uses them for the prompt's own prefill as well as its decoding. Its clusters are also much larger, 2,048 tokens in the main setting against 20 to 100 tokens in these methods, which makes centroid scoring and KV-outer attention cheap enough to afford a larger budget, with a separate selection for each query.

The inference-time methods above, which leave the language model unchanged, are evaluated in their own papers on Transformers in which every layer uses softmax attention (Appendix I). UniPrefill, which instead prunes tokens from an intermediate layer onward, measures several of the prefill methods on a linear-attention hybrid up to 128K tokens, where they speed up the time to first token by at most $1.11\times$ (Fan et al., 2026b). Concurrent work on Qwen3.5 trains a selector and keeps the prefill dense (Ahmad & Yun, 2026). When every layer uses softmax attention, most heads attend locally, and turning 85% of them into sliding-window attention loses little accuracy after a few hundred training steps (Zhou et al., 2026). In a hybrid model such as MiniMax-M1 (MiniMax, 2025) or Qwen3.5 (Qwen Team, 2026), a quarter or fewer of the layers use softmax attention, and they are the only layers with exact access to distant tokens, so results on full-attention models do not show how sparse these layers can be made. These layers are also only part of the model's cost, so the attention-only speedups that several of the methods report overstate the gain on a hybrid model.

3 Method

Our method groups prompt keys into semantic blocks, selects blocks for each query, and computes attention over the selected tokens (Figure 2). We choose the block size and token budget together to balance task accuracy and whole-model latency, including the cost of building and using the index. All model weights remain fixed, and no selector is trained.

We describe one full-attention layer and one KV head shared by a group $\mathcal{H}$ of query heads under grouped-query attention (GQA). Let L be the prompt length and d the attention head dimension. We

write $k_i, v_i \in \mathbb{R}^d$ for the key and value at position i, and $q_{t,h} \in \mathbb{R}^d$ for the query at position t in head $h \in \mathcal{H}$. We omit layer and KV-head indices.

3.1 Building semantic blocks

Let $\tilde{k}_i$ and $\tilde{q}_{t,h}$ denote the keys and queries normalized to unit L_2 norm. For each prompt, layer, and KV head, we run spherical k-means on a sample of the normalized keys to obtain C unit-norm centroids $\mu_1, \ldots, \mu_C$. We assign every prompt key to the centroid with the largest inner product and define its block as

$$a_i = \operatorname*{arg\,max}_{c \in \{1,\ldots,C\}} \tilde{k}_i^\top \mu_c, \qquad \mathcal{B}_c = \{i \in \{1, \ldots, L\} : a_i = c\}. \tag{1}$$

Each $\mathcal{B}_c$ contains the positions of tokens assigned to cluster c. We call these groups *semantic blocks* because they group similar keys regardless of their positions in the sequence.

Figure 3 and Appendix J show tokens that the clustering places in the same block. A block can contain many different tokens. In the upper example, the members closest to the centroid are words about the environment from distant passages. In the lower example, they are the leading digits of different identifiers.

A target mean block size u determines the cluster count approximately as $C = L/u$ (Appendix B gives the clustering details). Each layer and KV head has its own partition, and the members of a block are generally scattered across the prompt, so we store each block's keys and values contiguously (Section 3.4).

MRCR · layer 3, KV head 2, cluster 17

Cosine	Position	
0.939	104256	… the shrinking **habitats**, and the urgent …
0.937	83001	… to maintaining **ecosystems** is needed. …
0.936	42982	… helps preserve **biodiversity** and the …
		…
0.685	231403	… Arabic numeral **system** revolutionized …
0.683	161830	… during the 19th **century**, the croissant is …
0.632	138508	… in managing HIV/**AIDS**. The process of …

MK-3 · layer 7, KV head 0, cluster 147

Cosine	Position	
0.983	262904	… special magic uuids for **1**1ac61a0-1483 …
0.982	174418	… special magic uuids for **2**484c754-8d1 …
0.982	353541	… special magic uuids for **1**022e6ce-c52a- …
		…
0.660	523286	… special magic uuid for **4**5c4b52e-fcee-4 …
0.545	77	… 34-87f9-13184e7bd9a1 is: **1**f9dd22f-0d2b …
0.422	40	… special magic uuids for **3**c1b5987-b71a …

Figure 3: Examples of two clusters, with members of high and low cosine similarity to the centroid.

3.2 Selecting blocks and computing attention

Prefill queries are processed in contiguous chunks. Each query attends to all causally visible tokens in its current chunk and selects clusters from the earlier history. During decoding, recent generated tokens are similarly included without selection. Let $\mathcal{D}_t \subseteq \{1, \ldots, t\}$ denote the positions included without selection for the query at position t. The remaining visible positions in cluster c are $\mathcal{B}'_{c,t} = (\mathcal{B}_c \cap \{1, \ldots, t\}) \setminus \mathcal{D}_t$.

We compute softmax scores over centroids within each query head and sum them across the GQA group before selecting blocks. We also apply a size penalty to account for the number of tokens retrieved from each cluster,

$$r_h(c) = \frac{\exp(\tilde{q}_{t,h}^\top \mu_c / \tau)}{\sum_{c'=1}^{C} \exp(\tilde{q}_{t,h}^\top \mu_{c'} / \tau)}, \qquad s(c) = \frac{\sum_{h \in \mathcal{H}} r_h(c)}{(1 + |\mathcal{B}'_{c,t}|)^\beta}. \tag{2}$$

We use temperature $\tau = 0.1$ and size-penalty exponent $\beta = 0.25$.

We select clusters in decreasing order of $s(c)$, taking all tokens in each $\mathcal{B}'_{c,t}$ until their total number reaches or exceeds the budget K. Let $\mathcal{J}_t$ be the clusters selected at position t. The positions used for attention are

$$\mathcal{S}_t = \mathcal{D}_t \cup \bigcup_{c \in \mathcal{J}_t} \mathcal{B}'_{c,t}. \tag{3}$$

This selection is computed for each query position and shared by the query heads in its GQA group. The implementation approximates the score-ordered selection rule with threshold refinement instead of explicitly sorting the scores (Appendix B).

Attention uses the model's original queries, keys, and values,

$$o_{t,h} = \frac{\sum_{i \in \mathcal{S}_t} \exp(q_{t,h}^\top k_i / \sqrt{d})\, v_i}{\sum_{i \in \mathcal{S}_t} \exp(q_{t,h}^\top k_i / \sqrt{d})}. \tag{4}$$

Centroids are used only for selection. For models whose attention uses RoPE (Su et al., 2024) or partial RoPE, clustering and selection can use keys and queries either before or after the rotation, with similar accuracy (Appendix F.1). Our experiments use them before the rotation. Setting $\mathcal{S}_t = \{1, \ldots, t\}$ recovers dense attention.

3.3 Prefill and decoding

At each layer, we build the index from the prompt keys and use it for that layer's prefill and for decoding. For the initial query chunks, we skip selection and use dense attention over the visible prefix, setting $\mathcal{D}_t = \{1, \ldots, t\}$. This avoids selection overhead when the prefix is short (Appendix B gives the chunk size and the cutoff). During decoding, $\mathcal{D}_t$ can contain all generated tokens, as in our main experiments, or only recent ones, with older ones added to the index (Section 5.5).

For simplicity, our main experiments run spherical k-means on keys sampled from the whole prompt. Block selection for a prompt query can therefore depend on later prompt keys, although each query attends only to positions at or before it. This is possible because the whole prompt is given before generation starts, and every generated token still depends only on the tokens before it. Because k-means needs every key of the prompt in this setting, prefill runs one layer at a time over the whole prompt, instead of passing chunks of the prompt through all layers as ordinary chunked prefill does (Appendices B.4 and B.5). Alternatively, running k-means only on the first 64K keys and assigning each later key to its nearest centroid allows ordinary chunked prefill, with similar accuracy on the four 512K tasks we tested (Section 5.4 and Appendix B.5).

Each query scores C centroids instead of the prompt's L keys. Because both C and K grow in proportion to L in our experiments, prefill remains quadratic in prompt length, but with a smaller constant factor than dense attention. Appendix A gives the cost of each step.

3.4 Efficient execution

Efficient dense-attention kernels such as FlashAttention-3 (Shah et al., 2024) reuse each KV tile across many neighboring queries. When sparse attention lets each query choose its own blocks, neighboring queries may select different subsets of KV blocks, reducing this reuse. Making neighboring queries use the same set of blocks improves reuse but prevents them from selecting blocks independently (Jiang et al., 2024; Lai et al., 2025; Jones et al., 2026; Fan et al., 2026a). We instead process each KV block with the queries that selected that block, as MoBA and MSA do for contiguous blocks (Lu et al., 2025; Lai et al., 2026).

For each prefill chunk and KV head, we pair each block's KV with the queries that selected the block. FlashAttention-3 processes these pairs as batches of variable-length sequences. Attention within the current chunk uses causal FlashAttention-3. We combine its output with the block-wise results using their softmax normalizers (Appendix B). Following MSA (Lai et al., 2026), we call this block-wise execution *KV-outer*. On Qwen3.5-4B at 512K, prefill with KV-outer is $2.6\times$ faster than processing neighboring queries over their selected-block union (Appendix E.4).

We pack each block's KV contiguously, preserving token order within the block. Keys from before a prefill chunk then occupy a prefix of each block and can be read directly. We build this block-ordered cache once per layer and use it for both prefill and decoding. Without this layout, gathering the selected KV for every prefill chunk increases 512K prefill time by about 9% (Appendix E.4).

During decoding, each step reads prompt KV from the selected blocks rather than the full prompt. This reduces the prompt tokens read per step from L to roughly K. Fused kernels reduce the overhead of scoring and selecting blocks, and the runtime is integrated into vLLM (Kwon et al., 2023).

With a fixed budget, smaller blocks give finer selection but require scoring more centroids, processing more query–block pairs, and combining more partial results (Xiao et al., 2025). The lower overhead of larger blocks leaves room for a larger budget, which can offset their coarser selection (Section 5.3).

4 Experimental setup

Models and baselines. We evaluate Qwen3.5-4B and Qwen3.5-35B-A3B (Qwen Team, 2026), replacing their full-attention layers (8 of 32 and 10 of 40) while keeping all weights fixed, and

Table 1: Long-context performance at $K = L/16$, with L the upper bound of the length bin for MRCR-8 and of the length band for LongBench-v2, and whole-model prefill latency at 1M. MRCR-8 reports a match ratio, and the other quality scores are percentages. Appendix F.3 gives other budgets.

	RULER-5			MRCR-8			LongBench-v2			Prefill (s)
	256K	512K	1M	128K	256K	512K	≤256K	>256K	all	1M
35B-A3B, dense	85.7	84.4	72.9	0.846	0.655	0.525	64.4	65.2	64.5	173.5
35B-A3B, ours	84.5	84.4	69.9	0.844	0.668	0.459	63.1	61.7	62.9	57.4
4B, dense	84.7	78.4	55.8	0.797	0.574	0.430	54.3	59.2	55.0	144.1
4B, ours	82.7	74.7	49.8	0.788	0.561	0.408	53.9	55.7	54.2	49.3
2B, dense	77.7	67.1	46.5	0.726	0.414	0.246	36.7	35.8	36.6	56.2

also apply the method to a hybrid of another family, Nemotron 3.5 Lightning (NVIDIA, 2026) (Appendix H.2). Dense attention runs on the same vLLM build with chunked prefill, and dense Qwen3.5-2B serves as a smaller-model reference. As a controlled sparse baseline, we replace the semantic blocks with positional blocks, each a run of B contiguous tokens represented by the mean of its keys, and, as block-sparse prefill methods do (Jiang et al., 2024; Lai et al., 2025; Fan et al., 2026a), let g neighboring queries share one block selection to reduce its cost. It uses the same token budget, the same positions $\mathcal{D}_t$ read without selection, and the same attention kernels as our method, so the two methods differ mainly in how blocks are formed and whether queries share a selection. It isolates these two choices rather than reproducing the full selection policy of any one prior method.

Settings. Both Qwen models use the same configuration, with target mean block size $u = 2{,}048$ and budget $K = L/16$ for RULER and latency, and most ablations use the 4B model. MRCR uses $K = L_{\text{bin}}/16$, where L_{bin} is the upper bound of the length bin, and LongBench-v2 uses $K = 16{,}384$ up to 256K and $K = 65{,}536$ above it. The budget excludes the tokens attended without selection and may be exceeded by the last selected block. Beyond the native 256K context, dense and sparse runs both use factor-4 YaRN interpolation (Peng et al., 2024) without its attention scaling, chosen using dense attention (Appendix D.2). MRCR and LongBench-v2 were not used to choose any setting.

Benchmarks. We use RULER (Hsieh et al., 2024), eight-needle MRCR (OpenAI, 2025), and LongBench-v2 (Bai et al., 2025). RULER-5 averages five RULER tasks, namely multi-key retrieval (MK-2 and MK-3), common-word extraction, and HotpotQA, which remain hard at long lengths, and multi-value retrieval, which checks that several answers are recovered. Most other tasks are too easy to separate the methods at 512K, and the 4B model's average over all thirteen stays close to dense attention (Appendix D.1). RULER and MRCR use about 100 examples per task and length, and LongBench-v2 uses the 470 of 503 examples that fit without truncation. RULER uses greedy decoding without thinking, and MRCR and LongBench-v2 use thinking with default sampling. RULER-5 averages three clustering seeds, and MRCR and LongBench-v2 average three runs (Appendix C).

5 Results

5.1 Overall performance

We measure latency at batch size one on a single H200 in BF16, with FlashAttention-3 for prefill and full-step CUDA graphs for decoding in both methods, and report the median of three repetitions after warm-up (Appendix C). Prefill latency includes index construction, KV packing, selection, gathering, and merging. On Qwen3.5-4B and 35B-A3B, whole-model prefill is 2.3–2.4× faster and decoding 1.7–2.0× faster than with dense attention at 512K (Figure 1). The gains grow with context length, reaching 2.9–3.0× for prefill and 2.3–2.9× for decoding at 1M. The full-attention layers themselves prefill 3.9–4.3× faster at these lengths, and the whole-model speedup is lower because the other layers are unchanged (Tables 13 and 28).

Figure 4 shows where the prefill time goes on the 4B model at 512K. Our attention kernels run about 5× faster than dense attention. Index construction, selection, gathering, and merging together take about 1.8 s, of which fitting the centroids is a small part, compared with roughly 25 s saved in

attention computation. The rest of the model is unchanged and takes about half of the sparse prefill time. Besides the selected blocks, our attention kernels also read the whole visible prefix for the first query chunks, and each query's own chunk, so they are not $16\times$ faster at a budget of $L/16$. Processing fewer chunks densely does not reduce latency, because the remaining queries then need selection, gathering, and merging (Appendices E.2 and E.3).

The quality loss differs between the two models (Table 1). On the 35B-A3B model, RULER-5 remains close to dense attention through 512K, with a modest gap at 1M. The 4B model loses more on multi-key retrieval, especially at 1M, while multi-value retrieval and HotpotQA remain close to dense attention across lengths (Appendix D.1). MRCR-8 scores stay close to dense attention through 256K on both models and fall below it at 512K, more on the 35B-A3B model. Overall LongBench-v2 accuracy changes little on either model. A smaller dense model also reduces latency, but the sparse 4B model scores higher than dense Qwen3.5-2B on RULER-5 at every tested length and on LongBench-v2, with similar prefill latency at 512K and lower latency at 1M (Table 1 and Appendix E). On Nemotron 3.5 Lightning, with its own clustering settings, multi-key accuracy falls further at $L/16$, so we use $K = L/8$. Whole-model prefill is then 1.5–2.0$\times$ faster from 256K to 1M, and accuracy at 512K stays within five points of dense attention except on multi-key retrieval (Appendix H.2).

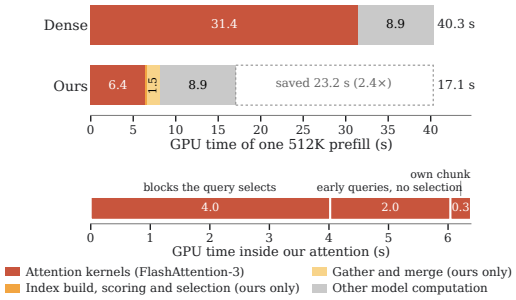

Figure 4: GPU-time breakdown for a 512K prefill on Qwen3.5-4B. The upper panel shows whole-model GPU time, and the lower panel breaks down our attention-kernel time (Table 14). Times are profiler totals, not wall-clock latency.

5.2 Semantic blocks and per-query selection

Two choices shape both the accuracy and the speed of block selection. The first is how blocks are formed, either from B contiguous positions or as semantic clusters. The second is whether each query selects its own blocks or g neighboring queries share one selection, as block-sparse prefill methods do. A group scores each block by the sum of its queries' exponentiated scores, as in FlashPrefill (Fan et al., 2026a), and the other group scores we tried gave similar or lower accuracy (Appendix F.4). Sharing lets the group attend to the same blocks as one dense tile, but each query then reads the tokens chosen for the group rather than for itself. Figure 5 compares these choices at the same token budget on multi-key retrieval, where the 4B model is most sensitive to sparsification. Large positional blocks are fast but retrieve poorly, and smaller ones improve accuracy at a higher execution cost, particularly with per-query selection. At $B = 32$, per-query positional selection is still less accurate than our method and more than $4\times$ slower to prefill. Sharing selections makes positional blocks faster, but the most accurate shared setting ($B = 8$, $g = 32$) remains below our method and is about

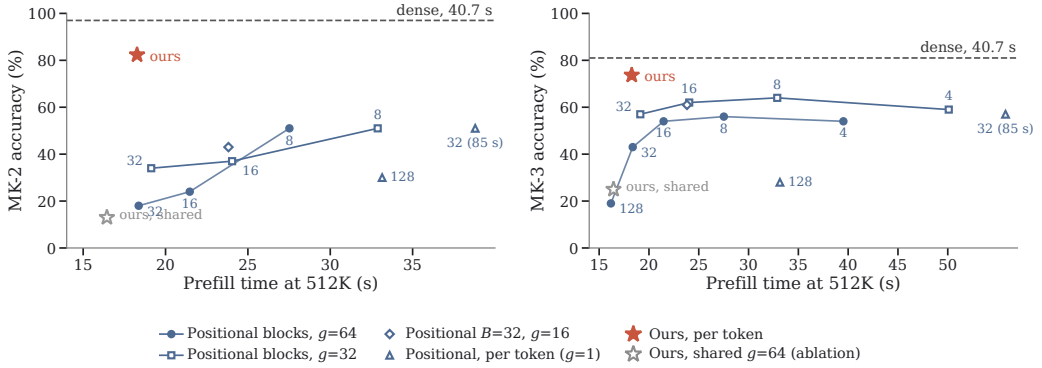

Figure 5: MK-2 and MK-3 accuracy against prefill latency on Qwen3.5-4B at 512K. Numbers are the positional block size B, and g is the number of queries sharing one selection.

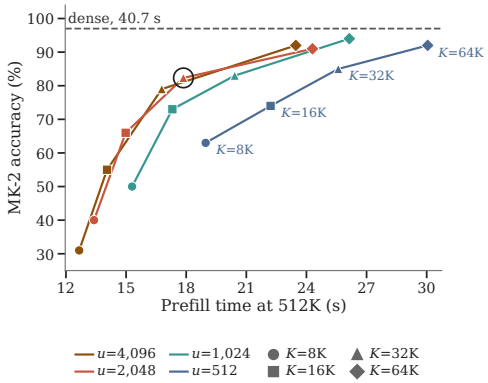

Figure 6: Cluster size and token budget on MK-2 at 512K, Qwen3.5-4B. Larger clusters with a larger budget can improve both accuracy and latency over finer clusters with a smaller budget.

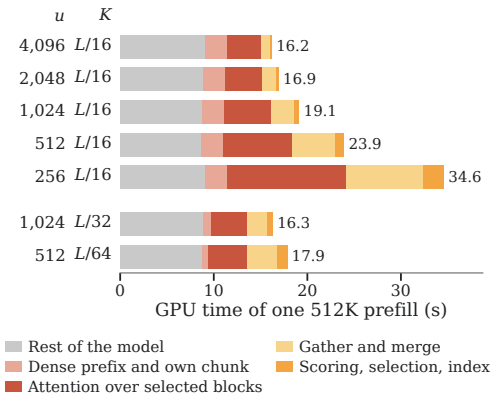

Figure 7: GPU time of one 512K prefill on Qwen3.5-4B by cluster size u and token budget K.

$1.8\times$ slower. On MK-2 the gap is wider, with every positional setting at least 31 points below our method. On Qwen3.5-35B-A3B, the only positional setting we measured that matches our MK-3 accuracy takes $1.8\times$ as long to prefill and is 16 points less accurate on MK-2 (Appendix F.4).

Sharing one selection among 64 queries also hurts semantic blocks, sharply reducing MK-3 accuracy for only a small latency gain. Per-query selection of semantic blocks thus gives the best accuracy–latency trade-off among these configurations. Its blocks are large enough to keep KV-outer execution cheap, whereas small positional blocks multiply the queries to gather and the outputs to merge (Section 3.4). Semantic blocks also leave far fewer candidates to score (256 against 16,384 for 32-token positional blocks at 512K). Several sparse attention methods include the first positions of the prompt in every attention pattern, because attention sinks form there (Xiao et al., 2024; Jiang et al., 2024). Adding such a fixed initial block of 1,024 positions improves both methods, but ours remains 5 points more accurate and over $2.5\times$ faster than the best positional setting (Table 23 and Appendix D.3).

5.3 Block size and token budget

Cluster size and token budget jointly determine the accuracy–latency trade-off. On Qwen3.5-4B (Figure 6), fine clusters are much more accurate at small budgets, but as the budget grows, coarse clusters catch up while staying faster, because cheaper selection and KV-outer attention let them read more tokens in the same time. At 512K with $K = L/16$, clusters of 2,048 tokens nearly match the accuracy of finer ones on this model at lower latency, whereas coarser clusters lose accuracy, so we use this size. Our main configuration is also faster and more accurate than clusters a quarter the size with half the budget. The same grid on MK-3 and on the 35B-A3B model shows the same pattern (Appendix F.2). Coarse clusters do have a cost at a fixed budget. Most of the attention that our selection misses comes from reading whole clusters rather than from how they are scored, but doubling the budget recovers more of it than an oracle that ranks clusters by the dense model's attention (Appendix G.1). In the sparse runtime, that oracle still recovers half of the gap to dense attention on MK-2 (Table 25), so better scoring remains a way to improve retrieval.

Figure 7 shows where smaller clusters lose time. At the same budget, going from 2,048 to 512 tokens per cluster lengthens the 4B model's prefill by 41%, mostly in attention over the selected blocks, gathering, and merging, which grow with the number of query–block pairs, while scoring, selection, and index construction stay small. The 35B-A3B model shows the same breakdown (Figure 12). At a similar prefill time, clusters of 512 tokens with a quarter of the budget spend about as long on attention over the selected blocks as clusters of 2,048 tokens, yet read a quarter as many tokens and spend the rest on gathering, merging, and scoring (Figure 7, bottom rows). With 2,048-token clusters, attention over the selected blocks runs at about 70% of the FLOP rate of dense FlashAttention-3 on

the same GPU, but with 512-token clusters it falls below 40%, because each query–block pair holds fewer keys. Kernels specialized for small blocks (Xiao et al., 2025) could narrow this gap, so the best cluster size depends on the kernels as well as on the model. On Nemotron, going from 2,048 to 512 tokens per cluster adds only 27% (Figure 17) and gains more accuracy than on the Qwen models, so its best clusters mostly hold 512 to 1,024 tokens. These are still far larger than the clusters of earlier clustering methods (Section 2), and cluster size and budget are best chosen together for each model.

5.4 Prompt-specific clustering

While we build centroids for each new prompt, centroids fitted once on a text corpus could be used, as in SAAP, which also trains a query classifier to choose clusters (Mazaré et al., 2025). We vary only the fitting data (Table 2). Centroids fitted on a raw-text corpus retain much of the HotpotQA accuracy but lose almost all multi-key retrieval accuracy, and on common-word extraction they put more than half the prompt into one cluster, which the runtime rejects. Fitting one shared set of centroids to the RULER-5 evaluation prompts recovers much of the multi-key loss and matches fitting each prompt separately on multi-key retrieval, but remains slightly below it on HotpotQA and common-word extraction. Prompt-specific fitting adapts to each input without choosing a matching corpus in advance (Appendix F.5). Fitting on each prompt's first 64K tokens gives similar accuracy and lets the prefill run in ordinary chunks with less memory (Section 3.3 and Appendix B.5).

Table 2: Centroid fitting data on Qwen3.5-4B at 512K ($K = L/16$).

Centroids fitted on	MK-3	MK-2	Hotpot	CWE
Dense attention	81.0	97.0	52.0	62.1
Each prompt (ours)	73.7	82.3	53.3	64.6
Each prompt's first 64K	75.0	88.0	54.0	65.0
A raw-text corpus	3.0	7.0	50.0	*failed*
RULER-5 examples	75.0	82.0	51.0	63.2

5.5 Decoding over long generations

Our main experiments attend to all generated tokens exactly, so each decoding step becomes slower as the generation grows, which matters for the thinking traces of MRCR and LongBench-v2 (blue in Figure 10, Appendix E.5). Older generated tokens can instead be assigned to the existing centroids and added to the index, without refitting and without moving the cache buffers, so the captured decode graph stays valid. When they are added every 1,024 generated tokens and the newest 512 stay outside the index, decode latency stays nearly constant (orange in Figure 10), including over 64K generated tokens after a 1M prompt, and LongBench-v2 accuracy remains similar on both Qwen models (Appendices E.5 and H.1).

6 Limitations and future work

Our experiments use BF16 inference at batch size one on a single H200, and throughput under continuous batching and on other GPU architectures remains to be evaluated. The method retains the full prompt KV, so memory capacity still limits context length and batch size. Fitting the index on the whole prompt raises peak memory above dense chunked prefill, but fitting it on the first 64K tokens allows chunked prefill with similar accuracy (Appendix B.5). Our method can also be combined with other efficiency techniques, such as weight and KV-cache quantization (Liu et al., 2024) and cache compression (Li et al., 2024), which reduce different costs.

Training the model to use sparse attention, as in DSA (DeepSeek-AI, 2025) and QSA (Qiu et al., 2026), is out of our scope and may bring further gains. Our short distillation experiments did not consistently improve retrieval (Appendix G), and longer training on long-context task data may help preserve accuracy under smaller budgets.

7 Conclusion

We presented a training-free sparse attention method that builds semantic blocks from each prompt and uses them for both prefill and decoding. Centroid scoring keeps selection cheap, and KV-outer execution lets each query keep its own selection while still speeding up prefill, addressing the costs of selection and execution identified by a large study of training-free methods (Nawrot et al., 2026). The resulting savings allow a larger token budget and yield whole-model speedups on two hybrid LLMs.

AI USE STATEMENT

In this work, we used generative AI tools to implement the method, including performance optimization, to design and run experiments, including the preparation of benchmark data and proposing or refining hypotheses during the experiments, to analyse and interpret the results, and to support the writing, including translation assistance. We have not used generative AI tools to generate data sets, to develop theoretical models or the method's conceptual framework, or to formulate or prove mathematical claims, and the remaining tasks with required disclosure are not applicable to this work. Additionally, we used generative AI tools to draft parts of the paper and edit it for readability, to suggest its structure, to create and modify figures and tables through scripts, to format references, to search for and summarise related literature, to suggest experimental parameters, and for brainstorming. We have reviewed all AI-assisted work. The code was verified by tests and by consistency checks on its results, and scripts and tests were run so that every number in the paper agrees with the recorded experiments. We take responsibility for the final content of this work, including text, claims, code and figures produced with the aid of generative AI.

ETHICS STATEMENT

This work studies an inference-time efficiency technique for existing language models. It involves no human subjects and no private data. The models and benchmarks are publicly available, and the benchmark prompts are generated from their public generators. Making long-context inference cheaper lowers the cost of the applications the underlying models already serve, and the risks of those applications are unchanged by this work.

REPRODUCIBILITY STATEMENT

Appendix C describes the measurement protocol and the benchmark setup, and Appendix B the runtime configuration of every reported setting.

REFERENCES

Huzama Ahmad and Se-Young Yun. SpotAttention: Plug-in block-sparse routing for pretrained long-context transformers. *arXiv preprint arXiv:2606.22874*, 2026.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025.

Yushi Bai, Qian Dong, Ting Jiang, Xin Lv, Zhengxiao Du, Aohan Zeng, Jie Tang, and Juanzi Li. IndexCache: Accelerating sparse attention via cross-layer index reuse. *arXiv preprint arXiv:2603.12201*, 2026.

DeepSeek-AI. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Dhruv Deshmukh, Saurabh Goyal, Nipun Kwatra, and Ramachandran Ramjee. Kascade: A practical sparse attention method for long-context LLM inference. *arXiv preprint arXiv:2512.16391*, 2025.

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex Attention: A programming model for generating optimized attention kernels. *arXiv preprint arXiv:2412.05496*, 2024.

Qihang Fan, Huaibo Huang, Zhiying Wu, Bingning Wang, and Ran He. FlashPrefill V2: Block-sparse prefill attention for long-context LLM serving. *arXiv preprint arXiv:2608.19758*, 2026a.

Qihang Fan, Huaibo Huang, Zhiying Wu, Bingning Wang, and Ran He. UniPrefill: Universal long-context prefill acceleration via block-wise dynamic sparsification. *arXiv preprint arXiv:2605.06221*, 2026b.

Qihang Fan, Huaibo Huang, Zhiying Wu, Juqiu Wang, Bingning Wang, and Ran He. FlashPrefill: Instantaneous pattern discovery and thresholding for ultra-fast long-context prefilling. *arXiv preprint arXiv:2603.06199*, 2026c.

Coleman Richard Charles Hooper, Sehoon Kim, Hiva Mohammadzadeh, Monishwaran Maheswaran, Sebastian Zhao, June Paik, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. Squeezed attention: Accelerating long context length LLM inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025.

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

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. In *Advances in Neural Information Processing Systems*, 2024.

Dalton Jones, Junyoung Park, Matthew J. Morse, Mingu Lee, Matthew Harper Langston, and Christopher Lott. QuoKA: Query-oriented KV selection for efficient LLM prefill. In *International Conference on Learning Representations*, 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 ACM Symposium on Operating Systems Principles*, 2023.

Xunhao Lai, Jianqiao Lu, Yao Luo, Yiyuan Ma, and Xun Zhou. FlexPrefill: A context-aware sparse attention mechanism for efficient long-sequence inference. In *International Conference on Learning Representations*, 2025.

Xunhao Lai, Weiqi Xu, Yufeng Yang, Qiaorui Chen, Yang Xu, Lunbin Zeng, Xiaolong Li, Haohai Sun, Haichao Zhu, Vito Zhang, Jinkai Hu, Jiayao Li, Rui Gao, Zekun Li, Songquan Zhu, Jingkai Zhou, and Pengyu Zhao. MiniMax Sparse Attention. *arXiv preprint arXiv:2606.13392*, 2026.

Hoang Anh Duy Le, Sahil Joshi, Zeyu Yang, Zhaozhuo Xu, and Anshumali Shrivastava. Scout before you attend: Sketch-and-walk sparse attention for efficient LLM inference. *arXiv preprint arXiv:2602.07397*, 2026.

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. *arXiv preprint arXiv:2404.14469*, 2024.

Guangda Liu, Chengwei Li, Jieru Zhao, Chenqi Zhang, and Minyi Guo. ClusterKV: Manipulating LLM KV cache in semantic space for recallable compression. In *Proceedings of the 62nd ACM/IEEE Design Automation Conference*, 2025.

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.

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. *arXiv preprint arXiv:2502.13189*, 2025.

Pierre-Emmanuel Mazaré, Gergely Szilvasy, Maria Lomeli, Francisco Massa, Naila Murray, Hervé Jégou, and Matthijs Douze. Inference-time sparse attention with asymmetric indexing. *arXiv preprint arXiv:2502.08246*, 2025.

MiniMax. MiniMax-M1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*, 2025.

Piotr Nawrot, Jianing Li, Renjie Huang, Sebastian Ruder, Kelly Marchisio, and Edoardo Ponti. The sparse frontier: Sparse attention trade-offs in transformer LLMs. In *Findings of the Association for Computational Linguistics: ACL 2026*, 2026.

NVIDIA. NVIDIA-Nemotron-3.5-Lightning-30B-A3B. `https://huggingface.co/nvidia/NVIDIA-Nemotron-3.5-Lightning-30B-A3B-BF16`, 2026.

OLMo Team. OLMo 3. *arXiv preprint arXiv:2512.13961*, 2025.

OpenAI. OpenAI-MRCR: Multi-round coreference retrieval. `https://huggingface.co/datasets/openai/mrcr`, 2025.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. YaRN: Efficient context window extension of large language models. In *International Conference on Learning Representations*, 2024.

Zihan Qiu, Zekun Wang, Xiao Li, Yanpeng Li, Yang Xu, Yixuan Wang, Huaqing Zhang, Rui Men, Bochao Mao, Chengruidong Zhang, Fan Zhou, Hao Luo, Haofeng Huang, Haoran Lian, Haoyan Huang, Hongqing Chen, Jianwei Zhang, Jing Xu, Junjie Wang, Langshi Chen, Liangyu Wang, Linlang Jiang, Man Yuan, Minmin Sun, Peng Jin, Siqi Zhang, Siyu Wang, Xingzhang Ren, Yakai Wang, Yi Zhang, Yiming Dong, Yizhong Cao, Yubo Ma, Yunfei Mao, Bo Zheng, and Dayiheng Liu. On the design of Qwen3.8-Next architecture: Evaluation, efficiency, and training stability. *arXiv preprint arXiv:2608.30320*, 2026. Introduces Qwen Sparse Attention (QSA).

Qwen Team. Qwen3.5: Towards native multimodal agents. `https://huggingface.co/collections/Qwen/qwen35`, 2026.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

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*, 2024.

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

Guangxuan Xiao, Junxian Guo, Kasra Mazaheri, and Song Han. Optimizing mixture of block attention. *arXiv preprint arXiv:2511.11571*, 2025.

Yufei Xu, Fanxu Meng, Fan Jiang, Yuxuan Wang, Ruijie Zhou, Zhaohui Wang, Jiexi Wu, Zhixin Pan, Xiaojuan Tang, Wenjie Pei, Tongxuan Liu, Di Yin, Xing Sun, and Muhan Zhang. HISA: Efficient hierarchical indexing for fine-grained sparse attention. *arXiv preprint arXiv:2603.28458*, 2026.

Shang Yang, Junxian Guo, Haotian Tang, Qinghao Hu, Guangxuan Xiao, Jiaming Tang, Yujun Lin, Zhijian Liu, Yao Lu, and Song Han. LServe: Efficient long-sequence LLM serving with unified sparse attention. In *Proceedings of Machine Learning and Systems*, volume 7, 2025.

Jiayi Yuan, Cameron Shinn, Kai Xu, Jingze Cui, George Klimiashvili, Guangxuan Xiao, Perkz Zheng, Bo Li, Yuxin Zhou, Zhouhai Ye, Weijie You, Tian Zheng, Dominic Brown, Pengbo Wang, Markus Hoehnerbach, Richard Cai, Julien Demouth, John D. Owens, Xia Hu, Song Han, Timmy Liu, and Huizi Mao. BLASST: Dynamic BLocked attention sparsity via softmax thresholding. In *Proceedings of Machine Learning and Systems*, 2026.

Yanke Zhou, Yiduo Li, Hanlin Tang, Maohua Li, Kan Liu, Tao Lan, Lin Qu, Yuan Yao, and Xiaoxing Ma. Full attention strikes back: Transferring full attention into sparse within hundred training steps. *arXiv preprint arXiv:2605.16928*, 2026.

Kan Zhu, Tian Tang, Qinyu Xu, Zhan Jin, Yile Gu, Zhichen Zeng, Rohan Kadekodi, Liangyu Zhao, Ang Li, Arvind Krishnamurthy, and Baris Kasikci. Tactic: Adaptive sparse attention with clustering and distribution fitting for long-context LLMs. In *International Conference on Learning Representations*, 2026.

Qianchao Zhu, Jiangfei Duan, Chang Chen, Siran Liu, Xiuhong Li, Guanyu Feng, Xin Lv, Xiao Chuanfu, Dahua Lin, and Chao Yang. SampleAttention: Near-lossless acceleration of long context LLM inference with adaptive structured sparse attention. In *Proceedings of Machine Learning and Systems*, volume 7, 2025.

A COST OF THE INDEX

This section analyzes the computational cost of one sparse attention layer. Section 3 describes a single KV head, and here we include all heads. The layer has H_{kv} KV heads, each shared by a group $\mathcal{H}$ of $G = |\mathcal{H}|$ query heads, so it has $H_q = GH_{kv}$ query heads. The prompt has L tokens, queries and keys have dimension d, and each KV head has $C \approx L/u$ semantic blocks of mean size u. We divide the cost into building the index, scoring centroids, selecting blocks, and attending to the selected tokens. Table 14 gives the measured GPU time for one 512K prefill, with the gathering and merging around attention as a separate row.

Dense attention. Dense attention costs $\Theta(H_q L d)$ per generated token and $\Theta(H_q L^2 d)$ over a prefill.

Building the index. Building the index is negligible in practice, taking about 0.2% of the GPU time of a 512K prefill (Table 14). It is built once per layer and prompt. Normalizing the keys costs $O(H_{kv} L d)$. Spherical k-means with I iterations on S sampled keys costs $O(H_{kv} ISCd)$, and assigning all L keys to their nearest centroids costs $O(H_{kv} LCd)$. Reordering the KV by block moves $O(H_{kv} Ld)$ values. We use $I = 8$ and $S = \min(L, \max(8{,}192, 32C))$ (Appendix B), so $IS < L$ and assigning the keys costs more than fitting the centroids at every length we report.

Scoring. Scoring is also minor in practice, and together with selection it takes about 1.4% of the same prefill time. Each query head takes inner products with all C centroids, which costs $O(H_q Cd)$ per query position and $O(H_q LCd)$ over a prefill. The factor is H_q rather than H_{kv} because Equation 2 takes a softmax in each query head before summing over the group.

Selection. Selection costs much less than scoring and adds little in practice. It uses the summed scores, so it runs once per KV head and query position. Sorting the C scores would cost $O(H_{kv} C \log C)$ per position. The implementation instead refines a score threshold in $J = 16$ passes (Appendix B), which costs $O(H_{kv} JC)$ and needs no sort.

Attention. Attention is the largest part of the cost that remains. At 512K, it takes one fifth of the time of dense attention, yet still about 37% of our prefill GPU time, a third of which is the dense prefix. Gathering queries and merging partial outputs add another 9%.

Each query attends to its selected blocks and to $\mathcal{D}_t$. Selection stops as soon as the selected blocks reach the budget K, so they exceed it by at most one block. With $u_{\max}$ the size of the largest block, a query therefore attends to at most $K + u_{\max} + |\mathcal{D}_t|$ tokens. This bound holds for the exact selection rule. The largest block depends on the partition and can be much larger than the mean size u. In decoding, $\mathcal{D}_t$ holds all generated tokens, so a step costs $O(H_q d(K + u_{\max} + |\mathcal{D}_t|))$ and grows with generation length. In prefill, $\mathcal{D}_t$ is the visible part of the query's own chunk of width w, so attention over the prompt costs $O(H_q dL(K + u_{\max} + w))$. Query chunks that start at position $4K$ or earlier skip selection and attend to their whole prefix (Section 3.3), which adds $O(H_q d \min(L, 4K + w)^2)$. This cutoff is set for speed, not accuracy. Lowering it from $4K$ to K barely changes MK-3 and MK-2 accuracy at 512K (Appendix E.3), so it could equally be held at a fixed length.

Growth with prompt length. Our experiments fix u, so that $C \approx L/u$, and set $K = L/16$. Each query then scores about L/u centroids instead of L keys and attends to roughly $L/16$ tokens. Scoring and selected attention therefore grow quadratically over a prefill, as dense attention does, but with much smaller constants, and linearly per generated token. A budget proportional to L is a choice made for accuracy, not a requirement of the method. Prefill becomes subquadratic if K grows more slowly than L and C grows as well, but more slowly than L, with the largest block staying near the mean size L/C. A large study of training-free methods finds that longer sequences tolerate a smaller attention budget relative to their length, which suggests such a budget (Nawrot et al., 2026). How far the budget can shrink in our setting depends on the model and task. Halving it leaves LongBench-v2 accuracy on Qwen3.5-35B-A3B nearly unchanged, while multi-key retrieval and MRCR-8 at 512K lose accuracy (Tables 22 and 31).

Comparison with per-token indexers. Several recent methods select tokens with a lightweight indexer that scores every earlier token. DSA scores tokens directly (DeepSeek-AI, 2025), and MSA

scores them before pooling the scores into blocks (Lai et al., 2026). Even with index vectors of a smaller dimension $d_r < d$ and one index query per KV head, this scoring costs $\Theta(H_{kv} L d_r)$ per decoded token and $\Theta(H_{kv} L^2 d_r)$ over a prefill. Prefill therefore stays quadratic however few tokens the attention reads (Lai et al., 2026; Xu et al., 2026). Our scoring instead compares each query head with C centroids rather than L tokens. At equal head count and dimension, this is cheaper by a factor of about u, and it becomes subquadratic when C grows more slowly than L.

Memory. The index holds $O(H_{kv} C d)$ centroid values and $O(H_{kv} L)$ block assignments. The sparse layers store their prompt KV once, in block order, instead of in the paged cache. Appendix B.5 measures the rest, including temporary activations and kernel workspace.

B Implementation details

B.1 Routing and clustering

Clustering and selection use the pre-RoPE queries and keys of each head, 256-dimensional vectors normalized as whole vectors. Each KV head has a separate prompt partition. For a target mean block size u, the number of clusters is $C = \min\big(L,\ \max(128,\ 2^{\lceil \log_2(L/u) \rceil})\big)$. Table 3 lists the resulting cluster counts at $u = 2{,}048$. At 128K, $L/u = 64$, so $C = 128$ and the mean block size is 1,024, half the target. Centroids start from uniformly sampled keys and are refined for eight iterations on the sample in the table, with empty clusters re-seeded from it. All prompt keys are assigned after fitting.

In the fit-budget sweep, the sample count ranges from 2,048 to all prompt keys and the iteration count from two to sixteen. MK-3 accuracy at 512K ranges from 71% to 78% over 100 items, while prefill latency varies by at most six percent. A fixed seed controls sampling, but numerical changes in model states or execution can still affect the resulting partition.

The PCA ablation uses leading eigenvectors of the uncentered key Gram matrix, fitted on approximately one million keys per layer and KV head from raw-corpus documents. It projects both queries and keys before whole-vector normalization. These projections are corpus-fitted, although they do not use attention distillation. Our main configuration uses no projection and requires no such fit.

B.2 Selection and attention

The prefill selector uses 16 threshold-refinement steps to approximate the score-ordered budget selection in Equation 3. Its size penalty uses the visible block sizes $|\mathcal{B}'_{c,t}|$ of Equation 2, and blocks with no visible token are never selected. The selected clusters are represented by a bitmask per query and KV head. Counts and prefix sums convert this to cluster-major query lists and query-major merge slots, without materializing the full Boolean selection matrix or sorting partial outputs afterward. Within each block of this cache, tokens stay in position order, and every block starts at a page boundary. Each prefill chunk passes the exact count of preceding tokens in every cluster to FlashAttention-3, including a partially filled final page. The selected history is therefore entirely before the chunk, and only the separate current-chunk call needs a triangular causal mask. The GQA heads are folded into query rows for the history call. A query can receive outputs from several blocks and from the region it attends without selection, and these cannot be averaged because each carries a different amount of softmax mass. If region j returns output o_j and log-sum-exp ℓ_j, the combined output is $o = \sum_j \exp(\ell_j - \ell)\, o_j$ with $\ell = \operatorname{logsumexp}_j \ell_j$. The composition runs in FP32 and the result is cast back to BF16, which recovers attention over the selected union up to floating-point rounding and adds no further sparsity approximation. Apart from the finite precision of the threshold, which can move the budget boundary slightly, this computes attention over the positions of Equation 3.

The decode backend of the latency runs uses five launches per sparse layer, namely suffix write, query normalization, fused scoring and selection, split attention, and output merge. Selection directly emits contiguous cluster ranges. The attention kernel reads selected prompt ranges and the tiles of the generated suffix in the same launch, then merges their softmax statistics. All live lengths are read on device and workspace is reused, avoiding per-step host reads and allocations. The RULER and MRCR-8 runs and the 4B model's LongBench-v2 runs in Table 1 use a fused decode backend with more launches per layer, which selects the same clusters, under vLLM's piecewise CUDA graphs.

Table 3: Clustering per KV head at $u = 2{,}048$, at the four prompt lengths we report.

Prompt length	Clusters	Mean block size	Keys used to fit
128K	128	1,024	8,192
256K	128	2,048	8,192
512K	256	2,048	8,192
1M	512	2,046	16,384

Table 4: Main Qwen3.5-4B runtime parameters. Benchmark-specific budgets are described in Section 4.

Quantity	Value
Replaced layers	Eight full-attention layers of 32
Query / KV heads	16 / 4, head dimension 256
Routing	Pre-RoPE Q/K, whole-vector normalization, identity map
Target mean block size	$u = 2{,}048$, which sets the cluster count of Table 3
Clustering	Eight iterations, $\min(L, \max(8{,}192, 32C))$ sampled keys. The main RULER results average seeds 0–2, and other results use seed 0 unless stated otherwise (Appendix C.4)
Score parameters	$\tau = 0.1$, $\beta = 0.25$
Length-controlled budget	$K = L/16$ for RULER and the main latency sweep
Query chunk	8,192 positions
Dense prefix	A query chunk starting at position $4K$ (four times the budget) or earlier attends to its whole prefix without selection
Attended without selection ($\mathcal{D}_t$)	The visible part of the current chunk in prefill, every generated token in decoding
Numerics	BF16 attention, FP32 score aggregation and output merge
Short prompts	Sparse attention at every measured length, with the runtime's dense fallback for short prompts switched off
Decode graph	The whole decode step captured as one CUDA graph in the latency runs, for dense and ours
Prefill schedule	Sliced (32,768-position slices) at 1M, for the task evaluations above 256K, and for the main RULER-5 evaluations at every length (Appendix B.4). Single-pass for the latency sweep through 512K and the other task evaluations through 256K

Table 4 summarizes the resulting runtime parameters. The runtime can leave prompts below a length threshold to dense attention, since selection saves little there. All our measurements set this threshold to zero, so every reported prompt, including the 64K and 128K ones, uses sparse attention in the replaced layers.

B.3 Compilation and decode graphs

Measurements use `TORCHINDUCTOR_DETERMINISTIC=1`, with combo kernels and their benchmarking disabled. These settings reduce numerical differences between independent compilations, but do not guarantee bitwise agreement between different execution schedules. Prefill uses piecewise graphs where applicable. For the main decode latency results, both methods use vLLM's `FULL_DECODE_ONLY` mode, which captures the whole decode step as one CUDA graph. Each sparse layer has a fixed-address arena containing prompt KV, centroids, posting lists, generated-token storage, and workspace. The graph is captured at engine warm-up and reused after the prompt's index is copied into the arena. One-time graph capture is excluded from steady decode latency for both methods.

B.4 Prefill schedules

An index fitted on the whole prompt needs all of a layer's prompt keys before that layer's queries can be routed. The runtime therefore cannot use ordinary chunked prefill, which passes one token chunk through all layers before the next chunk. Instead, every layer processes the whole prompt before the next layer starts, in one of two ways (Figure 8). *Single-pass* prefill runs the whole prompt

Table 5: Qwen3.5-4B memory on one H200, in GiB. Dense totals are derived request-sized estimates that exclude excess preallocated cache capacity, while sparse peaks are measured allocated memory, so dense and sparse rows are accounted differently.

Method	Length / schedule	Weights	KV cache etc.	Transient	Total
Dense (derived)	512K / chunked	8.35	16.00	1.58	25.93
Ours (measured)	512K / single-pass	8.35	20.84	53.50	82.69
Ours (measured)	512K / sliced	8.35	20.84	11.91	41.09
Ours, prefix index (measured)	512K / chunked	8.35	20.86	6.43	35.64
Dense (derived)	1M / chunked	8.35	31.97	1.58	41.90
Ours (measured)	1M / sliced	8.35	39.65	22.37	70.37
Ours, prefix index (measured)	1M / chunked	8.35	39.72	11.89	59.96

through the model as one forward pass, so every projection and feed-forward call processes all prompt positions at once and holds full-length activations. *Sliced* prefill loops over the layers itself instead of calling the model's forward pass once, and processes each layer in slices of 32,768 positions. A full-attention layer first projects only the keys and values of every slice, builds the index, and packs the block-ordered KV cache, and a second pass over the slices projects the queries and the output gate, performs attention, and writes the outputs. Linear-attention layers carry their recurrent and convolution states between slices, and feed-forward layers and normalization are sliced as well. Both schedules compute the same attention up to numerical differences, with selection and attention in 8,192-position query chunks inside each layer, and differ in peak memory (Table 5). Sliced prefill is required where single-pass prefill exceeds device memory, and Table 4 lists which schedule each measurement uses. The schedule is selected outside the compiled decode path, and elementwise operators are compiled separately for the sliced path.

Dense attention, chunked

```
for each 16K-token chunk:
  for each layer:
    Q, K, V <- project(chunk)
    attend(chunk, KV so far)
    feed-forward(chunk)
```

Ours, single-pass

```
for each layer:
  Q, K, V <- project(all)
  fit index on all keys
  for each 8K query chunk:
    select, attend
  feed-forward(all)
```

Ours, sliced

```
for each layer:
  for each 32K slice:
    K, V <- project(slice)
  fit index on all keys
  for each 32K slice:
    Q <- project(slice)
    for each 8K query chunk:
      select, attend
    feed-forward(slice)
```

Figure 8: The order of operations in one prefill, shown for a full-attention layer and its feed-forward block, where "all" is every prompt position. Linear-attention layers follow the same order without the index.

B.5 MEMORY

In the runs measured here, as in all latency runs, the sparse layers declare no paged cache to vLLM. Their block-ordered KV cache is the only copy of the prompt KV, with a reserve for generated tokens and workspace. The 4B model's single-pass 128K and 256K RULER runs also write the prompt KV to vLLM's paged cache, which changes memory but not the computation. Table 5 lists peak memory against dense chunked prefill. For the sparse rows, "KV cache etc." is allocated memory that remains after the prompt is processed, excluding the weights, and is mostly the KV cache. Transient is peak allocated memory minus the weights and this memory.

The additional memory over dense chunked prefill has two sources. The sliced schedule of Appendix B.4 holds the residual stream for the whole prompt and, until a layer's index is built, that layer's keys and values, which grow with prompt length. The KV-outer attention of Section 3.4 holds, for one query chunk and KV head at a time, the gathered queries and their partial outputs, which grow with the query chunk and the number of blocks its queries selected. The 512K latency runs use single-pass prefill, whose peak is listed separately and is set by the full-length feed-forward intermediates. Both methods fit on one H200 at every length measured. Sparsity does not reduce memory here, because the full prompt KV is kept.

The prefix-index rows fit the centroids on the prompt's first 64K tokens, assign every later key to them as it arrives, and run the prompt through ordinary chunked prefill with 8K-token chunks, so no full-length activation is held. What remains above dense is the output buffer and the attention kernels' per-chunk buffers. This prefill is about a sixth slower than the sliced one, and its partition gives accuracy on the four RULER tasks similar to the full-prompt index's (Table 2). Running this prefill itself gives an MK-3 accuracy of 74.0 at 512K, close to the 75.0 of the same partition in Table 2. The reported results use the full-prompt index.

C MEASUREMENT PROTOCOL

C.1 LATENCY

All headline latency measurements use one request on one H200 and BF16 weights. Both methods use FlashAttention-3 for prefill, and our decoding uses the fused backend of Appendix B. Each latency measurement uses one warm-up followed by three measured repetitions. Ours is timed on two different prompts of each length through 512K and on one prompt at 1M. Dense attention is timed on two prompts at 512K and on one prompt at 128K, 256K, and 1M. Where there are two prompts, we report their mean. The 35B-A3B model is timed on one prompt at every length under the same protocol. Both decode paths use full-step CUDA graphs. Sparse latency uses single-pass prefill through 512K on the 4B model and through 256K on the 35B-A3B model, with sliced prefill at longer lengths. The dense baseline uses vLLM's chunked prefill. Layer timings sum kernels attributed to the attention backend, with decode accounting over the final 32 generated steps. They are profiler totals, not whole-model wall-clock measurements.

C.2 QUALITY

RULER uses cached prompts, 100 items per task and length, with greedy decoding and thinking disabled. The two models share the tokenizer and so the prompts. The 4B model's task evaluations above 256K tokens, on RULER, MRCR-8, and LongBench-v2, use sliced prefill, as do its main RULER-5 evaluations at 128K and 256K, unlike its single-pass 512K latency run. All sparse-attention task evaluations of the 35B-A3B model use sliced prefill. The two schedules implement the same selection policy but can differ numerically. The sliced schedule of these runs prefills 512K tokens in 18.5 seconds (Table 16) with a peak of 41.1 GiB (Table 5). We use the eight-needle split of MRCR at revision `f4c69fa` (8 December 2025), which follows the 5 December 2025 regeneration of the needle files. MRCR uses the model's thinking sampling preset. Its length bins keep the official boundaries, but each prompt is assigned to a bin by its length in Qwen3.5 tokens after the chat template, with 100 items at 128K and 512K and 99 at 256K. Its generation limit accommodates the longest answer and a 32,768-token thinking budget. LongBench-v2 uses 470 retained items, thinking enabled, and at most 40,960 generated tokens. Its sampling parameters are temperature 1.0, top-p 0.95, top-k 20, min-p 0, and presence penalty 1.5. The MRCR-8 and LongBench-v2 results in the main table average three runs, each with its own sampling seed and, for our method, its own clustering seed. For LongBench-v2, we keep the items whose raw prompts fit within 983,040 tokens. The token budget is set by length for RULER, by bin upper bound for MRCR, and by the two fixed length bands for LongBench-v2 (Section 4). Independent quality jobs may run concurrently on different GPUs, but each model instance uses one GPU.

C.3 PROMPT LENGTHS

The prompts come from lm_eval's port of the RULER generator, with NVIDIA's filling for common-word extraction beyond 128K, where the port's word list runs out. The measured median RULER-5 prompt lengths are within five percent of nominal at 128K and two percent from 256K onward, except multi-value retrieval at 1M, whose generator runs out of material at about 647K tokens. Of the other tasks at 512K, frequent-word extraction is the one the generator sizes by words rather than tokens, and its prompts run from 418K to 505K tokens with a median of 476K. The budget follows the nominal length. SQuAD QA is not in RULER-5 because filling it repeats the answer paragraph, about two and a half times at 512K and five times at 1M, so raising the nominal length also changes how often the answer appears.

Table 6: The clustering seed and the seeding procedure, with RULER multi-key accuracy in percent at $K = L/16$ and $u = 2{,}048$, 100 items per task and length.

Model	Task	Length	Seeding	Mean (s.d.)
Qwen3.5-4B	MK-2	512K	random	82.3 (1.2)
			farthest	81.0 (0.0)
			k-means++	81.7 (2.3)
	MK-3	512K	random	73.7 (3.1)
			farthest	78.0 (2.0)
			k-means++	73.7 (1.5)
Nemotron 3.5	MK-3	256K	random	73 (1 seed)
			farthest	86 (1 seed)
		512K	random	84 (1 seed)
			farthest	80 (1 seed)

Table 7: RULER results of our method in the main table, Qwen3.5-4B at $K = L/16$ and $u = 2{,}048$, 100 items per task and length. Each entry is the mean (sample s.d.) over three clustering seeds. RULER-5 is the mean over the five tasks, and its s.d. is computed over the three seeds' five-task means.

Prompt	MK-3	MK-2	MV	CWE	HotpotQA	RULER-5
128K	100.0 (0.0)	98.7 (1.5)	99.8 (0.3)	78.3 (0.5)	57.3 (1.2)	86.8 (0.3)
256K	97.7 (0.6)	91.0 (2.0)	99.3 (0.4)	70.3 (1.4)	55.0 (1.0)	82.7 (0.1)
512K	73.7 (3.1)	82.3 (1.2)	99.5 (0.7)	64.6 (0.7)	53.3 (2.3)	74.7 (1.0)
1M	20.0 (1.0)	59.3 (1.5)	99.7 (0.1)	24.8 (2.2)	45.3 (2.1)	49.8 (0.6)

C.4 Clustering seeds and seeding

The partition is fitted by k-means from a seeded random start, and the seed matters at the item level. Over three seeds on the same 512K items, MK-3 accuracy varies by six points and MK-2 accuracy by two, and up to 14 percent of the items of either task change between correct and incorrect from one seed to another (Table 6 gives the means). Every RULER-5 result of our method in the main tables, on the 4B model and on the 35B-A3B model, is therefore the mean over three seeds of the runtime's seeding (Table 7, and the 35B-A3B model's three seeds are averaged in Table 9). Unless stated otherwise, the appendix tables that compare configurations use one clustering seed, and their differences should be read against the spread across seeds in Table 7. Where such a table measures the main configuration in the same run as its variants, the main configuration's row keeps that run's value, which differs by a few points between tables because the fitted partition differs between runs (Appendix B). Where the main configuration stands alone as a reference, its row is the three-seed mean. Three seedings are compared. Random is the runtime's seeding, a random draw from a sample of 8,192 keys. k-means++ is D^2 sampling on the same sample. Farthest is greedy farthest-point seeding on a sample of 65,536 keys for Qwen seed 0 and 262,144 keys for seeds 1 and 2, and 1,024 keys per centroid on Nemotron. Neither alternative seeding is a consistent improvement in the mean. Greedy farthest-point seeding on a larger fit sample raises MK-3 and lowers MK-2, k-means++ seeding on the runtime's sample leaves MK-3 unchanged and lowers MK-2 slightly, and on Nemotron farthest-point seeding moves the two lengths in opposite directions. Farthest-point seeding also narrows the spread, with fewer items changing their outcome between its seeds on both tasks, whereas k-means++ seeding changes about as many items as random seeding. The runtime keeps random seeding, and single-seed appendix results use seed zero unless stated otherwise. This spread limits conclusions drawn from small differences between configurations.

D QUALITY IN DETAIL

D.1 RULER RESULTS BY TASK

Table 8: Every RULER task on Qwen3.5-4B at a nominal 512K, in percent, 100 items per task, $K = L/16$. Ours on the starred tasks and SQuAD is the mean over three clustering seeds, on the others one seed. The second row is the median length of each task's prompts in thousands of tokens (Appendix C.3). S1–S3 are the single-needle tasks, MK-1 to MK-3 multi-key, MQ multi-query, MV multi-value, VT variable tracking, CWE and FWE common- and frequent-word extraction, SQuAD the QA task with its paragraphs repeated, and Hotpot HotpotQA. The starred tasks form RULER-5.

	S1	S2	S3	MK-1	MK-2*	MK-3*	MQ	MV*	VT	CWE*	FWE	SQuAD	Hotpot*	Mean
Prompt (K tokens)	524	524	524	524	524	523	524	524	524	523	476	523	517	
Dense	100.0	100.0	99.0	99.0	97.0	81.0	100.0	99.8	100.0	62.1	99.0	80.1	52.0	89.9
Ours	100.0	99.0	99.0	97.0	82.3	73.7	100.0	99.5	100.0	64.6	96.3	78.6	53.3	88.0

Across all thirteen RULER tasks at 512K the 4B model's macro average stays close to dense attention (Table 8). Most tasks omitted from RULER-5 are near ceiling for both configurations. The full task results also show that the largest losses occur on multi-key retrieval, where the needles of multi-key 2 are identified by words and those of multi-key 3 by UUIDs. RULER-5 focuses on the harder tasks and retains multi-value retrieval to check whether multiple answers can still be recovered. Common-word extraction and SQuAD QA are on prompts filled to 512K as the official generator fills them (Appendix C.3). The filled SQuAD prompt holds every paragraph of its pool about two and a half times, which is why that task is not in RULER-5.

The italic rows under the 4B model are the dense Qwen3.5-2B model on the same items, a reference for how much a smaller dense model loses. Both multi-key tasks lose more than the others from 256K on, multi-key-2 by the widest margin (Table 9). HotpotQA and common-word extraction do not deteriorate through 512K, but their small changes do not establish that sparsity improves those tasks, and both fall a few points below dense attention at 1M.

D.2 POSITIONAL SCALING BEYOND THE NATIVE CONTEXT

YaRN changes position interpolation and attention scaling (Peng et al., 2024). The runtime's additional factor $\alpha = 1/\mathrm{mscale}(s)$ cancels the latter while retaining interpolation. Table 10 compares these settings on matched 512K prompts on the 4B model. The 35B-A3B model uses the chosen setting without a sweep of its own. Unless stated otherwise, the interpolation factor is four.

The setting was chosen by the sweep's five-task mean, whose QA and common-word scores were measured on the port's capped prompts (Appendix C.3). The multi-key tasks kept in Table 10 rank the settings the same way. The dense sweep was measured separately from the main table, which accounts for its small difference in score, and the bold row of our method is an earlier single-seed run of the main configuration at 512K. Interpolation without extra attention scaling performs best among the tested dense settings and also improves sparse attention. The reference model, Qwen3.5-2B, prefers the same setting by a similar margin, so its rows elsewhere use it. The benefit is therefore not specific to sparsification. We have not established why this checkpoint prefers the modified scaling, and do not infer a cause from its partial rotary embedding.

D.3 OUR METHOD WITH A FIXED INITIAL BLOCK

Many attention heads put much of their weight on the first positions of a sequence, which Xiao et al. (2024) call attention sinks, and several sparse attention methods therefore include these positions in every attention pattern (Xiao et al., 2024; Jiang et al., 2024). Our method can do the same with a fixed initial block. The first 1,024 prompt positions are then left out of the clusters, and every query selects this block first, in prefill and in decoding. The block counts toward the budget K, so the clusters fill the remaining tokens. Nothing is retrained, and all other settings follow the main results, which do not use the initial block. Section 5.2 adds the same block to the positional baseline (Table 23).

Tables 11 and 12 compare our method with and without the initial block on the same items. The initial block helps most where our method loses most to dense attention. On Qwen3.5-4B it recovers

Table 9: RULER-5 by task and length on the three models, in percent, on 100 examples per task at $K = L/16$. For the Qwen models, the ours rows are means over three clustering seeds, and the 35B-A3B model also has multi-key rows at half and a quarter of the budget at 512K. Nemotron is measured at 512K with one seed, and its last row doubles the budget. Common-word extraction beyond 128K is on the filled prompts (Appendix C.3).

Prompt	K	Method	Multi-key-3	Multi-key-2	Multi-value	Common-word	HotpotQA	Mean
Qwen3.5-4B								
128K	8,192	Dense	100.0	100.0	100.0	78.7	58.0	87.3
		Ours	100.0	98.7	99.8	78.3	57.3	86.8
		2B dense	100.0	100.0	99.8	77.8	52.0	85.9
256K	16,384	Dense	100.0	100.0	100.0	68.3	55.0	84.7
		Ours	97.7	91.0	99.3	70.3	55.0	82.7
		2B dense	97.0	99.0	100.0	46.7	46.0	77.7
512K	32,768	Dense	81.0	97.0	99.8	62.1	52.0	78.4
		Ours	73.7	82.3	99.5	64.6	53.3	74.7
		2B dense	77.0	98.0	96.5	17.8	46.0	67.1
1M	65,536	Dense	26.0	76.0	100.0	28.9	48.0	55.8
		Ours	20.0	59.3	99.7	24.8	45.3	49.8
		2B dense	8.0	88.0	97.8	8.0	31.0	46.5
Qwen3.5-35B-A3B								
128K	8,192	Dense	100.0	100.0	100.0	79.7	63.0	88.5
		Ours	98.7	100.0	99.0	82.2	63.7	88.7
256K	16,384	Dense	100.0	99.0	100.0	69.5	60.0	85.7
		Ours	96.3	99.0	99.4	68.6	59.3	84.5
512K	32,768	Dense	100.0	99.0	100.0	65.9	57.0	84.4
		Ours	98.3	98.3	98.4	66.9	60.0	84.4
	16,384	Ours, $K = L/32$	70.0	88.0	–	–	–	–
	8,192	Ours, $K = L/64$	19.0	58.0	–	–	–	–
1M	65,536	Dense	84.0	91.0	99.8	39.9	50.0	72.9
		Ours	76.0	88.7	99.8	35.1	50.0	69.9
Nemotron 3.5 Lightning								
512K	32,768	Dense	100.0	99.0	100.0	95.7	61.0	91.1
		Ours	84.0	49.0	71.5	92.4	53.0	70.0
	65,536	Ours, $K = L/8$	90.0	81.0	95.5	95.1	56.0	83.5

Table 10: Qwen3.5-4B and Qwen3.5-2B under alternative context-extension settings at 512K, with multi-key and multi-value accuracy in percent, $n = 100$ per task, one clustering seed. Unqualified YaRN rows use factor four, and interpolation-only cancels attention-temperature scaling.

Configuration	MK-3	MK-2	MV
Dense, no extension	56.0	90.0	98.5
Dense, YaRN	58.0	84.0	99.8
Dense, YaRN $s = 2$	62.0	95.0	99.8
Dense, $s = 2$, interp only	69.0	96.0	99.8
Dense, interpolation only	**80.0**	97.0	99.8
Ours, no extension	46.0	66.0	99.8
Ours, YaRN	62.0	62.0	98.8
Ours, interpolation only	**73.0**	**86.0**	99.8
Qwen3.5-2B dense, YaRN	51.0	80.0	98.3
Qwen3.5-2B dense, interpolation only	**77.0**	**98.0**	96.5

much of the multi-key loss at 512K and raises RULER-5 at every length. On Qwen3.5-35B-A3B, which is already close to dense attention, it changes little. On Nemotron 3.5 Lightning it improves multi-value retrieval but lowers multi-key-2 accuracy. On MRCR-8 and LongBench-v2 the effect is mixed. LongBench-v2 is slightly higher on both Qwen models, while MRCR-8 is higher on the 35B-A3B model and lower on the 4B model at 512K.

Table 11: RULER-5 by length and its five tasks at 512K, without and with the initial block, on Qwen3.5-4B, Qwen3.5-35B-A3B, and Nemotron 3.5 Lightning.

	RULER-5				Tasks at 512K				
Method	128K	256K	512K	1M	MK-3	MK-2	MV	CWE	HotpotQA
Qwen3.5-4B									
Dense	87.3	84.7	78.4	55.8	81.0	97.0	99.8	62.1	52.0
Ours	86.8	82.7	74.7	49.8	73.7	82.3	99.5	64.6	53.3
Ours with initial block	87.0	83.6	77.2	50.7	80.0	85.7	100.0	65.2	55.3
Qwen3.5-35B-A3B									
Dense	88.5	85.7	84.4	72.9	100.0	99.0	100.0	65.9	57.0
Ours	88.7	84.5	84.4	69.9	98.3	98.3	98.4	66.9	60.0
Ours with initial block	88.6	84.5	83.9	71.2	97.3	98.3	99.2	66.5	58.0
Nemotron 3.5 Lightning									
Dense	–	–	91.1	–	100.0	99.0	100.0	95.7	61.0
Ours	–	–	70.0	–	84.0	49.0	71.5	92.4	53.0
Ours with initial block	–	–	74.7	–	81.0	42.0	99.0	96.5	55.0
Ours, $K = L/8$	–	–	83.5	–	90.0	81.0	95.5	95.1	56.0
Ours, $K = L/8$, with initial block	–	–	83.3	–	91.0	73.0	99.8	96.8	56.0

Table 12: MRCR-8 and LongBench-v2 of Qwen3.5-4B and Qwen3.5-35B-A3B without and with the initial block.

	MRCR-8			LongBench-v2		
Method	128K	256K	512K	$\leq$256K	>256K	all
Qwen3.5-4B						
Dense	0.797	0.574	0.430	54.3	59.2	55.0
Ours	0.788	0.561	0.408	53.9	55.7	54.2
Ours with initial block	0.783	0.561	0.369	55.5	61.2	56.3
Qwen3.5-35B-A3B						
Dense	0.846	0.655	0.525	64.4	65.2	64.5
Ours	0.844	0.668	0.459	63.1	61.7	62.9
Ours with initial block	0.885	0.655	0.495	64.7	64.7	64.7

The Qwen rows of Table 11 are means over three clustering seeds and the MRCR-8 and LongBench-v2 entries means over three runs, with $K = L/16$ throughout except the Nemotron rows at $K = L/8$. Common-word extraction beyond 128K is on the filled prompts (Appendix C.3).

E LATENCY IN DETAIL

E.1 LATENCY BY LENGTH

Table 13 gives the measurements of Figure 9, whose top and bottom rows are the upper row of Figure 1. The sparse prefill is single-pass through 512K and sliced at 1M. The last-chunk columns give the cost per token of the last 64K prompt tokens, which is what a chunked-prefill measurement at a given length reports. For the whole model, this is $[T(L) - T(L - 65{,}536)]/65{,}536$, where T is the prefill time from two separate runs, and for the attention layers it is the last chunk in one profiler trace. Every query in this chunk attends to almost the whole prompt, so the ratio between dense and our method is larger here than over the whole prefill. The nominal 1M latency prompt contains 1,047,552 tokens, and the measurement protocol is in Appendix C.

Appendix H draws the same grid for Qwen3.5-35B-A3B (Figure 15) and the prefill and decoding rows for Nemotron 3.5 Lightning (Figure 16).

E.2 PREFILL KERNEL BREAKDOWN

Each GPU kernel is attributed to its launching operator. Index construction normalizes keys, fits centroids, assigns prompt tokens, and builds posting lists. Scoring and selection use centroid matrix products and threshold refinement instead of a sort. Gathering collects, for each block, the queries that selected it, and merging combines the partial outputs of FlashAttention-3. The dense-prefix rule and exact current chunk add attention beyond the token budget. At 512K, the rule processes 17 of 64 query chunks densely because the boundary is inclusive. Thus $K/L = 1/16$ does not imply a sixteen-fold reduction in prefill arithmetic.

E.3 DENSE-PREFIX CUTOFF

The main configuration uses dense attention for a query chunk whose zero-based starting position satisfies $p \leq 4K$, four times the token budget. We vary this cutoff to $p \leq xK$ for $x \in \{4, 2, 1\}$ while keeping the token budget and other settings fixed. Because the boundary is inclusive and selection is decided per chunk, xK is the cutoff for chunk starts rather than the exact number of densely processed queries.

Reducing the cutoff substantially lowers the dense-prefix attention time (Table 15). However, the affected queries now require block selection and grouped attention, increasing selected-block computation, query gathering, and output merging. At $x = 2$, these added costs largely offset the savings, leaving total prefill time nearly unchanged. At $x = 1$, total time increases. Both paths use FlashAttention-3, but the sparse path additionally gathers queries into block-specific groups and merges their partial outputs. The sweep therefore measures the trade-off between these execution paths, not just the amount of dense attention. Neither smaller cutoff consistently improves the two retrieval scores, so we retain $x = 4$.

The dense reference is from Table 14. The quality comparisons use one clustering seed on the same examples, whereas the main RULER-5 results average three seeds.

E.4 ABLATION OF THE KERNEL DESIGN

Table 16 times three implementation choices of Section 3.4 against simpler alternatives on the 512K sliced prefill of Qwen3.5-4B at $K = 32{,}768$, on one GPU, with the selection and the rest of the runtime unchanged, so every row computes the same attention. Q-outer execution, in which each tile of 32 queries reads the union of its queries' blocks and masks every row to its own selection, takes more than twice as long as KV-outer execution and longer than dense attention, because a block is read again for every tile that selected it and the masked rows are still computed. This is the execution a block-sparse kernel gives a per-query selection, and it is what KV-outer execution replaces. Gathering each chunk's selected blocks out of the prompt-order KV, instead of reading the cluster-ordered copy built once per prompt, lengthens the prefill by about 1.7 seconds, or 9%, because a block's visible prefix grows with every chunk and the same tokens are gathered again. Replacing the fused selection kernel by a matrix product and a sort costs well under a second.

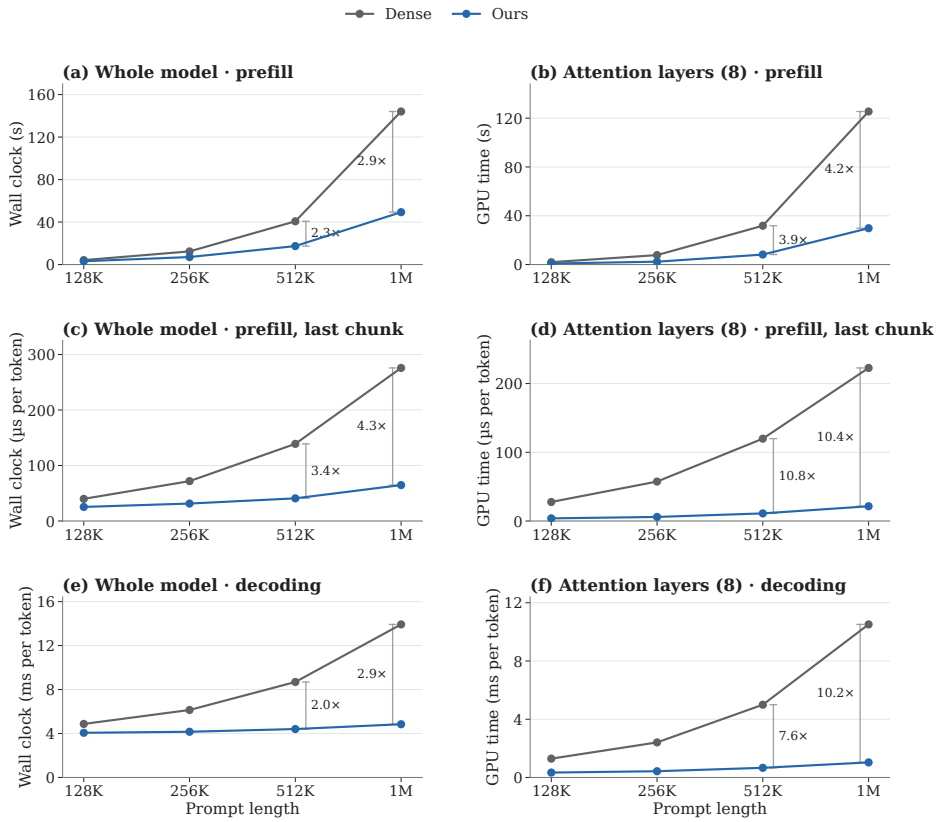

Figure 9: Qwen3.5-4B, detailed latency with the main configuration on one H200. The left column is the whole model by wall clock and the right column the eight full-attention layers by profiler accounting. The rows are prefill, last-chunk prefill per token as defined below, and decoding.

Table 13: Qwen3.5-4B latency on one H200. Whole-model columns are wall-clock measurements, the attention-layer columns profiler totals over the eight full-attention layers. The last rows are the dense reference model, Qwen3.5-2B.

	Prefill (s)				Prefill, last chunk (μs/token)				Decode (ms/token)			
	Whole model		Attention layers (8)		Whole model		Attention layers (8)		Whole model		Attention layers (8)	
Prompt	Dense	Ours	Dense	Ours	Dense	Ours	Dense	Ours	Dense	Ours	Dense	Ours
128K	4.18	3.11	1.92	0.84	39.9	25.4	27.7	3.84	4.87	4.07	1.29	0.34
256K	12.37	7.06	7.75	2.34	71.8	31.4	57.4	5.91	6.14	4.16	2.41	0.43
512K	40.69	17.39	31.81	8.18	139.0	40.8	119.8	11.07	8.69	4.41	5.00	0.66
1M	144.08	49.30	125.47	29.79	275.8	64.7	222.7	21.45	13.93	4.84	10.51	1.03
Reference, Qwen3.5-2B with dense attention, whole model												
128K	1.72	–	–	–	–	–	–	–	2.89	–	–	–
256K	4.99	–	–	–	–	–	–	–	3.48	–	–	–
512K	16.01	–	–	–	–	–	–	–	4.73	–	–	–
1M	56.16	–	–	–	–	–	–	–	7.48	–	–	–

The same comparison on one real chunk of that prefill gives the kernel-level picture. FlashAttention-3 with KV-outer execution takes about 21 milliseconds there, and FlexAttention (Dong et al., 2024) with a block mask carrying the identical per-query selection, the off-the-shelf way to run the same sparse attention, takes about 245. At this block size the per-query selection marks most 128 by 128 blocks as partially selected, so a block-sparse kernel reads nearly all of them, and KV-outer execution, in which a block is read once for all the queries that selected it, is what avoids that.

Table 14: GPU time in seconds for a 512K prefill on Qwen3.5-4B. The attention rows contain only FlashAttention-3 kernels, with scoring, selection, gathering, and merging reported separately. Totals come from profiling rather than the wall-clock medians in Table 13.

	Dense	Ours
Attention (8 attention layers)	31.4	6.4
selected blocks		4.02
dense prefix		2.02
current chunk		0.34
Scoring, selection, index build (ours)	0.0	0.3
score the centroids and select		0.24
k-means fit		0.02
index build		0.02
Gather, merge and other overhead (ours)	0.0	1.5
gather and index operations		0.72
merge of partial outputs		0.38
masks, casts and copies		0.38
Linear projections (all 32 layers)	6.4	6.4
Linear-attention layers (24 GDN)	1.6	1.8
Norms, activations, RoPE	0.7	0.7
Other	0.1	0.0
Total	40.3	17.1

Table 15: Effect of the dense-prefix cutoff on Qwen3.5-4B at 512K and $K = 32{,}768$. Timings are GPU seconds for one prefill, and accuracy is measured on 100 examples per task.

	GPU seconds of one 512K prefill							Accuracy (%)	
Setting	Selected blocks	Dense prefix	Own chunk	Attention	Scoring	Gather, merge	Total	MK-3	MK-2
Dense attention	—	—	—	31.44	—	—	40.30	81.0	97.0
$x = 4$ (ours)	4.02	2.03	0.34	6.38	0.24	1.48	17.07	75.0	85.0
$x = 2$	5.02	0.53	0.40	5.95	0.27	1.81	17.03	74.0	86.0
$x = 1$	5.65	0.16	0.43	6.24	0.29	2.10	17.64	73.0	84.0

E.5 Indexing older generated tokens

The main method attends exactly to every generated token, so the decode step grows with the generation length. An optional policy keeps the newest W generated tokens exact and adds the oldest F to the index whenever the generated suffix reaches $W + F$ tokens, assigning them to the existing centroids without refitting. The block-ordered cache is updated between decode steps without moving its buffers, so the captured decode graph stays valid. Each point in Figure 10 averages the decode steps since the previous point, and all times include the index updates.

With this policy the decode time stays nearly flat as the generation grows (Figure 10 and Table 17). On LongBench-v2 (Table 18), flushing changes accuracy by about as much as a repeated run of the same configuration does. We did not test reasoning traces as long as 64K tokens.

F Design choices on Qwen3.5-4B

F.1 The routing key

We compare routing representations while keeping attention computation unchanged. The main configuration uses unit-normalized pre-RoPE keys and queries, with spherical k-means. We test using only the 192 non-rotary dimensions, since the model rotates 64 of its 256 key dimensions. We also test raw keys with Euclidean k-means, using unnormalized cluster means and queries at the attention scale $\tau = \sqrt{d}$.

Table 16: Ablation of the kernel design on Qwen3.5-4B at 512K and $K = 32{,}768$, with the prefill and decode times of each execution.

Execution	Prefill (s)	Decode (ms/token)
Dense attention	40.69	9.27
Ours: KV-outer over one cluster-ordered copy	18.51	4.49
Selected blocks gathered from prompt order every chunk	20.24	4.45
Same selection, Q-outer instead of KV-outer	47.56	4.29
Selection by a GEMM and a sort	19.09	4.56

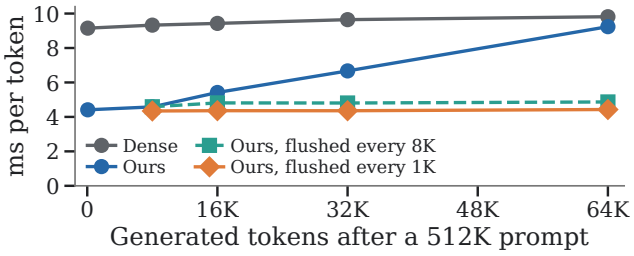

Figure 10: Decode latency of Qwen3.5-4B after a 512K prompt, index updates included.

Both variants stay within four points of our multi-key accuracy at 512K, at or below it on MK-3 and above it on MK-2, differences of the size by which separate runs of the same configuration differ (Table 19 and Appendix C.4). The non-rotary variant also gives similar MK-3 accuracy at 1M, with a small decline on MK-2. Changing the temperature does not improve the raw-key variant. Post-RoPE routing gives similar multi-key accuracy to pre-RoPE routing over three seeds. PCA projections fitted to raw-corpus keys keep similar multi-key accuracy at 128 and even 32 dimensions. Timed with random projections of the same size, reducing the dimension from 256 to 32 shortens the 512K prefill by about 5%, from 18.0 to 17.1 seconds, and leaves decoding unchanged. These results support using the original keys without a specialized routing representation on Qwen3.5. The non-rotary variant matches our MK-3 accuracy over three seeds, its MK-2 accuracy from an earlier run is a few points higher, and a closer study of which key dimensions to route on is left to future work. The choice is more consequential on Nemotron, where key centering prevents the severe cluster imbalance found without it (Table 32).

F.2 Token budget and block size at 512K

Table 20 lists the values plotted in Figure 6. It crosses four mean block sizes u with four token budgets K and scores each point on MK-2 over the same 100 items at 512K. The main setting's accuracy is the three-seed mean of the main RULER table, and the other points are single runs of one code version. Each prefill time is the mean over two prompts. We use MK-2 rather than MK-3 here because its gap to dense attention is larger, so it separates the settings more clearly.

At small budgets, finer blocks are much more accurate, but the gap closes as the budget grows, while finer blocks stay slower at every budget. At the main budget, halving the block size to 1,024 adds 2.6 seconds of prefill and quartering it adds 7.7 seconds, for accuracy gains of at most three points, whereas doubling it to 4,096 saves about one second and loses about three points. The coarse partition with the main budget is both faster and more accurate than the fine partition with half that budget. Because whole clusters are taken until the budget is reached, the clusters a query selects hold slightly more tokens than K, besides those of $\mathcal{D}_t$, on average 0.4 to 14 percent more across the grid and 2.4 percent at the main setting, so this comparison holds for the tokens actually selected as well. These results support choosing budget and block size together, rather than minimizing either independently.

Figure 11 draws the grid on MK-3 and MK-2 for the three models, with the values for the two Qwen models in Table 21 and for Nemotron in Table 34. Colour marks the cluster size and marker shape the budget. Points that were not measured are left out, namely $u = 1{,}024$ and 512 at $K = 65{,}536$ on MK-3 for the 4B model, and for the 35B-A3B model $K = 65{,}536$ and the finer points at $K = 32{,}768$, where $u = 2{,}048$ already comes close to dense attention. The main setting's accuracy on each model

Table 17: Qwen3.5-4B, mean decode time over the entire generation after a nominal 1M prompt, in milliseconds per token. The columns average over 64 and 65,536 generated tokens, respectively, rather than measuring only the last step.

	64 generated	65,536 generated
Dense	15.39	16.06
Ours	4.81	7.85
Ours, flushed every 8K ($W = 2{,}048$)	4.83	5.09
Ours, flushed every 1K ($W = 512$)	4.82	4.82

Table 18: Qwen3.5-4B LongBench-v2 accuracy (%) without and with flushing ($F = 1{,}024$, $W = 512$), on the same items and sampling seed. The sparse columns are separate runs from Table 1's.

	Dense	Ours	Ours, flushed
$\leq$256K (403 items)	53.1	54.8	53.8
$>$256K (67 items)	52.2	64.2	58.2
All (470)	53.0	56.2	54.5

is the three-seed mean of the main RULER table, and the other points are single runs. On the 4B model, the MK-3 grid shows the same pattern as MK-2, with fine clusters ahead at small budgets and coarse clusters catching up as the budget grows. The 35B-A3B model follows the same pattern on both tasks. When timing its $u = 512$ points, the runtime splits the gathering for the KV-outer attention into two pieces so that they fit in device memory.

Figure 12 repeats the profile of Figure 7 on Qwen3.5-35B-A3B and shows the same pattern. The densely processed prefix scales with the budget in every setting, and at a fixed budget, shortening it leaves prefill time and accuracy about unchanged (Appendix E.3), so the shorter dense prefix of the smaller budgets does not make those settings faster.

F.3 Budget scaling across context lengths

Increasing the budget beyond $L/16$ gives little additional MK-3 quality in these runs while increasing latency (Table 22). Prefill times vary slightly from run to run, so they differ slightly from those in Table 13. Reducing the budget lowers latency at the cost of retrieval accuracy. Neither a fixed token count nor a fixed fraction predicts accuracy across lengths, because the dense model's own retrieval performance also declines.

Figure 13 extends the comparison at 512K to MRCR-8 and LongBench-v2 on both Qwen models, over budgets from $L/64$ to $L/8$. Because prompt lengths differ across items, the horizontal axis is the prefill time of the RULER timing prompt at the matching length and budget, used as a uniform axis, namely 512K for the 512K bin of MRCR-8 and 256K for LongBench-v2, the upper bound of the band that holds most of its items. Filled markers are means over three runs, and hollow markers are single runs. For the 35B-A3B model, LongBench-v2 at $L/16$ is not drawn because $L/32$ already matches dense attention. MK-3 and MRCR-8 lose accuracy below $L/16$, while doubling the budget to $L/8$ does not raise MRCR-8 beyond the spread of the three $L/16$ runs. LongBench-v2 holds its accuracy down to $L/32$ and drops at $L/64$.

F.4 Positional baselines

We form contiguous blocks of B tokens and represent each block by its mean pre-RoPE key. Queries score these means without vector normalization at temperature $\sqrt{d}$, and head probabilities are summed within each GQA group before selecting the top $\lceil K/B \rceil$ blocks. These descriptor and scoring choices differ from the normalized centroid scores of our semantic blocks. Semantic blocks lose only a few points under the positional baseline's scoring, with raw keys, unnormalized cluster means, and temperature $\sqrt{d}$ (Table 19), and remain above positional blocks, so normalization and temperature do not explain the gap between the two. Both methods use the same token budget and dense-prefix, current-chunk, and generated-suffix policies. With per-query selection, positional blocks use the same

Table 19: The routing key on Qwen3.5-4B, with RULER multi-key and multi-value accuracy in percent at 512K, $K = L/16$, $u = 2{,}048$, 100 items per entry. Our row, the post-RoPE row, and the multi-key entries of the non-rotary row are means over three clustering seeds, and all other entries use seed 0. A dash marks an entry that was not measured.

Routing key	MK-3	MK-2	MV
Dense	81.0	97.0	99.8
Whole key, normalized, $\tau = 0.1$ (ours)	73.7	82.3	99.5
Whole key after RoPE, normalized, $\tau = 0.1$	77.0	80.0	99.8
Non-rotary 192 dimensions, normalized, $\tau = 0.1$	73.7	85.3	99.8
Raw key, Euclidean k-means, $\tau = 16$	70.0	86.0	99.3
Raw key, Euclidean k-means, $\tau = 4$	67.0	83.0	99.3
PCA to 128 dimensions, normalized, $\tau = 0.1$	75.0	86.0	99.8
PCA to 32 dimensions, normalized, $\tau = 0.1$	73.0	84.0	99.8

Table 20: The values behind Figure 6, Qwen3.5-4B at 512K.

	MK-2 (%) at K				Prefill (s) at K			
u	8K	16K	32K	64K	8K	16K	32K	64K
Dense	97.0				40.69			
4,096	31.0	55.0	79.0	92.0	12.66	14.05	16.78	23.47
2,048	40.0	66.0	82.3	91.0	13.40	14.98	17.86	24.30
1,024	50.0	73.0	83.0	94.0	15.30	17.31	20.42	26.14
512	63.0	74.0	85.0	92.0	18.97	22.21	25.59	30.05

KV-outer execution as semantic blocks (Section 3.4). For shared selection, g consecutive queries score the blocks together with the tile score of FlashPrefill (Fan et al., 2026a). Within each query head, every query of the group scores every block with the configuration's routing normalization and scoring rule. The group's score for a block is the sum of the exponentiated scores over its queries, normalized over blocks. These head scores are summed within each GQA group, as for per-query selection, and the top blocks fill the budget. All g queries use the resulting block selection, while attention uses their original query vectors and causal masks. Groups stay within a prefill chunk, and an incomplete final group is padded by repeating its last query. The same procedure is used for semantic and positional blocks. Only prefill selection changes, since decoding still selects blocks for each new token. The group attends to its selected blocks in one FlashAttention call, as in structured sparse prefill (Jiang et al., 2024; Lai et al., 2025). A Triton kernel computes the group scores without storing the score of each query.

Every query of the group contributes to its score, rather than a representative subset. FlashPrefill keeps the blocks whose score is within a fraction of the maximum, and its second version shares one selection among 32 query positions, within the range of g measured here (Fan et al., 2026a). FlexPrefill's query-aware pattern pools queries and selects blocks using an attention-mass threshold (Lai et al., 2025). QuoKA instead keeps a small set of representative queries, the ones least aligned with the mean query, and selects the keys most aligned with them (Jones et al., 2026). Our ablation uses a fixed token budget to compare semantic and positional blocks with the same group score. These controlled ablations vary the block layout and the sharing span at a fixed budget and do not reproduce the complete selection policies of MInference, FlexPrefill, QuoKA, or FlashPrefill. Several of them would also not run unchanged on these models, since the released MInference and FlexPrefill kernels available to us accept head dimensions of at most 128, whereas Qwen3.5 uses 256, and we found no public QuoKA code at the time of submission. We also measured three other group scores at most of these settings, namely the group's averaged query as in MInference and FlexPrefill (Jiang et al., 2024; Lai et al., 2025), the maximum over the group's queries, and a sum that also runs over the heads of a GQA group as in FlashPrefill V2. Each gave accuracy similar to or lower than the summed exponentials.

On Qwen3.5-35B-A3B (Table 24), MK-3 is near its ceiling, and positional blocks of 8 tokens shared over 32 queries reach our accuracy, but their prefill takes $1.8\times$ as long, as on the 4B model. Every

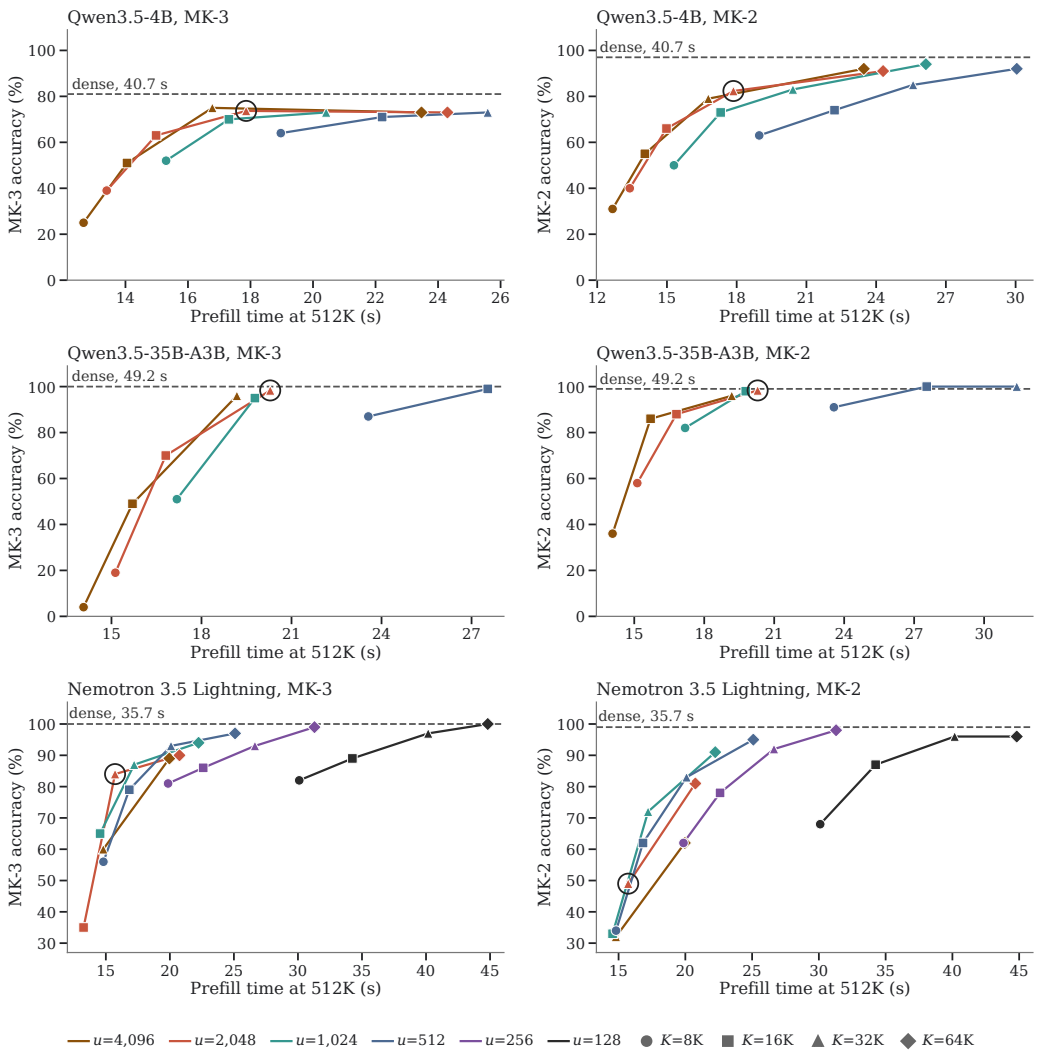

Figure 11: Cluster size and token budget on MK-3 and MK-2 at 512K for Qwen3.5-4B, Qwen3.5-35B-A3B, and Nemotron 3.5 Lightning.

faster positional setting is at least 8 points less accurate. On MK-2, the same setting is 16 points below our three-seed mean of 98.3%, and every other positional setting is lower still.

The second MK-3 column adds the fixed initial block of Appendix D.3 to every row, which changes the prefill times by at most 5%.

A Quest-inspired min/max descriptor does not improve the positional baseline in this pre-RoPE setting (Tang et al., 2024). This comparison tests the descriptor, not Quest's complete decoding method. Its shared rows and the temperature sweeps below score the group's averaged query rather than the summed exponentials.

The min/max descriptor bounds the dot product with each block's pre-RoPE keys. It is not an upper bound on the model's post-RoPE attention logits. At $B = 32$ with shared selection, increasing its scoring temperature from 16 to 64, 256, or 1,024 gives MK-3 scores of 16, 19, and 17 percent, respectively. The mean-key descriptor's temperature, swept at the same point over 4, 16, 64, 256 and 1,024, gives 36, 40, 36, 35 and 34 percent, so the paper's $\sqrt{d} = 16$ is the best of the five, and none moves the baseline towards semantic blocks.

Table 21: The values behind Figure 11, Qwen3.5-4B and Qwen3.5-35B-A3B at 512K. A dash marks a point that was not measured.

	MK-3 (%) at K				MK-2 (%) at K				Prefill (s) at K			
u	8K	16K	32K	64K	8K	16K	32K	64K	8K	16K	32K	64K
Qwen3.5-4B												
Dense		81.0				97.0				40.69		
4,096	25.0	51.0	75.0	73.0	31.0	55.0	79.0	92.0	12.66	14.05	16.78	23.47
2,048	39.0	63.0	73.7	73.0	40.0	66.0	82.3	91.0	13.40	14.98	17.86	24.30
1,024	52.0	70.0	73.0	–	50.0	73.0	83.0	94.0	15.30	17.31	20.42	26.14
512	64.0	71.0	73.0	–	63.0	74.0	85.0	92.0	18.97	22.21	25.59	30.05
Qwen3.5-35B-A3B												
Dense		100.0				99.0				49.19		
4,096	4.0	49.0	96.0	–	36.0	86.0	96.0	–	14.07	15.70	19.18	–
2,048	19.0	70.0	98.3	–	58.0	88.0	98.3	–	15.13	16.81	20.29	–
1,024	51.0	95.0	–	–	82.0	98.0	–	–	17.18	19.78	–	–
512	87.0	99.0	–	–	91.0	100.0	100.0	–	23.56	27.54	31.40	–

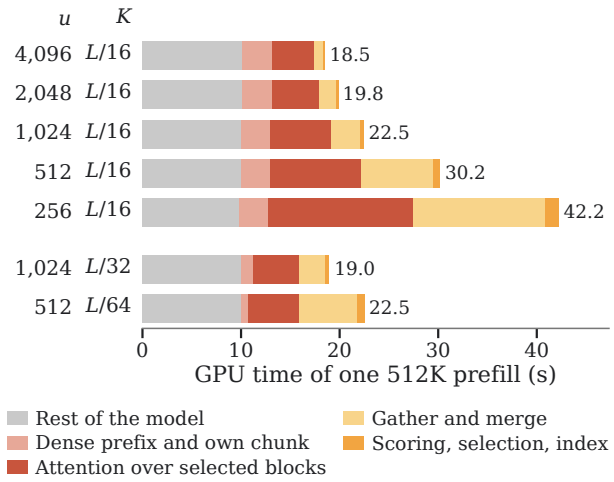

Figure 12: GPU time of one 512K prefill on Qwen3.5-35B-A3B by cluster size u and token budget K, drawn as Figure 7.

F.5 Centroids fitted on a corpus

Table 2 compares centroids fitted to each prompt with centroids fitted once and reused, with 256 centroids per layer and KV head. All rows keep the original query and key representations and the same selection rule, so this comparison does not include SAAP's learned query classifier (Mazaré et al., 2025). Per-prompt results average three clustering seeds, and the other rows are single runs. Raw-corpus centroids severely reduce multi-key accuracy and assign over half of the common-word prompt to one cluster, which the runtime rejects, so the table marks that task as failed. Fitting once to twenty RULER-5 examples per task recovers much of the loss. This fitting set includes evaluation examples, so the row diagnoses corpus mismatch rather than held-out generalization. The first-64K row fits each prompt's centroids on its first 64K tokens and assigns every later key to them, as the prefix index of Appendix B.5 does.

Table 22: Qwen3.5-4B, MK-3 accuracy in percent ($n = 100$) and prefill speedup at each length and budget. The $K = L/16$ column is the three-seed mean of the main table, and the other entries are single runs.

Prompt	Dense	$K = L/8$	$K = L/16$	$K = L/32$	$K = L/64$
256K	100 (12.4 s)	97 (1.36×)	97.7 (1.74×)	88 (1.86×)	42 (1.98×)
512K	81 (40.7 s)	73 (1.62×)	73.7 (2.25×)	65 (2.68×)	41 (2.98×)
1M	26 (144.1 s)	22 (1.83×)	20.0 (2.84×)	14 (3.68×)	10 (4.33×)

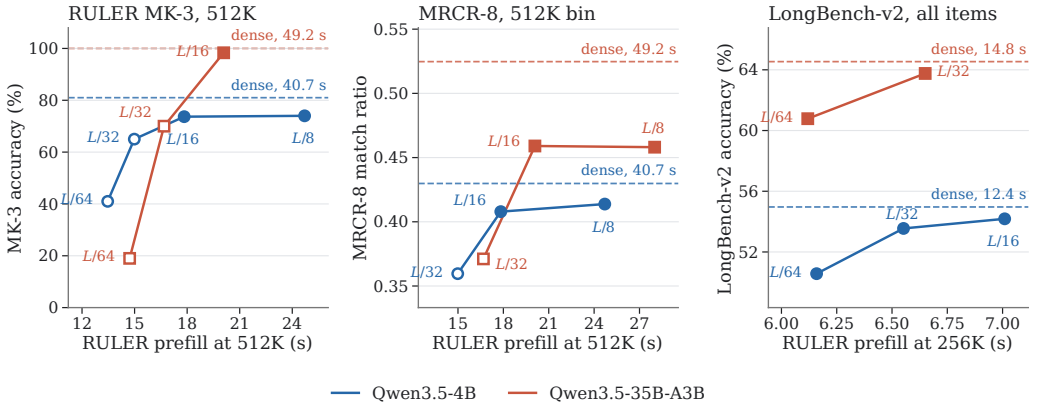

Figure 13: Quality against prefill time over token budgets from $L/64$ to $L/8$ on Qwen3.5-4B and Qwen3.5-35B-A3B.

Table 23: Positional and semantic retrieval on Qwen3.5-4B at 512K, $K = L/16$, MK-3 accuracy in percent with $n = 100$, without and with the fixed initial block of Appendix D.3, and MK-2 without it. B is positional block size and g query sharing span. Min/max rows describe each positional block by its minimum and maximum keys, and their shared rows use the averaged query. Our per-token row is the mean over three clustering seeds. A dash denotes an unmeasured entry.

			MK-3 (%), initial block		MK-2 (%)	
Index	B	g	Without	With	Without	Prefill (s)
Positional, shared	2,048	64	7	4	–	15.6
	128	64	19	21	–	16.2
	32	64	43	51	18	18.4
	16	64	54	65	24	21.5
	8	64	56	71	51	27.5
	4	64	54	69	–	39.5
	32	32	57	64	34	19.1
	16	32	62	71	37	24.0
	8	32	64	74	51	32.9
	4	32	59	75	–	50.1
	32	16	61	66	43	23.8
Positional, per token	2,048	1	10	10	–	16.1
	128	1	28	35	30	33.1
	32	1	57	59	51	84.9
Min/max, shared	2,048	64	6	6	–	–
	32	64	19	20	–	18.7
	16	64	23	24	–	–
Min/max, per token	32	1	25	26	–	87.7
Ours, per token	–	1	73.7	80.0	82.3	18.3
Ours, shared	–	64	25	17	13	16.4
Dense	–	–	81	–	97	40.7

Table 24: Positional and semantic retrieval on Qwen3.5-35B-A3B at 512K, $K = L/16$, without the initial block, MK-3 and MK-2 accuracy in percent with $n = 100$ (ours on clustering seed 0) and 512K prefill time.

Index	B	g	MK-3 (%)	MK-2 (%)	Prefill (s)
Positional, shared	16	64	75	45	30.8
	32	32	77	51	22.5
	16	32	91	67	27.4
	8	32	99	82	37.1
Positional, per token	128	1	50	50	34.4
Ours, per token	–	1	99	98	20.4
Dense	–	–	100	99	49.2

G SOURCES OF QUALITY LOSS AND TRAINING BASELINES

G.1 ATTENTION MASS MISSED BY CLUSTER SCORING

Our selection can lose attention mass in two ways. It can pick the wrong clusters, and even well-chosen clusters include tokens with little attention weight, so the budget covers fewer of the high-weight tokens. We separate the two by replacing the centroid scores with dense attention while keeping the partition fixed.

We build the partition from the dense model's keys and compare three selections for the dense model's queries. The first applies our selection rule to the fitted clusters. The second sums the dense attention probabilities within each cluster and greedily selects whole clusters under the same budget, ranking by total mass or by mass per token and keeping the better of the two for each query and GQA group. The third selects the K tokens that receive the most dense attention, without reading whole clusters. We call the last two *dense-ranked clusters* and *dense-ranked tokens*. Both need dense attention at inference time, so they serve only as references.

Coverage is the percentage of the dense model's attention on the prompt that falls on the selected tokens. For example, 80% coverage means that the selected tokens receive 80% of that attention, not that 80% of answers or attention outputs agree. Each query's current chunk is always read exactly and does not count toward the budget, and for generated queries coverage counts only attention to the prompt.

The analysis uses ten prompts per task at 512K, with the last 64 prompt positions and up to 32 generated tokens as queries, over the 4B model's eight full-attention layers and four KV heads. The MK-2 prompts are chosen to include both correct and incorrect sparse outputs. The ten MK-3 prompts come from a separate set of 400 generated prompts rather than from the MK-3 items of our tables, and MRCR uses the first ten prompts of its 512K bin. The budgets are $L/16$ and $L/8$ of each prompt's actual length. Dense probabilities are summed over a GQA group to rank tokens or clusters, and the retained mass is normalized by each query head's attention on the prompt and averaged over heads, queries, layers, and prompts. Because heads can put different total attention on the prompt during generation, the ranking and the reported coverage are not exactly the same quantity. These queries emphasize retrieval near the end of the prompt and during generation, not the average prefill query.

At $K = L/16$, selecting individual tokens raises coverage more than scoring clusters by dense attention does, on all three tasks (Figure 14). Cluster selections can exceed the budget by part of a cluster, but giving the token reference the same number of tokens changes its coverage by at most 0.3 points. Better cluster scores thus recover only part of the mass lost by reading whole clusters.

We also replace the centroid scores inside the sparse runtime and measure RULER accuracy (Table 25). At every full-attention layer, we compute dense attention probabilities from the current queries and prompt keys in FP32, sum them within each cluster, and select clusters in that order under the runtime's budget rule. The clustering, the regions read without selection, the attention computation, and the handling of generated tokens are unchanged. Queries and keys come from this sparse run, so the comparison includes the effect of earlier sparse layers on the model states. We use the first 50 prompts of each multi-key task at 512K, with clustering seed 0 for the dense-attention scores and the mean and standard deviation over three seeds for the centroid scores.

On MK-2, scoring clusters by dense attention recovers six of the twelve points lost relative to dense attention on these prompts. On MK-3, the difference is within the variation across clustering seeds, and on the same seed the two selections disagree on only five of the 50 prompts. Changing the scores can therefore recover a substantial part of the accuracy loss, even though it raises average coverage only a little.

Coverage has two limits as a measure. It ignores the value vectors, so a small loss of attention mass can change the output substantially when the missed values carry the answer, and barely when they are redundant. It also holds the dense model's states fixed and so misses errors that accumulate through sparse layers, which the runtime experiment above includes. Accordingly, mean coverage does not separate correct from incorrect sparse outputs in these samples. On MK-2 it ranges from 75.8% to 76.9% on five correct prompts and from 75.8% to 77.9% on five incorrect ones.

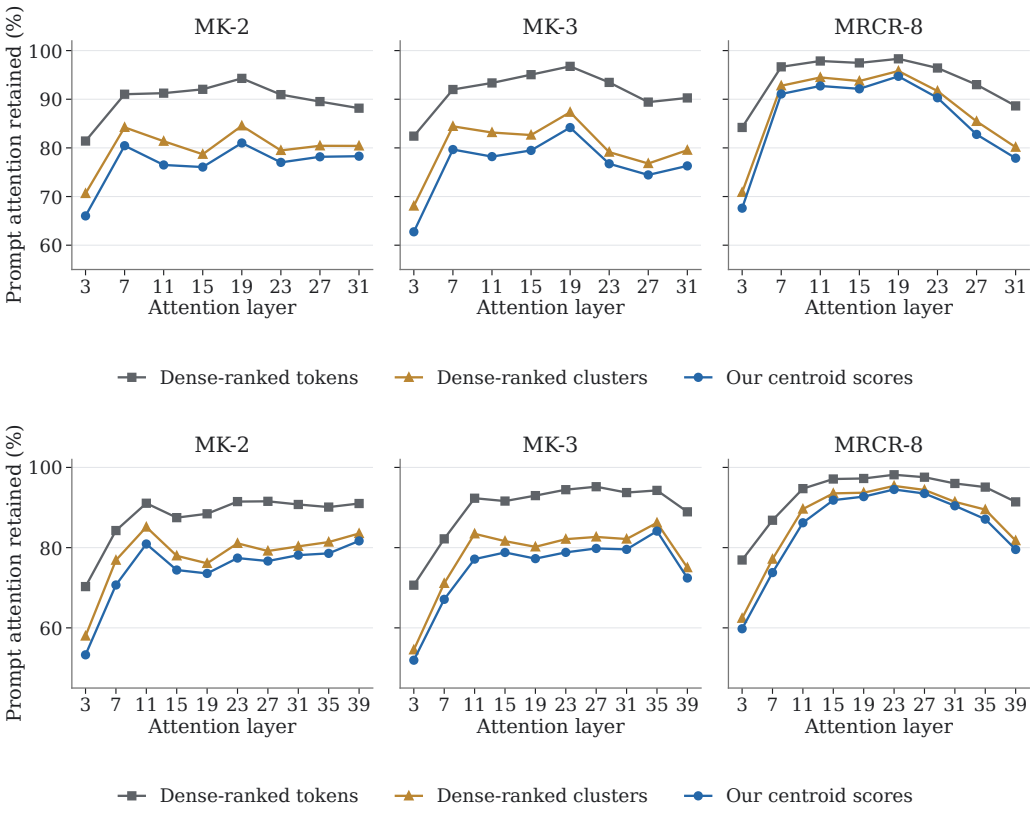

Figure 14: Prompt-attention coverage by full-attention layer at $K = L/16$ under our centroid scores, dense-ranked clusters, and dense-ranked tokens, on ten 512K examples per task, for Qwen3.5-4B (top, eight layers) and Qwen3.5-35B-A3B (bottom, ten). All curves use dense-model states.

Table 25: Cluster scoring on Qwen3.5-4B at 512K, $u = 2{,}048$, $K = L/16$. Accuracy (%) on 50 examples per task.

Selection	MK-3	MK-2
Dense attention	80.0	100.0
Centroid scores	74.7 (5.0)	88.0 (2.0)
Dense attention mass	78.0	94.0

G.2 Learning the routing map

Appendix G.1 shows that better cluster scores would recover part of the gap to dense attention. Although training is out of our scope, this subsection and the next test whether two simple training baselines close this gap. The first learns linear maps for selection while leaving the model weights and attention kernels unchanged. For each full-attention layer, we train a query map $W_q \in \mathbb{R}^{d\times d}$ per query head and a key map $W_k \in \mathbb{R}^{d\times d}$ per KV head, initialized to the identity. We also train the temperature τ and size penalty β. The penalty is clamped to nonnegative values in the forward pass with a straight-through gradient, matching its use at inference time.

For each training document, we fit clusters on the mapped keys and sum the dense model's attention probabilities within each cluster. The loss is the KL divergence from this cluster-mass distribution to the normalized selection scores in Equation 2, with an L2 penalty keeping the maps near the identity. Cluster assignments are recomputed without gradients. The normalized means of the assigned mapped keys carry gradients to W_k.

Table 26: Learning query and key maps for clustering and selection on Qwen3.5-4B. The left columns report held-out KL loss, retained attention mass, and selection parameters. The right columns report long-context task performance.

	Objective		Rule		512K		RULER-5		MRCR-8
Configuration	KL	mass at $L/16$	τ	β	MK-3	MK-2	256K	512K	512K
Dense attention	–	–	–	–	81.0	97.0	84.7	78.4	0.447
Identity map (ours)	1.07	0.674	0.10	0.25	73.7 (3.1)	84.7 (1.5)	82.7 (0.5)	75.4 (0.5)	0.411
Learned W_q, W_k, τ, β	0.31	0.695	0.036	0	69.7 (0.6)	87.0 (2.6)	83.1 (0.2)	75.1 (0.5)	0.383

Training uses 25,684 raw-text documents of 65,536 tokens each, about 1.7B tokens, drawn from the PDF documents of the Dolma 3 Longmino mix (OLMo Team, 2025). We leave out the mix's synthetic common-word extraction and rewriting tasks, which resemble RULER tasks. We train for 3,000 steps with eight data-parallel workers, a total batch size of eight documents, and learning rate 10^{-3}. The objective is evaluated on eight held-out documents and largely plateaus after roughly half the run. For downstream evaluation, RULER scores are percentages averaged over three clustering seeds, with standard deviations in parentheses. MRCR-8 reports the mean match ratio over 100 examples.

Training reduces the held-out KL by about two thirds, but changes the attention mass retained by the selected clusters much less (Table 26). Much of the KL reduction accompanies a lower temperature, which sharpens the score distribution, while the size penalty reaches zero. The objective favors clusters receiving more total attention mass, which can reward large clusters and conflict with the runtime's size penalty. Allowing a negative penalty reaches a similar objective.

The learned map gives similar aggregate RULER-5 scores to the identity map and a lower MRCR-8 score. Its 512K MK-3 is four points below the identity map's over three seeds, close to the few points by which separate runs of the same configuration differ (Appendix C.4), and its other RULER changes are comparable to the observed variation across clustering seeds. Earlier runs used a corpus 8.5 times smaller and a quarter to a third of the training steps, and tested an unconstrained penalty, the same map with the penalty clamped at evaluation, and a query-only map. None gave a consistent retrieval improvement.

G.3 Adapting the model to its sparse attention with LoRA

The second baseline adapts the model weights to the sparse attention. A copy of the model with rank-8 LoRA adapters (Hu et al., 2022) runs sparse attention and is trained to match the dense model's next-token distribution by forward KL over the full vocabulary. One configuration adapts the query, key, and output projections of the eight full-attention layers. The other also adapts the MLP and Gated DeltaNet projections. The index and the selection are recomputed from the adapted model's current keys and queries with the same code as at inference. Gradients pass through attention over the selected tokens, but not through the discrete selection.

Each run uses 300 steps with four raw-text documents per step, from the same documents. The documents are 64K tokens long, and the loss is computed at every position. As an implementation check, the model with zero-initialized adapters reproduces the dense model when both use dense attention. For evaluation, adapters are merged into the weights and the sparse runtime is unchanged. Each task uses 100 examples and one clustering seed.

The held-out distillation objective falls substantially during training, but the task results do not improve (Table 27). At 512K, MK-2 stays close to the training-free configuration, while MK-3 and MRCR-8 are lower. The MRCR decline comes with the adapted models entering thinking mode on fewer prompts and beginning the answer in the expected format less often. The frequency of outputs reaching the generation limit does not increase, so simple truncation does not explain this decline.

These runs use raw text and documents shorter than the evaluation prompts. Other work on adapting models to sparse attention also trains on synthetic long-context tasks (Zhou et al., 2026), which we do not test. Our results show that reducing raw-text distillation loss need not improve long-context retrieval, but do not establish a general limitation of sparse-aware training.

Table 27: Adapting Qwen3.5-4B to sparse attention with LoRA trained on 64K documents. RULER multi-key scores are accuracy (%), and MRCR-8 reports a mean match ratio. All configurations use the same sparse runtime and token budget.

	128K		512K		
Adapters	MK-2	MK-3	MK-2	MK-3	MRCR-8
Dense	100	100	97	81	0.447
Ours, no training	98	100	86	73	0.411
Attention	99	99	86	66	0.290
+MLP+GDN	100	99	84	70	0.264

H OTHER ARCHITECTURES

This section provides the full results for Qwen3.5-35B-A3B and Nemotron 3.5 Lightning. The two models show that the same execution design applies to models whose keys are distributed differently, while the clustering settings and token budget do not transfer equally well.

H.1 QWEN3.5-35B-A3B

Qwen3.5-35B-A3B has 40 layers, including ten full-attention layers with 16 query heads, two KV heads, and head dimension 256. It uses 256 experts with eight active and shares the 4B model's tokenizer. The sparse implementation uses the same query and key representations, clustering settings, and selection rule as the 4B model. All sparse quality measurements use sliced prefill. Latency measurements use it from 512K onward, where single-pass prefill exceeds device memory. The dense baseline uses chunked prefill.

Table 9 gives the task scores beside the 4B model's. Figure 15 draws its latency as Figure 9 does for the 4B model, with the values in Table 28 and the last-chunk cost, as defined in Appendix E.1, in Table 29.

RULER-5 remains close to dense attention through 512K, including on both multi-key tasks. At 1M, the larger losses are in multi-key retrieval and common-word extraction. Halving or quartering the budget at 512K substantially reduces multi-key accuracy. The QA and common-word results of those halved and quartered runs used capped prompts and are not shown. The cluster-size grid behaves as on the 4B model (Figure 11). At the main budget, reducing u from 2,048 to 512 gives a small MK-2 gain but substantially increases prefill latency.

The 512K prefill profile in Figure 18 shows that the unchanged expert and Gated DeltaNet layers account for much of the remaining sparse prefill time. Because the sparse prefill is sliced at this length, these layers run in 32K slices, and the mixture-of-experts kernels are the largest part of the time not attributed to attention.

For long generations, reading the entire generated suffix exactly increases decode cost. After a 512K prompt, the decode step grows from 4.9 ms per token at the start to 10.5 ms between 32K and 64K generated tokens, above dense attention's 9.1 ms, while flushing keeps it near 5.1 ms. Flushing avoids this growth after both 512K and 1M prompts, with similar aggregate LongBench-v2 accuracy in the measured runs (Tables 30 and 31).

Our method either attends to all generated tokens exactly or flushes the older ones into the index every F tokens, keeping the newest W outside it. Table 31 compares the main budget with half that budget on MRCR-8 and LongBench-v2. In these single runs, MRCR-8 at the main budget remains close to dense attention across the tested lengths, although the three-run means of Table 1 fall below it at 512K. Halving the budget reduces the score most at 512K. LongBench-v2 scores vary less across the sparse configurations, but the single-generation measurements do not establish an ordering between these budgets. Flushing changes aggregate LongBench-v2 accuracy by less than one percentage point.

The last row flushes the older generated tokens into the index every $F = 1{,}024$ tokens, keeping the newest $W = 512$ outside it (Appendix E.5).

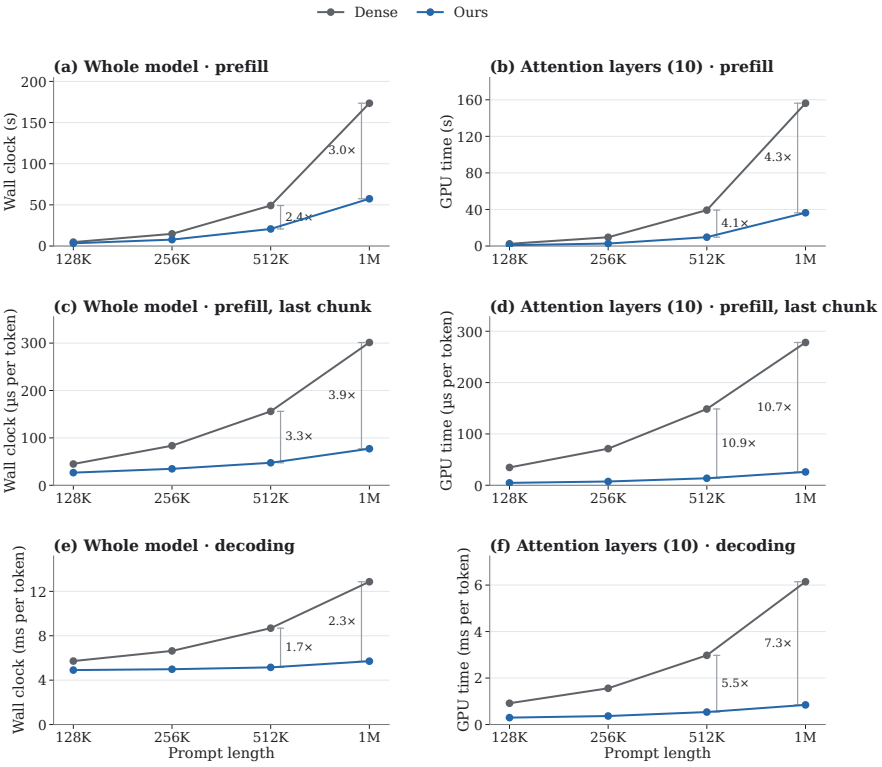

Figure 15: Qwen3.5-35B-A3B, latency against prompt length on one H200 at $K = L/16$, drawn as Figure 9. The left column is the whole model by wall clock and the right column the ten full-attention layers by profiler accounting. The rows are prefill, the last chunk of prefill per token as in Table 29, and decoding.

Table 28: Qwen3.5-35B-A3B latency on one H200 at $K = L/16$ and $u = 2{,}048$, as Table 13. Whole-model columns are wall clock, the attention-layer columns profiler totals over the ten full-attention layers. The last rows halve and quarter the budget at 512K.

		Whole model		Attention layers (10)	
Prompt	Configuration	Prefill (s)	Decode (ms)	Prefill (s)	Decode (ms)
128K	Dense	4.75	5.72	2.41	0.92
128K	Ours	3.31	4.91	0.96	0.30
256K	Dense	14.77	6.64	9.58	1.56
256K	Ours	7.76	4.98	2.69	0.37
512K	Dense	49.19	8.68	39.25	2.98
512K	Ours	20.73	5.15	9.68	0.54
1M	Dense	173.49	12.87	156.24	6.14
1M	Ours	57.42	5.71	36.38	0.84
512K	Ours, $K = L/32$	17.21	5.03	–	–
512K	Ours, $K = L/64$	15.21	4.96	–	–

The coverage diagnostic shows the same ordering as on the 4B model (Figure 14). Dense-ranked clusters retain more attention mass than centroid scoring, and selecting individual tokens retains more than either cluster-based method. Mean coverage is slightly lower than on the 4B model even though the task-accuracy loss is smaller. This comparison illustrates why coverage alone is insufficient to predict the effect on task accuracy.

Table 29: Qwen3.5-35B-A3B, the last-chunk cost of Table 13, the last 64K tokens of a prefill in microseconds per token. The whole model is the difference between prefills of L and $L - 64$K tokens, the sparse value the mean of two runs of the same prompt. The ten attention layers are the last chunk's attention and routing kernels in the profiler trace.

	Whole model			Attention layers (10)		
Prompt	Dense	Ours	Ratio	Dense	Ours	Ratio
128K	44.9	26.6	1.7	34.7	4.66	7.4
256K	83.5	34.7	2.4	71.3	7.35	9.7
512K	155.9	47.5	3.3	148.6	13.64	10.9
1M	301.3	77.1	3.9	278.1	25.99	10.7

Table 30: Qwen3.5-35B-A3B, mean decode time after a 1M prompt in milliseconds per token, over 64 and over 65,536 generated tokens, $K = L/16$, one H200.

Configuration	64	65,536
Dense	12.90	13.42
Ours	5.57	9.21
Ours, flushed every 1K ($W = 512$)	5.55	5.49
Ours, flushed every 8K ($W = 2{,}048$)	5.57	5.91

H.2 Nemotron 3.5 Lightning

Nemotron 3.5 Lightning 30B-A3B combines Mamba-2 and MoE layers with six full-attention layers, each with two KV heads and no rotary embedding. We replace these attention layers and retain the original model weights. Compared with Qwen3.5, this configuration centers keys, uses 32 clustering iterations with at least 128 sampled keys per centroid, and slices non-attention layers to avoid an offset overflow in the Mamba-2 convolution kernel. The clustering settings were selected on a 64K pilot and retained for the longer evaluations. In that pilot, whitening the keys instead of centering them, with or without k-means++ seeding, did worse (Table 32). Centering addresses a strong shared direction in the keys that otherwise leads to cluster collapse. It changes only routing representations, not the KV used for attention. We have not established whether the lack of RoPE causes this anisotropy.

Figure 16 draws the latency columns of Table 33. Both decode paths use full-step graphs. The nominal 1M latency prompt is 1,028,096 tokens, while the 1M task evaluations use the 1,048,576-token RULER prompts.

At $u = 2{,}048$ and $K = L/16$, prefill is about twice as fast as dense attention at 256K and approaches three times as fast at 1M. The MK-3 loss is substantial at every length, however. Decode gains are smaller because the replaced attention layers account for less of the original decode step.

Figure 11 (bottom row) repeats the grid of Figure 6 on MK-3 and MK-2 at 512K, extended to $u = 256$ and 128, with its values in Table 34. Points below 30% accuracy are not drawn, and the ringed point is $u = 2{,}048$ with $K = 32{,}768$. The $u = 256$ and 128 points ran the KV-outer gather in batches of 512 clusters to fit in device memory. At a fixed budget, accuracy rises as u shrinks down to $u = 512$. At equal prefill time, however, $u = 256$ and 128 do not pay off, because $u = 512$ or 1,024 with a larger budget is about as fast and more accurate. The best cluster size on this model is therefore around $u = 512$ to 1,024, finer than on the Qwen models. Figure 17 breaks down the prefill time by cluster size as Figure 7 does for the Qwen models. Smaller clusters again cost time mainly in attention over the selected blocks, gathering, and merging, but they add less time than on the Qwen models.

Table 9 extends the 512K evaluation to all five tasks, beside the Qwen models. Unlike the Qwen models, Nemotron loses substantially on multi-value retrieval as well as multi-key retrieval. Doubling the budget recovers much of this loss, while retaining prefill speedups of 1.5–2.0$\times$ from 256K to 1M (Table 33). Decode time increases less than prefill time, although the difference grows at 1M. These results support a larger budget or finer clusters for this model than for the Qwen models. They establish that the method can be applied to this architecture, with model-specific clustering and budget choices.

Table 31: Qwen3.5-35B-A3B on MRCR-8 (mean match ratio) and LongBench-v2 (accuracy over the same 470 items), thinking on, at $K = L/16$ and at half of it.

	MRCR-8			LongBench-v2		
	128K	256K	512K	$\leq$256K	>256K	all
Dense	0.849	0.644	0.497	65.3	62.7	64.9
Ours, $K = L/32$	0.835	0.596	0.371	64.3	64.2	64.3
Ours, $K = L/16$	0.839	0.682	0.491	62.3	67.2	63.0
Ours, $K = L/16$, flushed	–	–	–	63.0	64.2	63.2

Table 32: Nemotron 3.5 Lightning, the 64K pilot behind the clustering settings, with multi-key-3 accuracy in percent on 20 items, $K = L/16$. Centering subtracts the mean key before routing, whitening scales the centered keys by their covariance, and iterations and samples are the k-means fit.

Routing representation	MK-3
Dense	100
Uncentered keys	20
Centered, 32 iterations, 128 samples per centroid	100
Whitened	40
Whitened, k-means++ seeding, 32 iterations	35

In the Nemotron profile the non-attention layers use the same weights in both configurations but different slicing schedules, so their GPU times need not coincide.

Table 33: Nemotron 3.5 Lightning latency on one H200 and MK-3 accuracy in percent ($n = 100$) by prompt length, at $K = L/16$ unless noted. The attention-layer columns are profiler totals over the six full-attention layers.

			Whole model		Attention layers (6)	
Prompt	Configuration	MK-3 (%)	Prefill (s)	Decode (ms)	Prefill (s)	Decode (ms)
256K	Dense	100	11.07	3.92	6.91	0.51
256K	Ours, $u = 2{,}048$	73	6.15	3.39	1.94	0.17
256K	Ours, $u = 1{,}024$	93	6.82	3.38	2.34	0.18
256K	Ours, $u = 512$	98	7.68	3.42	3.07	0.22
256K	Ours, $u = 2{,}048$, $K = L/8$	96	7.59	3.44	3.15	0.22
512K	Dense	100	35.72	4.44	28.32	0.88
512K	Ours, $u = 2{,}048$	84	15.33	3.59	6.95	0.23
512K	Ours, $u = 1{,}024$	87	17.30	3.54	8.47	0.27
512K	Ours, $u = 512$	93	21.42	3.60	11.15	0.34
512K	Ours, $u = 2{,}048$, $K = L/8$	90	20.70	3.63	12.06	0.32
1M	Dense	100	120.18	5.37	104.12	1.64
1M	Ours, $u = 2{,}048$	70	41.09	3.65	24.54	0.35
1M	Ours, $u = 2{,}048$, $K = L/8$	85	61.62	3.87	44.74	0.51

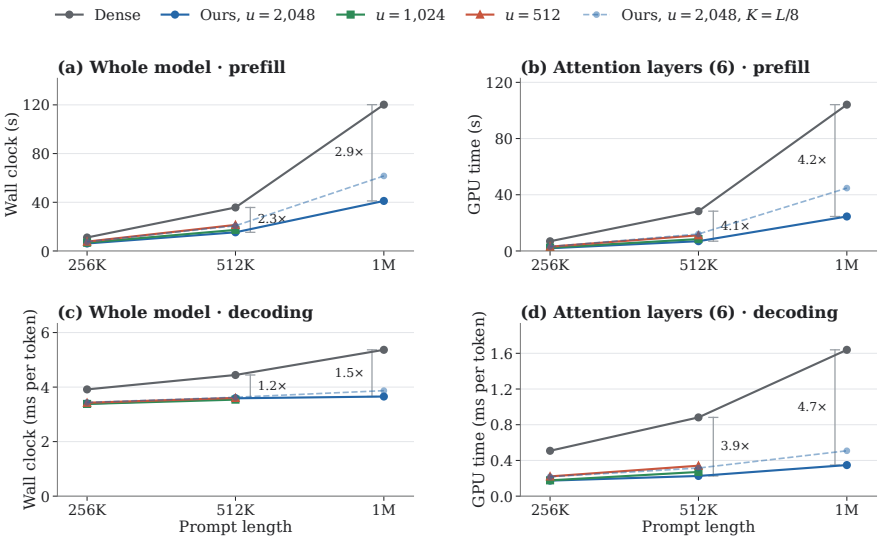

Figure 16: Nemotron 3.5 Lightning, latency against prompt length on one H200 at $K = L/16$ for three target mean block sizes u, with the doubled budget $K = L/8$ at $u = 2{,}048$ dashed. The left column is the whole model by wall clock and the right column the six full-attention layers by profiler accounting. The rows are prefill and decoding. The brackets are dense over ours at $u = 2{,}048$ and $K = L/16$, under the conditions of Table 33.

Table 34: The values behind the bottom row of Figure 11, Nemotron 3.5 Lightning at 512K.

	MK-3 (%) at K				MK-2 (%) at K				Prefill (s) at K			
u	8K	16K	32K	64K	8K	16K	32K	64K	8K	16K	32K	64K
Dense			100.0				99.0				35.72	
4,096	3.0	19.0	60.0	89.0	6.0	14.0	32.0	62.0	11.30	12.46	14.80	19.96
2,048	11.0	35.0	84.0	90.0	7.0	20.0	49.0	81.0	11.98	13.26	15.72	20.75
1,024	25.0	65.0	87.0	94.0	17.0	33.0	72.0	91.0	12.95	14.55	17.21	22.24
512	56.0	79.0	93.0	97.0	34.0	62.0	83.0	95.0	14.81	16.83	20.09	25.09
256	81.0	86.0	93.0	99.0	62.0	78.0	92.0	98.0	19.86	22.60	26.64	31.29
128	82.0	89.0	97.0	100.0	68.0	87.0	96.0	96.0	30.10	34.25	40.17	44.82

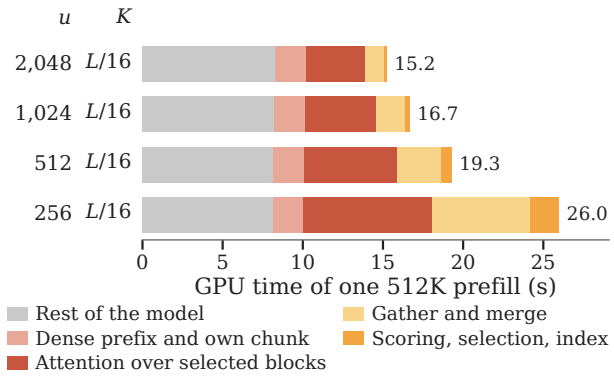

Figure 17: GPU time of one 512K prefill on Nemotron 3.5 Lightning at $K = L/16$ by cluster size u, drawn as Figure 7.

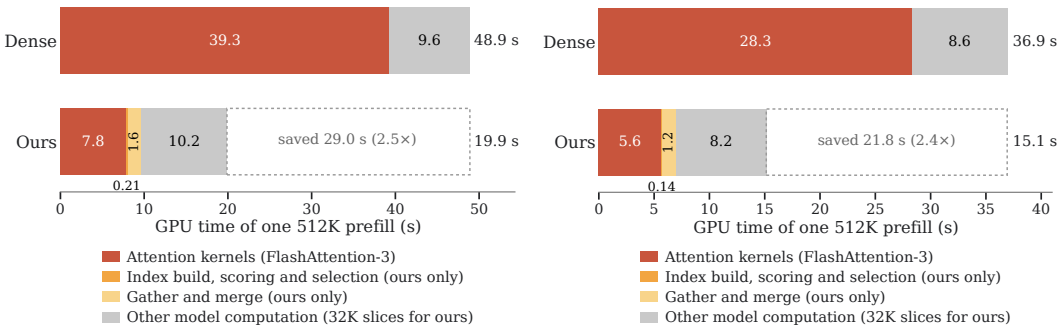

Figure 18: The 512K prefill profile of Qwen3.5-35B-A3B at $K = L/16$ (left) and of Nemotron 3.5 Lightning at $u = 2{,}048$ (right), drawn as Figure 4, with GPU seconds by kernel group from one profiled prefill of each configuration.

I PRIOR SPARSE-ATTENTION METHODS

Table 35 summarizes how the sparse-attention methods cited in this paper are obtained, where they are applied, and how they are evaluated. Only two of them are evaluated on a hybrid model, which interleaves linear-attention or state-space layers with a few attention layers. QSA trains its indexer together with the model. The concurrent SpotAttention trains an indexer for frozen models, including hybrid Qwen3.5 models, and keeps prefill dense, but it measures speed only on a model that is not hybrid. None of the training-free methods is evaluated on a hybrid model in its own paper. UniPrefill, a token-pruning method not included in this table, later measured several of the prefill methods on hybrid models up to 128K tokens (Fan et al., 2026b). Several methods time only the attention layer or kernel, which would overstate the gain on a hybrid model, where attention is a smaller share of the cost.

Table 35: Sparse-attention methods as their papers describe and evaluate them. The groups are training-free methods aimed mainly at prefill, training-free methods aimed mainly at decoding, and methods with learned components. Learned components are those trained for sparsity beyond the pretrained model. Whole-model time means that the paper times the whole model, such as end-to-end latency, time to first token, or time per output token, rather than only the attention layer or kernel.

Method	Learned components	Sparse phases	Hybrid model	Whole-model time
MInference (Jiang et al., 2024)	None	Prefill	No	Yes
SampleAttention (Zhu et al., 2025)	None	Prefill	No	Yes
FlexPrefill (Lai et al., 2025)	None	Prefill	No	No
FlashPrefill / V2 (Fan et al., 2026a;c)	None	Prefill	No	Yes
QuoKA (Jones et al., 2026)	None	Prefill, decode	No	Yes
Quest (Tang et al., 2024)	None	Decode	No	Yes
Squeezed (Hooper et al., 2025)	None	Prefill, decode	No	No
ClusterKV (Liu et al., 2025)	None	Decode	No	Yes
Tactic (Zhu et al., 2026)	None	Decode	No	Yes
SAAP (Mazaré et al., 2025)	Query classifier	Decode	No	No
SpotAttention (Ahmad & Yun, 2026)	Indexer	Decode	Yes	Yes
LServe (Yang et al., 2025)	Head gates	Prefill, decode	No	Yes
RTPurbo (Zhou et al., 2026)	Indexer and model	Prefill, decode	No	No
MoBA / FlashMoBA (Lu et al., 2025; Xiao et al., 2025)	Model	Prefill	No	No
DSA (DeepSeek-AI, 2025)	Indexer and model	Prefill, decode	No	No
MSA (Lai et al., 2026)	Indexer and model	Prefill, decode	No	No
QSA (Qiu et al., 2026)	Indexer and model	Prefill, decode	Yes	No
HISA (Xu et al., 2026)	Indexer and model	Prefill, decode	No	Yes
This work	None	Prefill, decode	Yes	Yes

Among the training-free methods, ClusterKV and Tactic build a key index alongside dense prefill and use it only in decoding (Liu et al., 2025; Zhu et al., 2026), and Squeezed builds its index offline for a fixed context, such as a document, and sparsifies attention from new input to it (Hooper et al., 2025). Our method builds the index for each prompt within the request and uses it to sparsify that prompt's own prefill. SAAP instead fits a key codebook on a corpus and learns a query classifier (Mazaré et al., 2025). Our corpus-fitted centroids in Section 5.4 are scored with the model's own queries, without SAAP's learned query classifier, so they do not reproduce SAAP.

Methods that skip attention blocks inside the kernel or reuse selections across layers, such as BLASST (Yuan et al., 2026), Sketch&Walk (Le et al., 2026), Kascade (Deshmukh et al., 2025), and IndexCache (Bai et al., 2026), address complementary parts of the computation and are not listed.

J EXAMPLES OF CLUSTER MEMBERS

Figure 3 examines one cluster from an MRCR prompt and one from an MK-3 prompt. Each highlighted span is a single member token, with surrounding text retained for interpretation. Cosine similarity is measured between the normalized pre-RoPE key and its assigned centroid. The partition shown is fitted to the unmodified model's keys, which are the keys our method clusters, so these are examples of the blocks rather than of what a sparse run selected. Each partition contains 256 clusters, and layer, KV-head, and token-position indices are zero-based. The figure shows the three nearest and three farthest members of each cluster among the saved examples.

The MRCR cluster groups environmental vocabulary across several passages. Its lower-similarity members extend to medicine, history, and mathematics. For MK-3, the nearest members occur at the beginning of key identifiers, but lower-similarity members also include the beginning of a value identifier. The cluster thus has a recognizable structural pattern without assigning every member the same role. These examples describe the groups formed by clustering, rather than which groups are important to a particular query.

Mixing token identities is common in the inspected prompts. Across two MK-3 and two MRCR prompts, about 70% and 90% of clusters, respectively, have no single token type occupying more than half their members. These counts use all cluster members across eight layers and four KV heads per prompt.

Cosine	Position	MRCR · layer 3, KV head 0, cluster 111
0.942	116899	… skies, You paint the horizons with **colors** that fly. Each river that flows, …
0.941	117439	… organic art. Blossoms burst with **colors** bold, In hues of red and pink …
0.939	13679	… us into fall with its crisp air and **colors**. It's a time of change and …
		…
0.684	81616	… animals showcasing warning **colors** or camouflage, and landscapes …
0.651	231577	… Binary numbers, composed of 0s and 1**s**, form the foundation of computer …
0.627	122892	… and those with stainless steel **blades** to prevent rusting. Information on …

Cosine	Position	MRCR · layer 11, KV head 1, cluster 208
0.903	268600	… From a psychological perspective**,** colors have profound effects on our …
0.886	268466	… psychological responses, serving **as** a powerful form of non-verbal …
0.882	268442	… affecting our emotions, behaviors**,** and even cultural practices. The vibran …
		…
0.551	111265	… Pens come in every color of the **rainbow**! Brighten up your notes and …
0.540	268568	… discovery that white light can **be** split into a range of colors using a …
0.522	268570	… discovery that white light can be split **into** a range of colors using a prism, …

Cosine	Position	MRCR · layer 3, KV head 0, cluster 41
0.970	94152	… an inclusive atmosphere where **their** voices are heard leads to better …
0.964	111494	… prosthetics, striving to replicate **their** dexterity and functionality. …
0.963	70391	… them from a distance, appreciating **their** complexity and charm. Yours …
		…
0.736	243256	… of Frogs in Ecosystems and **Their** Conservation I hope this letter …
0.733	110722	… by a variety of houses, each with **its** own unique character. There was the …
0.690	203583	… and the resources required to solve **them**. It investigates the limits of what …

Cosine	Position	MK-3 · layer 23, KV head 3, cluster 1
0.953	412289	… special magic uuids for 3505ceec-6bd8-**4**19f-b6b6-797 …
0.940	453901	… special magic uuids for 33549404-05cf-**4**258-9c84-c61 …
0.940	499923	… special magic uuids for 3491836b-b504-**4**ee9-8893-e959 …
		…
0.630	19026	… special magic uuids for 645e0314-**4**6b4-45a8-ac64 …
0.626	14880	… special magic uuids for dafc21a6-566d-**4**6e0-8cf5-e57afa …
0.615	820	… special magic uuids for bb576d70-d4f7-**4**7fd-bcb9-c8a485 …

Cosine	Position	MK-2 · layer 27, KV head 0, cluster 70
0.835	487952	… special magic numbers for **troubled**-arena is: 815482 …
0.826	422558	… special magic numbers for **troubled**-apology is: 173206 …
0.825	432399	… special magic numbers for **weary**-zero is: 2266208 …
		…
0.465	129	… special magic numbers for **nervous**-assist is: 383514 …
0.456	195	… special magic numbers for **guilt**less-asterisk is: 6811 …
0.443	29316	… is: 1419399. One of **the** special magic numbers for …

Figure 19: Five additional clusters containing color vocabulary, tokens in a passage about color, pronouns, UUID digits, and adjectives in MK-2 keys. Positions are token indices, and highlights mark member tokens. Rows are sorted by key–centroid cosine similarity, with a gap between the two ends.

J.1 Further clusters

Figure 19 shows five further clusters, using three members from each end of the saved cosine range. The first and third clusters use the same MRCR prompt as Figure 3, the second and fourth use separate MRCR and MK-3 prompts, and the fifth uses an MK-2 prompt.

The first cluster groups occurrences of *colors* across descriptions, while its lower-similarity members extend to other words. The second includes punctuation and function words within a passage about

color perception, illustrating how context can connect tokens with different surface forms. The third groups pronouns and possessives across different topics. In the fourth, the nearest members are the digit 4 at the start of the third group of a key UUID. One lower-similarity member instead occurs in the second group. The MK-2 prompt consists of lines such as “One of the special magic numbers for troubled-arena is: 8154823.” All ten of its saved nearest members are negative adjectives in different keys, such as *troubled*, *weary*, *abusive*, *disgusted*, and *ashamed*, while its lowest-similarity members include other tokens of the same lines. These examples illustrate lexical, contextual, grammatical, and structural similarities among keys, without implying that any cluster contains only one kind of token.