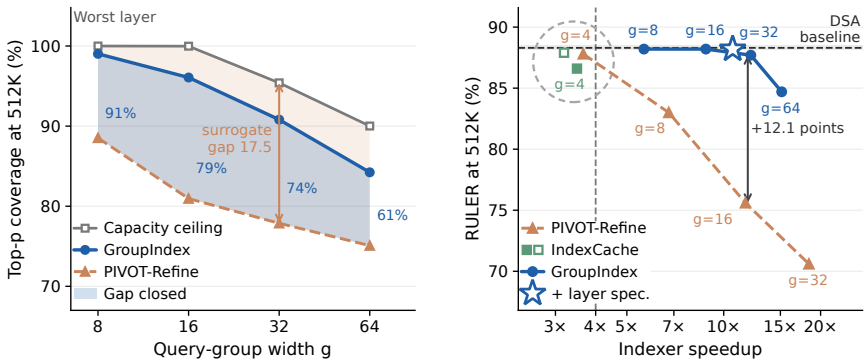
GROUPINDEX: ACCELERATING SPARSE ATTENTION VIA LEARNED CROSS-QUERY INDEXING

Anonymous authors
Paper under double-blind review

ABSTRACT

DeepSeek Sparse Attention (DSA) makes attention cheap by keeping only a few keys per query, but its lightweight indexer still scans the whole prefix for every query, and this scan comes to dominate long-context prefill. Existing accelerators reuse selections across layers, activate fewer indexer heads or filter key blocks. Sharing across layers and heads is capped by the fixed numbers of indexer layers and heads, and accuracy grows harder to keep as it nears that cap. Filtering key blocks instead trades the resolution of their summaries against recall. Sharing one scan across neighboring queries has no such fixed cap, since how many can share depends on their similarity and on context length. Existing query-group indexing, however, averages the queries of each group and loses accuracy as the group widens. We show that this width limit is representational rather than a matter of capacity. A shared scan must keep every key that any member selects, which calls for the maximum of the members' scores, whereas averaging their queries dilutes each member's own needs and loses keys that are important to individual members. We introduce **GroupIndex**, which jointly learns head-wise query aggregation and scoring projections in a Surrogate Indexer, and applies hardware-aware layer specialization to select each layer's query-group width and candidate budget. It indexes each query group with one scan whose surrogate query is learned rather than averaged. Training targets the group's selection union, uses the native indexer's own selections as labels, and runs offline on activations cached once, without loading the backbone. At inference the Surrogate Indexer runs as a sidecar ahead of the native indexer and proposes candidates for each group, among which the native indexer still makes each query's final selection. On DeepSeek-V4-Flash, GroupIndex matches the dense indexer's accuracy on RULER, LongBench-v2 and LongMemEval from 64K to 512K tokens, and at 1M tokens it speeds up the indexer about 11× and end-to-end prefill up to 1.9×.



(a) The limit is representational, not capacity **(b) Comparison with existing methods**

Figure 1: **GroupIndex closes most of the surrogate gap limiting cross-query sharing. (a)** GroupIndex stays near the capacity ceiling as groups widen, unlike mean-pooled PIVOT-Refine, and **(b)** it keeps dense-level RULER accuracy up to about 11× indexer speedup, whereas PIVOT-Refine degrades with wider groups, as does IndexCache.

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.

1 INTRODUCTION

Long context windows are now standard in large language models such as GPT-5.5 (OpenAI, 2026), Claude Opus 4.7 (Anthropic, 2026), Gemini 3 (Google DeepMind, 2025), Qwen3.5 (Qwen Team, 2026), GLM-5 (GLM-5-Team et al., 2026) and DeepSeek-V3.2 (DeepSeek-AI, 2025), and prefill bears most of the quadratic cost of attention (Vaswani et al., 2017). DeepSeek Sparse Attention (DSA) removes much of this cost by letting a lightweight indexer pick the top-k keys for each query, which cuts core attention over a T-token prompt from $O(T^2)$ to $O(Tk)$ (DeepSeek-AI, 2025). Yet the quadratic term does not disappear but moves into the indexer, which still scores every preceding key for every query. DeepSeek-V4-Flash inherits this design in Compressed Sparse Attention (CSA) (DeepSeek-AI, 2026), whose compression shrinks each scan only by a constant factor, so indexing still costs $\Theta(T^2)$ and takes about 63% of prefill FLOPs at one million tokens.

Prior work shrinks this indexing cost along three axes, each bounded in its own way. IndexCache reuses selections across layers (Bai et al., 2026), MISA activates a query-dependent subset of indexer heads (Zhou et al., 2026), and HISA filters key blocks before native refinement (Xu et al., 2026). On the layer and head axes, sharing is capped by the fixed numbers of indexer layers and heads, and accuracy grows harder to keep as it nears that cap. The key axis trades the resolution of its block summaries against refinement cost and recall. None of them changes the fact that every query searches the prefix on its own. Nearby queries select heavily overlapping keys (Liu et al., 2026), so one scan shared across a group of queries can amortize that repeated search. How wide the group can be depends on how similar the queries are and on the context length rather than on a fixed layer or head count, which gives the query axis a higher ceiling for amortization.

Reaching that ceiling requires one representation to stand in for a whole group, and PIVOT-Refine pools nearby queries into a surrogate query whose candidates the native indexer refines per query (Liu et al., 2026). Its accuracy falls as the group widens, but the cause is the shared representation rather than the candidate budget (Figure 1a). In the hardest layer at 512K, a group of 32 queries loses under five points of top-p coverage to the finite budget, yet the mean-pooled surrogate falls 17.5 points below that ceiling, a shortfall that grows with context length and that we call the **surrogate gap**. A shared scan must keep every key that any member selects, which calls for the maximum of the members' scores, whereas pooling averages their queries and dilutes the keys that only a few members need. A larger budget admits more keys without ranking these higher, and no fixed pooling rule escapes the dilution, as max pooling even falls below the mean (§5.4). The surrogate query should therefore be learned toward the union, with the group width chosen per layer, since layers differ widely in how much sharing they tolerate.

We introduce **GroupIndex**, which jointly learns head-wise query aggregation and scoring projections in a Surrogate Indexer, and applies hardware-aware layer specialization to select each layer's query-group width and candidate budget. It indexes each group of queries with a single scan and, in place of PIVOT's training-free pooling, learns the surrogate query used in that scan. The scoring projections form each position's query and gate, and the head-wise query aggregation mixes the positions with separate learned weights for every head, so the shared score can follow keys that only some members want. At inference the Surrogate Indexer runs as a sidecar ahead of the native indexer. It proposes shared candidates for each group, inside which the native indexer still makes each query's final selection, so the rest of the model sees the native interface. Training targets the group's selection union, takes its labels from the native indexer's own selections and runs offline on cached activations without loading the backbone. Hardware-aware layer specialization then sets the group width and candidate budget of each layer from kernel latency profiled on the target GPU. We summarize our contributions as follows.

- **GroupIndex.** We propose GroupIndex, whose learnable Surrogate Indexer replaces the pooled surrogate that limits prior query-sharing methods. It learns head-wise query aggregation and scoring projections jointly toward the members' selection union. One scan with this learned surrogate query serves a whole group of g queries, so the group's indexing cost falls from $O(gT)$ toward $O(T)$. The sidecar adds about 0.095% of the backbone's parameters, leaves each final selection to the native indexer, and trains offline on cached native selections without loading the backbone.
- **Hardware-aware layer specialization.** We fold sharing granularity and hardware cost into one decision. Subject to each layer's own coverage floor, GroupIndex picks the group

width and candidate budget with the lowest profiled kernel latency, turning cross-layer heterogeneity directly into speed. Under the 96% floor of Table 1, this lifts the indexing speedup from 7.31× for the fastest uniform plan that meets it to 10.66×.

- **Results.** On DeepSeek-V4-Flash, GroupIndex matches the native dense indexer on RULER, LongBench-v2 and LongMemEval from 64K to 512K tokens, while cutting indexing latency by roughly 11× and prefill latency by up to 1.9× at 1M tokens.

2 Related Work

Sparse attention and its indexer. Block-sparse attention keeps or drops contiguous key blocks, with patterns that are fixed (Child et al., 2019; Beltagy et al., 2020; Zaheer et al., 2020; Xiao et al., 2024), estimated per input (Jiang et al., 2024; Xu et al., 2025; Zhang et al., 2025; Wang et al., 2026b; Niu et al., 2026) or learned (Gao et al., 2025b; Yuan et al., 2025; Lu et al., 2025). Token-level selection in DSA (DeepSeek-AI, 2025) and DeepSeek-V4-Flash (DeepSeek-AI, 2026) avoids the coarseness of blocks, but its indexer scans the prefix for every query. IndexCache (Bai et al., 2026), MISA (Zhou et al., 2026) and HISA (Xu et al., 2026) shrink that scan across layers, heads and key blocks. Sharing across layers and heads is capped by the architecture and loses accuracy more readily near that cap, since distant layers select different keys and the indexer relies on the diversity of its heads. HISA's block summaries dilute their keys as blocks grow, and every query still searches the prefix on its own.

Sharing across queries. Sharing one scan across neighboring queries removes this repetition. PIVOT (Liu et al., 2026) averages each group's queries into one surrogate query and attributes its accuracy loss beyond groups of four to the union outgrowing the budget, which O3 (§3.2) traces instead to the mean-pooled query. LSA (Wang et al., 2026a) shares selections across decode steps and already trains on their union. These methods, and the query-block selections of MInference and SeerAttention, fix how positions are combined. GroupIndex learns the combination with its head-wise query aggregation, jointly with the scoring projections.

Learning the selector. Learning a selector on a frozen backbone is itself common, as in SeerAttention, VSPrefill (Chen, 2026), SparDA (Fu et al., 2026), LISA (Zhao et al., 2026) and LSA, mostly by matching an attention or selector distribution. GroupIndex keeps the frozen backbone but supervises only whether each key belongs to the selection union, with a pointwise BCE (§4.2).

Budget allocation. PyramidKV (Cai et al., 2025), MoA (Fu et al., 2025) and DuoAttention (Xiao et al., 2025) divide a key-side budget among layers or heads under a global constraint, while FlexPrefill (Lai et al., 2025), VSPrefill and Twilight (Lin et al., 2025) adapt budgets per input until a cumulative-attention threshold is met. Hardware-aware layer specialization adds the query-side group width, chosen jointly with the candidate budget under a per-layer coverage floor that plays the role of that threshold. With no global budget to share, each layer's choice is exact and independent, so the plan is solved once offline against latency profiled on the target GPU (§4.4).

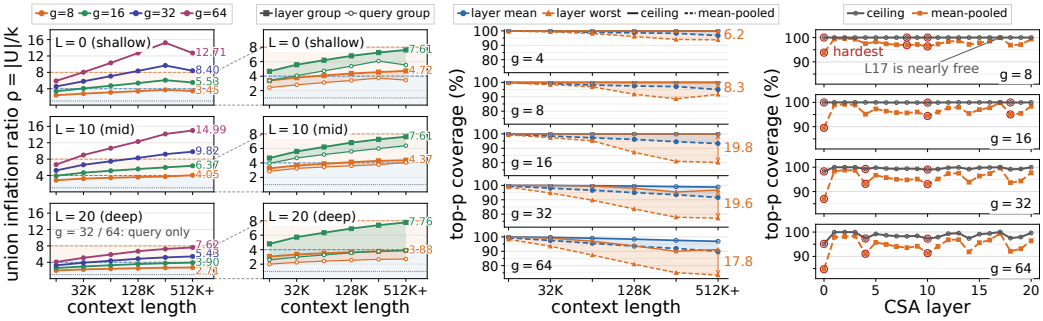
3 Preliminary

3.1 Background and setup

Sparse attention with a lightning indexer. DeepSeek Sparse Attention (DSA) (DeepSeek-AI, 2025) lets each query attend to only a few tokens. A lightweight *lightning indexer* scores every preceding token s against the current query token t,

$$I_{t,s} = \sum_{h=1}^{H^I} w^I_{t,h} \cdot \mathrm{ReLU}\big(\mathbf{q}^I_{t,h} \cdot \mathbf{k}^I_s\big), \qquad \mathcal{S}_t = \{\, s \mid I_{t,s} \in \text{Top-}k(I_{t,:}) \,\}, \tag{1}$$

where $\mathbf{q}^I_{t,h}$, $\mathbf{k}^I_s$ and $w^I_{t,h}$ are the indexer's per-head query, key and weight over H^I heads, and attention runs only over $\mathcal{S}_t$, the indexer's sole link to the rest of the model. This work targets the 21 Compressed Sparse Attention (CSA) layers of DeepSeek-V4-Flash (DeepSeek-AI, 2026), whose indexers score compressed keys $\mathbf{k}^{I\text{Comp}}_s$ and keep a top-k set $\mathcal{T}_t \equiv \mathcal{S}_t$ of k=512 keys per query.



(a) Union inflation: query vs layer groups **(b) Surrogate gap vs length** **(c) Per-layer coverage**

Figure 2: **Nearby queries overlap heavily, yet mean pooling, not the budget, limits sharing. (a)** Union inflation of query and layer groups. **(b)** Surrogate gap. **(c)** Per-layer coverage.

Sharing one scan. Every query still scans the whole prefix, so we let a group G of g native top-k selections share one scan, taken either at g neighboring query positions in one layer (the *query axis*) or at one position across g neighboring layers (the *layer axis*). With $\mathcal{T}_j$ the native top-k set of member j, the group's **selection union** is $U = \bigcup_{j \in G} \mathcal{T}_j$. A *shared scan* scores the prefix once and proposes a candidate set A of at most c keys, the *candidate budget*, inside which the native indexer selects each member's top-k. The **union inflation ratio** $\rho = |U|/k \in [1, g]$ measures how far the members disagree, and a budget holds the union exactly when $\rho \leq c/k$.

Coverage and capacity ceiling. Two measures, averaged over the members, quantify how much of the native selection a candidate set A preserves,

$$\mathrm{cov}_p(A) = \frac{1}{g} \sum_{j \in G} \frac{\sum_{n \in A \cap \mathcal{T}_j} p_j(n)}{\sum_{n \in \mathcal{T}_j} p_j(n)}, \qquad \mathrm{cov}_k(A) = \frac{1}{g} \sum_{j \in G} \frac{|A \cap \mathcal{T}_j|}{|\mathcal{T}_j|}. \tag{2}$$

Top-p coverage, written $R(A) \equiv \mathrm{cov}_p(A)$, is the primary measure because weighting each selected key by its native indexer mass $p_j(n) = \mathrm{softmax}(I_{j,\cdot})_n$ tracks more closely what attention loses. The stricter top-k coverage, which counts keys equally, serves as a secondary check. The **capacity ceiling** $R^\star(c)$, the coverage of a union oracle that keeps as much of U as a budget of c allows, equals 1 when $c \geq |U|$ and can only overstate the loss that the budget forces.

3.2 Observations

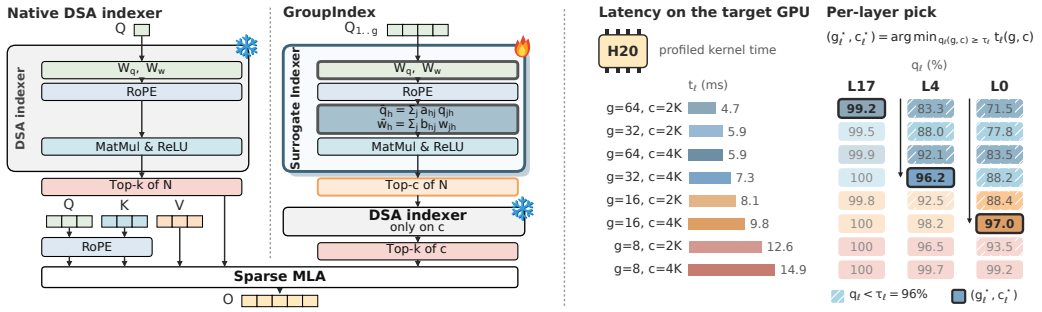
O1: The redundancy is real, and the query axis exploits it better than the layer axis. Sharing pays only if members largely agree, keeping ρ far below its worst case g. Along the query axis they do, and ρ grows so much more slowly than g that a budget of $c = 4096$ holds the union of up to 16 queries at every length shown (Figure 2a, left). The layer axis costs more, with a larger ρ at every matched size, length and layer shown (Figure 2a, right). A layer group also spans at most the 21 CSA layers, and sharing near that cap carries a high risk of accuracy loss. A query group faces no such cap, since its query pool grows with the prompt, along with the cost that sharing amortizes.

O2: Sharing need not erase individual selection. At $c = k$ the group is forced onto one shared top-k that discards the members' own selections, the route of PIVOT-Reuse and of cross-layer reuse such as IndexCache. With $c > k$ the work splits, as in PIVOT-Refine, and once the proposed A contains U, rescoring it reproduces every $\mathcal{T}_j$. The union is therefore the shared scan's only target, and the question becomes whether one shared representation can place U inside a budget-sized A.

O3: The mean-pooled surrogate falls short, and the shortfall is representational rather than budgetary. PIVOT's answer is a mean-pooled surrogate query (the *proxy* of PIVOT), which averages the members' queries. Its total loss splits in two,

$$\underbrace{1 - R(A)}_{\text{total loss}} = \underbrace{\big(1 - R^\star(c)\big)}_{\text{capacity loss}} + \underbrace{\big(R^\star(c) - R(A)\big)}_{\text{surrogate gap}}, \tag{3}$$

where the second term, the **surrogate gap**, is what the representation forfeits within the budget. That gap widens with context at every width and is most severe in the worst layer, which sets deployed



(a) Native DSA indexer vs. GroupIndex **(b) Hardware-aware layer specialization**

Figure 3: **GroupIndex learns a shared query per group and picks each layer's (g, c) by latency.** **(a)** A Surrogate Indexer proposes a shared top-c that the frozen native indexer re-ranks. **(b)** Each layer takes the fastest setting (H20, 512K) whose top-p coverage clears 96%.

quality because prefill runs the layers in series (Figure 2b). In the longest contexts the capacity ceiling stays high, yet the mean-pooled surrogate falls far below it, and max pooling at the same budget does worse still (§5.4). The shortfall is therefore representational: a larger candidate set cannot recover keys that the shared query never ranked. Capacity becomes a second constraint only once the union outgrows the budget, and even then it catches the gap only under the stricter top-k coverage (Figure 6).

O4: The gap is a max-versus-mean mismatch, which no fixed pooling closes. To cover the union, a shared score must rank a key by whether any member selects it, that is, by the pointwise maximum of the members' indexer scores,

$$s_U(n) = \max_{j \in G} I_{j,n} = \max_{j \in G} \sum_h w^I_{j,h} \, \mathrm{ReLU}\big(\mathbf{q}^I_{j,h} \cdot \mathbf{k}^{I\mathrm{Comp}}_n\big). \tag{4}$$

A mean-pooled surrogate query instead scores keys with the members' average, in which a direction that only one member wants keeps $1/g$ of its weight, so the keys along it fall below the cut. A longer prefix gives each member more keys of its own, which raises ρ (Figure 2a) and is why the gap widens with context. No fixed pooling rule escapes this, since pooling hidden states or averaging queries only mixes the members' own queries at preset weights (Appendix E.3). The query that best serves the union need not be such a mix, which points toward learning the surrogate query itself.

O5: Tolerance for sharing differs across layers but is stable within each. Learning addresses how well one query serves a group, not how wide the group should be, and that width differs across layers (Figure 2c). The ranking barely moves with g, with L0 the hardest layer at every width and L17 nearly free even at $g = 64$. A single global width is therefore pinned by the least tolerant layer, whereas a stable ranking lets the width be chosen per layer once before inference rather than per input. A learned shared representation (O3, O4) and a width and budget specialized per layer (O5) are thus the two pillars of GroupIndex.

4 Method

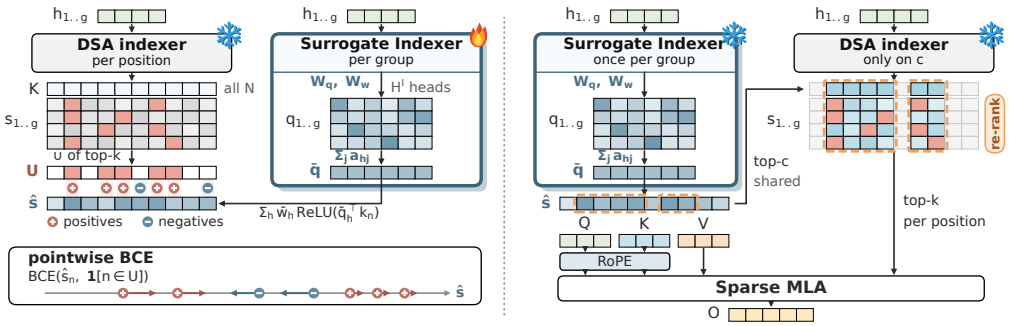
4.1 The Surrogate Indexer

Scoring projections. At each position j of a group G, W_q and W_w map the hidden state h_j to a query $q_{j,h} = (W_q \, h_j)_h$ and a gate $w_{j,h} = (W_w \, h_j)_h$ for each indexer head h, in the native form of §3.1 with its low-rank factorization and rotary.

Head-wise query aggregation. The per-position queries and gates are then contracted into one query and one gate per head, with learned weights $a, b \in \mathbb{R}^{H^I \times g}$,

$$\bar{q}_h = \sum_{j \in G} a_{hj} \, q_{j,h}, \qquad \bar{w}_h = \sum_{j \in G} b_{hj} \, w_{j,h}. \tag{5}$$

Each head weighs the positions on its own, so different heads can follow different members of the group. The aggregation has to act on queries, before the scan. The score is linear in the query up to



(a) Training Surrogate Indexer (b) Inference with Surrogate Indexer

Figure 4: **The Surrogate Indexer learns union membership and only proposes candidates. (a)** The union of the frozen native indexer's top-k sets labels keys for a pointwise BCE. **(b)** One scan per group keeps a shared top-c that the native indexer re-ranks per position.

the ReLU, so one aggregated query costs one pass over the keys, whereas combining per-position scores would cost g passes, the very cost that grouping removes.

Surrogate score. The surrogate query scores all N keys in one pass, $\hat{s}_n = \sum_h \bar{w}_h \,\mathrm{ReLU}(\bar{q}_h \cdot \mathbf{k}_n^{I\mathrm{Comp}})$, and its top-$c$ keys form the shared candidate set A. With native weights and $a = b = 1/g$ it is PIVOT's mean-pooled surrogate query, the special case with fixed $\bar{q}_h$ and $\bar{w}_h$.

The two parts are trained together because each falls short alone. Reweighting native queries cannot add a direction they lack, and new projections under a fixed average leave a direction that only one position wants with $1/g$ of its weight (O4), since averaging commutes with the linear projections. Trained together, the projections can supply the missing directions and the aggregation can give them weight, and in deep layers neither part alone recovers most of the gain (Table 3). The claim is bounded. Because the ReLU acts after the aggregation, no single scan reproduces the per-position maximum, so the Surrogate Indexer learns an approximation to the ranking that the union needs.

4.2 Training

Pointwise BCE loss. The labels come from the native indexer itself. Run frozen over the prefix, it gives each position $j \in G$ its top-k set $\mathcal{T}_j$, and each key is labeled by whether it belongs to their union U (Figure 4a). The positives $\mathcal{P} = U$ are the keys some position selects natively, and the negatives $\mathcal{N}$ are sampled from the keys outside the union. The surrogate score $\hat{s}_n$ is trained against these labels with a pointwise binary cross-entropy,

$$\mathcal{L} = \operatorname*{mean}_{n \in \mathcal{P} \cup \mathcal{N}} \mathrm{BCE}\big(\hat{s}_n,\ \mathbf{1}[n \in U]\big). \tag{6}$$

A key is either selected by some position or not, and the BCE places positives high and negatives low on one absolute scale, so the scores do not drift as a set.

4.3 Inference

Surrogate scan. The Surrogate Indexer scans the prefix once per group and keeps the top-c keys of $\hat{s}$ as the shared candidate set A (Figure 4b).

In-candidate re-ranking. Each position then re-ranks its candidate set $\mathcal{C}_j$, built from A, with its own native query and gate, applying Eq. equation 1 over $\mathcal{C}_j$ instead of the whole prefix, and passes its top-k to the Sparse MLA, recovering $\mathcal{T}_j$ whenever $\mathcal{C}_j$ contains it (O2).

Cost and deployment. The full-prefix scan that the native indexer would repeat for every position is paid once per group, so scoring a group over the prefix's N compressed keys costs $O(N + gc)$ instead of the native $O(gN)$. Only this proposal step changes, so the Sparse MLA still attends through the native selection, no key leaves the cache, and the rest of the model sees the interface of native DSA. The module therefore deploys as a sidecar with the model weights unchanged, and its per-layer Surrogate Indexers together hold about 270M parameters, 0.095% of the backbone's 284B, nearly all in the scoring projections.

Table 1: **GroupIndex keeps near-dense accuracy at four times the group width of PIVOT-Refine. Bold** and underline mark the best and second-best non-baseline method.

Method	LongBench-v2			LongMemEval		RULER					Speedup
	Short (33K)	Medium (127K)	Long (391K)	S (112K)	M (509K)	64K	128K	256K	512K	AVG	@ 512K
DSA (baseline)	55.0	49.3	54.6	85.6	40.4	96.6	96.0	89.8	88.3	92.68	1.00×
Random (g=4)	48.3	36.7	38.0	54.2	23.6	71.9	61.2	48.8	43.9	56.45	3.66×
Recency (g=4)	47.8	35.8	45.4	35.8	21.4	58.9	51.5	43.0	41.5	48.73	3.66×
IndexCache (Reuse, g=4)	53.3	45.1	53.7	87.0	41.0	95.4	94.5	88.2	86.6	91.18	3.50×
IndexCache (Refine, g=4)	55.6	47.4	51.9	86.6	40.8	96.0	95.5	89.0	87.9	92.10	3.19×
PIVOT-Refine (g=4)	55.6	50.2	55.6	85.2	41.4	**96.4**	95.6	89.7	87.8	92.38	3.66×
PIVOT-Refine (g=8)	**57.2**	**50.7**	54.6	85.4	41.4	95.8	94.1	86.5	83.0	89.85	6.75×
GroupIndex (g=16)	56.7	49.8	**56.5**	**87.8**	**41.8**	96.0	**95.7**	89.6	**88.2**	92.38	8.80×
↪ + layer spec.	54.4	**50.7**	55.6	85.8	41.4	96.1	95.4	**89.9**	**88.2**	**92.40**	**10.66**×

4.4 Hardware-aware layer specialization

The group width g and the candidate budget c, left open so far, need not be global choices. The width a layer tolerates is stable and differs widely across layers (O5), so giving each layer its own (g, c) turns the tolerance that one global setting leaves unused into speed (Figure 3b).

Exact per-layer plan. The per-layer choice can be solved exactly because the layers do not compete. A layer's coverage depends only on its own (g, c), and prefill runs the layers in sequence, so their latencies add. The fastest plan that keeps every layer above its floor τ_ℓ therefore splits into one choice per layer, the fastest setting that clears that layer's floor,

$$\min_{\{(g_\ell, c_\ell)\}} \sum_\ell t_\ell(g_\ell, c_\ell) \;\text{ s.t. }\; R_\ell(g_\ell, c_\ell) \geq \tau_\ell \;\; \forall \ell \quad \Longrightarrow \quad (g_\ell^\star, c_\ell^\star) = \underset{R_\ell(g,c)\geq\tau_\ell}{\arg\min}\; t_\ell(g, c), \tag{7}$$

where t_ℓ and R_ℓ are the latency and the offline top-p coverage of a setting at layer ℓ.

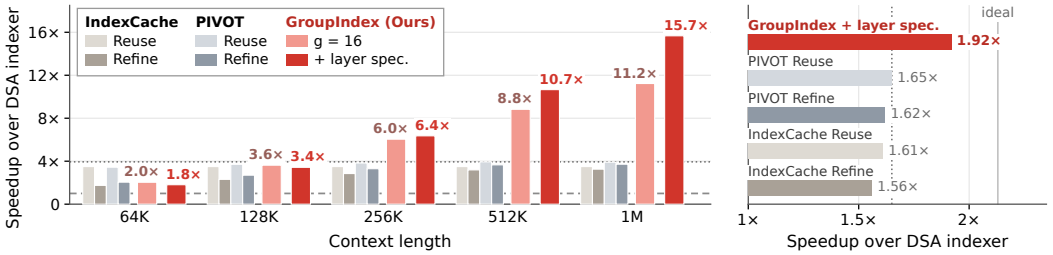
Profiled latency. The latency t_ℓ is profiled on the target GPU rather than counted in FLOPs, since a fixed per-layer cost that does not shrink with g makes FLOPs overstate what a wider group saves. In Figure 3b, an eightfold wider group at the same budget cuts H20 latency by less than a factor of three. Minimizing latency spends the slack of easy layers, lowering layer-averaged coverage by about a point, the right trade when the worst layer sets output quality.

5 Experiments

5.1 Setup

All experiments use DeepSeek-V4-Flash (DeepSeek-AI, 2026), with fixed configurations at $c = 2048$ or $c = 4096$ and a layer-specialized plan picking (g_ℓ, c_ℓ) per layer under a worst-layer top-p floor of $\tau = 96\%$. The Surrogate Indexers of all 21 layers train with the pointwise BCE of §4.2 in about 37 H20 GPU-hours in total at $g = 16$ (Appendix D.3). Baseline-anchored initialization makes step 0 coincide with PIVOT's mean-pooled aggregation (Appendix B.1). Speedup is that of the indexer's scoring stage over the dense indexer. End-to-end prefill runs on eight H20 GPUs with tensor parallelism.

PIVOT-Refine, which we reimplement on V4, runs the same two-stage pipeline as GroupIndex with the training-free mean-pooled surrogate query, and PIVOT-Reuse applies the group's shared top-512 directly. IndexCache, also reimplemented, computes the selection in one Full layer of every four, and the other three Shared layers reuse it (Reuse). We add a Refine variant of our own, in which each Shared layer re-selects its own top-512 from 2048 candidates proposed by its Full layer (Refine). Random and Recency replace only the candidate source and read no content. RULER (Hsieh et al., 2024) covers 13 tasks from 64K to 512K tokens, and LongBench-v2 (Bai et al., 2025) and LongMemEval (Wu et al., 2025) fit prompts into 512K tokens.



(a) Indexer speedup over DSA indexer **(b) End-to-end prefill speedup at 1M**

Figure 5: **Scoring-stage savings grow with context and speed up long-context prefill.** Dotted lines mark the best prior method, and the ideal has no scoring stage.

5.2 Main results

Learning the surrogate pushes out the frontier between accuracy and speed that bounds query sharing (Figure 1b, Table 1). The mean-pooled surrogate of PIVOT-Refine dilutes what each member needs alone (O4), so its RULER score at 512K falls steeply once the group exceeds four queries. GroupIndex learns the shared query instead, matches the dense indexer up to $g = 16$ and leads PIVOT-Refine by 12.1 points at about the same speed. Matching its accuracy forces PIVOT-Refine back to $g = 4$, at less than half the speed of GroupIndex. Cross-layer sharing meets the same limit on an axis where the union grows faster (O1), so IndexCache stays below $4\times$ while its Reuse variant already falls to 86.6 at 512K. Speed thus follows the widest group a method can share while matching the dense indexer.

Accuracy depends mainly on the coverage of the candidate set, because in-candidate re-ranking can only keep keys that the candidate set contains (O2). Random and Recency show this most clearly, losing more than 35 RULER points under the same re-ranking and budget. The surrogate-based methods, by contrast, match the dense indexer on LongBench-v2 and LongMemEval, so at these lengths one shared scan per query group costs little accuracy. Cross-layer sharing loses more, as IndexCache drops further below the dense indexer on the longer splits of LongBench-v2 (O1). Learned and mean-pooled surrogates are nearly indistinguishable on these two suites, while the advantage of learning shows on RULER, where multi-key retrieval depends on the keys that averaging dilutes.

5.3 Efficiency results

Kernel speedup. The kernel speedup grows with context. The surrogate scan reads the whole history but runs once per group, so its cost shrinks by a factor of g, while recomputing the surrogate's queries and re-ranking inside the candidates cost the same at every length (§4.3). At 64K this fixed work still dominates, so GroupIndex only ties PIVOT-Refine at $g = 4$ and trails both Reuse variants. As the history grows the scan takes over, and at 1M the scoring stage at $g = 16$ runs $11.2\times$ faster than the dense indexer (Figure 5a).

End-to-end prefilling speedup. Outside the kernel, the saving is capped by the share of prefill that the scoring stage takes, and that share grows with context, so the end-to-end gain comes mainly from long contexts (Figure 5b). At 1M the stage takes a little over half of prefill time, and GroupIndex with layer specialization reaches $1.92\times$, close to the ideal of removing the stage entirely. Prior methods stay further from this ideal, because they share across only four queries or four layers, which keeps their kernel speedup below $4\times$ and leaves more than a quarter of the scoring time in place.

5.4 Ablations

Representation versus budget. The width limit lies mainly in the representation rather than the budget (Table 2). At a fixed budget, PIVOT-Refine loses accuracy with every doubling of the group, as more members stand behind one averaged query (O4). GroupIndex stays near the dense indexer up to $g = 32$, and its lead grows to 14.5 RULER points at $g = 64$. Their coverage margin does not grow in step, so the lead comes less from how much of the union averaging loses than from which

Table 2: **The learned surrogate widens its lead as the group grows, and doubling c pays off mainly at the widest group.** Bold marks where GroupIndex beats PIVOT-Refine at the same budget and where doubling c helps. Coverage is compared only within a budget.

	RULER: AVG (512K)			**Coverage**: top-p (top-k)			**Speedup** @ 512K		
g	PIVOT-Refine	GroupIndex		PIVOT-Refine	GroupIndex		PIVOT-Refine	GroupIndex	
	c=2048	c=2048	c=4096	c=2048	c=2048	c=4096	c=2048	c=2048	c=4096
8	89.85 (83.0)	**92.48 (88.2)**	92.58 (88.3)	95.19 (91.60)	**98.53 (96.78)**	99.83 (99.64)	6.75×	5.65×	4.78×
16	84.93 (75.6)	**92.38 (88.2)**	92.43 (88.1)	93.04 (88.16)	**96.74 (93.18)**	99.40 (98.73)	11.71×	8.80×	7.31×
32	80.00 (70.6)	**92.55 (87.7)**	92.45 (88.4)	91.63 (85.94)	**95.00 (89.98)**	98.69 (97.34)	18.39×	12.14×	9.82×
64	76.65 (67.1)	**91.13 (84.7)**	**91.98 (86.5)**	90.08 (83.57)	**92.78 (86.73)**	97.29 (95.04)	25.41×	15.14×	12.02×

keys, those that only a few members need. Doubling the budget costs speed at every width, yet it moves the RULER average of GroupIndex by less than a point, mostly at $g = 64$, because there the union of the members' selections no longer fits in the budget (O3). Speed should therefore come from wider groups, whose accuracy learning keeps close to the dense indexer, and a larger budget is worth its cost only where the union no longer fits.

Joint training of the two parts. Training the head-wise query aggregation and the scoring projections together raises top-p coverage more than training either part alone in every layer shown, as §4.1 anticipates (Table 3). In deep layers the joint gain is about three to five times that of the better part alone. Much of it comes from the interaction, the gain that neither part delivers alone, which supplies over three-fifths of the 4.24-point joint gain at L10 and grows with context (Table 13). Joint training therefore carries most of the gain in deep layers at long context and is central to the design.

Table 3: **Aggregation and projections are complementary.** Top-p gain over step 0 at g=16, c=4K. Inter. = Both − Agg. only − Proj. only.

Layer	Step 0	Agg. only	Proj. only	Both	Inter.	
					All	>256K
L0	89.54	+5.43	+4.79	**+8.72**	−1.50	−2.55
L4	95.15	+1.76	+2.16	**+4.00**	+0.08	+0.10
L10	94.47	+0.53	+1.11	**+4.24**	**+2.61**	**+4.31**
L11	96.10	+0.19	+0.61	**+2.91**	**+2.11**	**+3.70**
L19	95.49	+1.07	+1.22	**+3.64**	**+1.35**	**+2.46**

Table 4: **Per-layer plans run up to 43% faster at a matched floor.** Each row matches a uniform plan's worst-layer coverage.

Uniform plan	Floor	Uniform	Per-layer	Gain
g=8, c=4K	97.81	4.78×	6.81×	+42.5%
g=16, c=4K	**96.07**	**7.31×**	**10.47×**	**+43.3%**
g=32, c=4K	88.20	9.82×	13.23×	+34.8%
g=64, c=4K	83.51	12.02×	14.45×	+20.3%
g=64, c=2K	71.46	15.14×	15.14×	0.0%

Hardware-aware layer specialization. Choosing the group width and budget per layer turns the spread between layers into speed, since a uniform plan holds every layer to the width its least tolerant layer allows (O5). Letting the easy layers take wider groups runs the scoring stage over 40% faster at the floor that uniform $g = 16$ reaches with $c = 4096$ (Table 4). Task accuracy shows the same gain, as the plan of the main results matches the RULER accuracy of GroupIndex at $g = 16$ and $c = 2048$ while its scoring stage runs 21% faster (Figure 1b).

Appendix C describes the loss design, and Appendix E the remaining ablations, where max pooling falls below the mean at every width.

6 Conclusion

Within the widths we study, sharing one indexer scan across a query group is limited mainly by the surrogate gap rather than the candidate budget, since averaging the members' queries dilutes keys that individual members need. GroupIndex jointly learns head-wise query aggregation and scoring projections in a Surrogate Indexer that only proposes candidates to the native indexer, and applies hardware-aware layer specialization to select each layer's query-group width and candidate budget. On DeepSeek-V4-Flash, it matches the dense indexer's accuracy on RULER, LongBench-v2 and LongMemEval up to 512K tokens, and at 1M tokens it speeds up the indexer about 11× and end-to-end prefill up to 1.9×.

AI USE STATEMENT

In this work, we did not use generative AI tools for developing the conceptual framework or hypotheses, formulating mathematical claims or proofs, or designing the proposed method. We used generative AI tools, under the authors' supervision, to implement parts of the data-processing, serving and evaluation code, to suggest parts of the evaluation protocol, to draft interpretations of experimental results, and to translate draft text into English, while the core training and inference code of GroupIndex was written by the authors. The remaining required-disclosure tasks are not applicable to this work. Additionally, we used generative AI tools to assist with improving the readability and presentation of the manuscript. We reviewed and revised all AI-assisted text and code. The authors independently verified the technical descriptions, experimental settings, reported results, and claims in the paper. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies cross-query index sharing for accelerating the prefill of sparse-attention large language models. It does not involve human subjects, personal data collection, or the release of new datasets. Our experiments use a publicly available pretrained model, a public long-document corpus, and established evaluation benchmarks, subject to their respective licenses and terms of use. While faster prefill may facilitate broader deployment of long-context language models, GroupIndex keeps the model weights frozen and does not alter the underlying model capabilities, training data, or alignment properties. Consequently, potential misuse and societal risks remain those associated with the underlying models and their deployment contexts. We encourage users to apply the method in accordance with applicable laws, licenses, and responsible AI practices.

REPRODUCIBILITY STATEMENT

We provide a detailed description of GroupIndex in Sections 4 and 5, including the Surrogate Indexer, its training objective, the inference pipeline with in-candidate re-ranking, and hardware-aware layer specialization. The appendix contains pseudocode for training and inference, the loss design, the benchmark and baseline settings, the coverage protocol, the training hyperparameters and cost, and further empirical analyses. We will also release the source code and detailed scripts for reproducing the experiments upon acceptance, including environment setup, model and benchmark configurations, data collection from the ProLong corpus (Gao et al., 2025a), training settings, and evaluation commands. Together, these materials are intended to enable reproduction of the reported accuracy, coverage, and speedup results.

REFERENCES

Anthropic. System card: Claude Opus 4.7, April 2026. URL `https://www.anthropic.com/claude-opus-4-7-system-card`.

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 (Volume 1: Long Papers)*, pp. 3639–3664, Vienna, Austria, 2025. Association for Computational Linguistics.

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. In *Third Conference on Language Modeling*, 2026.

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

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, and Wen Xiao. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling. In *Second Conference on Language Modeling*, 2025.

Guanzhong Chen. VSPrefill: Vertical-slash sparse attention with lightweight indexing for long-context prefilling. *arXiv preprint arXiv:2603.04460*, 2026.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

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

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

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

Yaosheng Fu, Guangxuan Xiao, Xin Dong, Song Han, and Oreste Villa. SparDA: Sparse decoupled attention for efficient long-context LLM inference. *arXiv preprint arXiv:2606.04511*, 2026.

Tianyu Gao, Alexander Wettig, Howard Yen, and Danqi Chen. How to train long-context language models (effectively). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7376–7399, Vienna, Austria, 2025a. Association for Computational Linguistics.

Yizhao Gao, Zhichen Zeng, Dayou Du, Shijie Cao, Peiyuan Zhou, Jiaxing Qi, Junjie Lai, Hayden Kwok-Hay So, Ting Cao, Fan Yang, and Mao Yang. SeerAttention: Self-distilled attention gating for efficient long-context prefilling. In *Advances in Neural Information Processing Systems*, volume 38, pp. 55846–55869, 2025b.

GLM-5-Team, Aohan Zeng, Xin Lv, Zhenyu Hou, et al. GLM-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Google DeepMind. Gemini 3 Pro model card, November 2025. URL `https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf`. Last updated May 2026.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

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

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*, volume 37, pp. 52481–52515, 2024.

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*, pp. 963–989, 2025.

Chaofan Lin, Jiaming Tang, Shuo Yang, Hanshuo Wang, Tian Tang, Boyu Tian, Ion Stoica, Song Han, and Mingyu Gao. Twilight: Adaptive attention sparsity with hierarchical top-p pruning. In *Advances in Neural Information Processing Systems*, volume 38, pp. 119406–119432, 2025.

Hong Liu, Yuan Cheng, Lin Niu, Yi Su, Yufei Xue, Anmin Liu, Guanghua Yu, and Jianchen Zhu. PIVOT: Efficient query-group indexing for token-level sparse attention. *arXiv preprint arXiv:2607.24593*, 2026.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

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, Yutao Zhang, Zhilin Yang, Xinyu Zhou, Mingxing Zhang, and Jiezhong Qiu. MoBA: Mixture of block attention for long-context LLMs. In *Advances in Neural Information Processing Systems*, volume 38, pp. 17790–17815, 2025.

Lin Niu, Xin Luo, Linchuan Xie, Yifu Sun, Guanghua Yu, Jianchen Zhu, and S. Kevin Zhou. Stem: Rethinking causal information flow in sparse attention. In *Forty-third International Conference on Machine Learning*, 2026.

OpenAI. Introducing GPT-5.5, April 2026. URL `https://openai.com/index/introducing-gpt-5-5/`.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

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

Yan Wang, Qifan Zhang, Jiachen Yu, Tian Liang, Dongyang Ma, Xiang Hu, Zibo Lin, Chunyang Li, Zhichao Wang, Miao Peng, Nuo Chen, Jia Li, Yujiu Yang, Haitao Mi, and Dong Yu. FlashMemory-DeepSeek-V4: Lightning index ultra-long context via lookahead sparse attention. *arXiv preprint arXiv:2606.09079*, 2026a.

Yixuan Wang, Huang He, Siqi Bao, Hua Wu, Haifeng Wang, Qingfu Zhu, and Wanxiang Che. ProxyAttn: Guided sparse attention via representative heads. In *International Conference on Learning Representations*, pp. 18603–18617, 2026b.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations*, pp. 86809–86836, 2025.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, pp. 21875–21895, 2024.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. DuoAttention: Efficient long-context LLM inference with retrieval and streaming heads. In *International Conference on Learning Representations*, pp. 37228–37253, 2025.

Ruyi Xu, Guangxuan Xiao, Haofeng Huang, Junxian Guo, and Song Han. XAttention: Block sparse attention with antidiagonal scoring. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 69819–69831. PMLR, 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. In *Third Conference on Language Modeling*, 2026.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 23078–23097, Vienna, Austria, 2025. Association for Computational Linguistics.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big Bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems*, volume 33, pp. 17283–17297, 2020.

Jintao Zhang, Chendong Xiang, Haofeng Huang, Jia Wei, Haocheng Xi, Jun Zhu, and Jianfei Chen. SpargeAttention: Accurate and training-free sparse attention accelerating any model inference. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 76397–76413. PMLR, 2025.

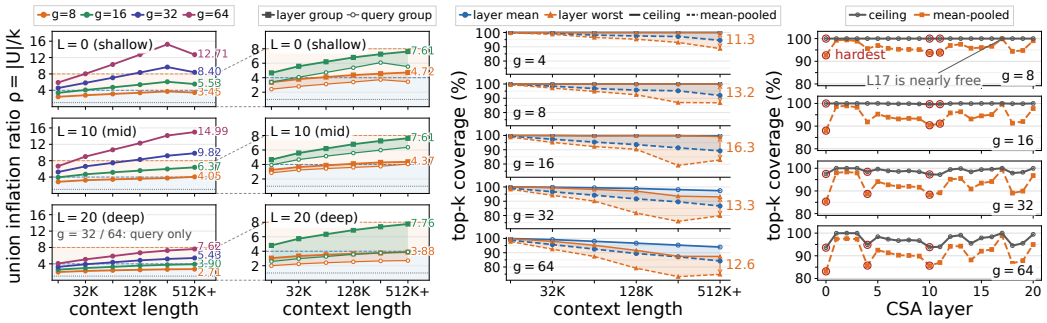
Yu Zhao, Zekun Zhang, Fan Jiang, Bo Zeng, Linlong Xu, Shimin Shan, Yu Liu, Longyue Wang, and Weihua Luo. LISA: Linear-indexed sparse attention for efficient long-context reasoning. *arXiv preprint arXiv:2607.19358*, 2026.

Ruijie Zhou, Fanxu Meng, Yufei Xu, Tongxuan Liu, Guangming Lu, Muhan Zhang, and Wenjie Pei. MISA: Mixture of indexer sparse attention for long-context LLM inference. *arXiv preprint arXiv:2605.07363*, 2026.

A MEASUREMENTS BEHIND THE PRELIMINARY STUDY

The capacity ceiling of Figure 2 is conservative. Its union oracle keeps the whole union when the budget holds it, and otherwise the c union keys with the highest native scores. Top-p coverage weights these keys unequally, so when the union overflows the budget a differently chosen set could score slightly higher, and the ceiling can only overstate the capacity loss and understate the surrogate gap. All coverage in the figure is top-p coverage at $c = 4096$ on the 600 held-out windows of Appendix D.2, and Figure 6 repeats it under top-k coverage.

Panel (a) shows the union inflation ratio in a shallow, a middle and a deep CSA layer, for query groups of $g = 8$ to 64 beside layer groups of the same size, which stop at $g = 16$ because the model has 21 CSA layers. Its dashed lines mark the largest unions that $c = 2048$ and $c = 4096$ hold. The two axes are orthogonal, so sharing across queries can still be combined with reuse across layers. Panel (c) covers the same widths over all length buckets, with per-layer values in Appendix E.6.



(a) Union inflation: query vs layer groups **(b) Surrogate gap vs length** **(c) Per-layer coverage**

Figure 6: **Figure 2 repeated with top-k coverage.** Panels (b) and (c) use top-k coverage in place of top-p coverage, and panel (a) is unchanged because the union inflation ratio does not depend on the coverage measure.

B TRAINING AND DEPLOYING THE SURROGATE INDEXER

B.1 TRAINING STARTS FROM THE TRAINING-FREE BASELINE

Training starts from the training-free baseline. The scoring projections copy the native weights and $a = b = 1/g$, so at step 0 the Surrogate Indexer is exactly the mean-pooled surrogate query of PIVOT, and every gain is a same-model improvement over it. Algorithm 1 trains one CSA layer with the initialization above, the pointwise BCE of Eq. equation 6 and the key roles of Appendix C.1, under the hyperparameters of Table 9.

B.2 DEPLOYMENT STAYS CAUSAL BEHIND THE NATIVE INTERFACE

The g positions of a group are causal and see prefixes of different lengths, so the shared candidate set keeps only keys in the prefix that every member sees. The newest keys, visible only to the later members, form a near window $\mathcal{W}_j$ of about $g/4$ keys that each position receives without competition, cut at its own causal limit,

$$\mathcal{C}_j = \underbrace{A = \text{top-}c\,(\hat{s})}_{\text{shared, competitive}} \cup \underbrace{\mathcal{W}_j}_{\text{near window}} . \tag{8}$$

The window keeps the scan causal, since the surrogate never ranks a key that some member cannot see. It also costs little, since a single surrogate query has little basis to rank the newest keys and they are few.

Algorithm 2 puts these steps together for one CSA layer during prefill and hands the selected sets to the Sparse MLA through the native DSA interface. A layer whose prefix holds at most $2c$ keys runs the native indexer directly, since its top-c would already cover at least half of the prefix and leave

Algorithm 1 Training the Surrogate Indexer of one CSA layer.

Require: windows cached once from the frozen backbone, each a group G of g consecutive positions with hidden states h_j, the compressed keys of its prefix and the native top-$2k$ keys of every position
budgets $c \in \{2048, 4096\}$ and token budget k
Ensure: W_q, W_w, a and b of the layer, with one checkpoint per budget c
copy the native indexer weights into W_q, W_w and set $a = b = 1/g$ ▷ baseline-anchored initialization
for 270K steps, five epochs over batches of 8 windows **do**
  for all windows G in the batch **do**
    $\mathcal{P} \leftarrow U = \bigcup_{j \in G} \mathcal{T}_j$ ▷ incumbents
    $\mathcal{B} \leftarrow$ keys ranked $k{+}1$ to $2k$ by some $j \in G$, outside U ▷ borderline, exempted
    $R \leftarrow$ 4,096 random keys outside $U \cup \mathcal{B}$ ▷ mining pool
    compute $\hat{s}_n$ for $n \in \mathcal{P} \cup R$ with the current parameters ▷ §4.1
    $\mathcal{N} \leftarrow$ top 512 of R by $\hat{s}$ ▷ contenders, mined online
    $\ell_G \leftarrow \mathrm{mean}_{n \in \mathcal{P} \cup \mathcal{N}} \mathrm{BCE}\big(\hat{s}_n, \mathbf{1}[n \in U]\big)$ ▷ pointwise BCE, Eq. equation 6
  end for
  take an AdamW (Loshchilov & Hutter, 2019) step without weight decay on ℓ_G averaged over the batch, after clipping the layer's gradient norm to 1, with learning rates 10^{-4} for W_q, W_w and 10^{-2} for a, b
  if the step is a multiple of 2,000 **then**
    for each c, save a checkpoint when its top-c holds a larger share of the union on 600 validation windows than at any earlier check
  end if
end for

little to save. The aggregation adds only $2H^I g$ parameters per layer, so nearly all of the Surrogate Indexer's parameters sit in the scoring projections.

Algorithm 2 GroupIndex inference in one CSA layer during prefill.

Require: hidden states h_j, native indexer queries $\mathbf{q}^I_{j,h}$ and weights $w^I_{j,h}$ of the positions j in a prefill chunk
the N compressed keys $\mathbf{k}^{I\mathrm{Comp}}_n$ of the prefix
group width g and candidate budget c of the layer, fixed or from Eq. equation 7, and token budget k
Ensure: the selected set $\widehat{\mathcal{T}}_j$ of k keys for every position j, which equals $\mathcal{T}_j$ whenever $\mathcal{C}_j$ contains it
if $N \le 2c$ **then** ▷ short prefix
  return the native top-k set of every position, Eq. equation 1
end if
for all groups G of g consecutive positions, aligned to multiples of g **do**
Surrogate scan, once per group
  $q_{j,h} \leftarrow (W_q h_j)_h,\ w_{j,h} \leftarrow (W_w h_j)_h$ for $j \in G$ ▷ scoring projections
  $\bar{q}_h \leftarrow \sum_{j \in G} a_{hj}\, q_{j,h},\ \bar{w}_h \leftarrow \sum_{j \in G} b_{hj}\, w_{j,h}$ ▷ Eq. equation 5
  $\hat{s}_n \leftarrow \sum_h \bar{w}_h\, \mathrm{ReLU}(\bar{q}_h \cdot \mathbf{k}^{I\mathrm{Comp}}_n)$ for $n = 1, \ldots, N$ ▷ one pass over the prefix
  $A \leftarrow$ Top-$c(\hat{s})$, keeping only keys before the near window of G
In-candidate re-ranking, per position
  for all $j \in G$ **do**
    $\mathcal{C}_j \leftarrow A \cup \mathcal{W}_j$ ▷ Eq. equation 8
    $\widehat{\mathcal{T}}_j \leftarrow$ Top-$k\big(\{I_{j,n}\}_{n \in \mathcal{C}_j}\big)$ ▷ native score, Eq. equation 1
  end for
end for
return $\{\widehat{\mathcal{T}}_j\}$ to the Sparse MLA through the native DSA interface

C LOSS DESIGN

GroupIndex trains with the pointwise BCE of Eq. equation 6, and along the way we tried several auxiliary losses and alternative training objectives. Two refinements act on the keys near the cut, where the union is won or lost, and full-field distillation replaces or supplements the membership labels with the dense indexer's score field. Neither refinement improves coverage by as much as a point, and a pairwise loss trained on its own collapses, so the training signal comes from the membership labels.

C.1 MEMBERSHIP LABELS CARRY THE SIGNAL

The candidate set is taken by rank, not by threshold, so whether the union fits depends only on the order of the few keys that compete near the cut. Matching the native score field would spend capacity on the order of keys far from the cut, so the pointwise BCE of Eq. equation 6 supervises only membership in the union U, through three roles for the sampled keys. Incumbents $\mathcal{P} = U$ are the positives, the keys some position selects natively, contenders $\mathcal{N}$ are the non-union keys that the current Surrogate Indexer scores highest and serve as the negatives, and borderline keys $\mathcal{B}$ fall just past a position's top-k, at native ranks $k{+}1$ to $2k$, and stay out of the loss.

Membership labels carry the training signal on their own. The pointwise BCE alone lifts coverage over step 0 at every layer, whereas the pairwise term trained without it falls below step 0 with either source of negatives, by 42 to 56 points of top-p coverage when they are mined (Table 5). An ordering constraint leaves the scale of the scores free, and the collapse is consistent with the pointwise term anchoring every score to one absolute scale. The refinements of Appendix C.2 build on this anchor and add little to it.

Table 5: **The reported runs use only the pointwise BCE; the pairwise term alone collapses.** Top-p gain over step 0 at g=16 and c=4096, top-k in parentheses (Appendix D.2). The last row is the change from that BCE baseline when borderline keys count as negatives.

Negatives	Loss	L0	L10	L19
	Step 0	89.54 (88.02)	94.47 (90.43)	95.49 (92.03)
Mined	Pointwise only	+8.70 (+9.58)	+4.22 (+6.56)	+3.64 (+6.00)
	Pairwise only	−46.40 (−52.68)	−42.08 (−55.01)	−55.54 (−68.17)
Random	Pointwise only	+8.68 (+9.54)	+4.21 (+6.53)	+3.64 (+6.00)
	Pairwise only	−43.68 (−50.19)	−11.52 (−16.32)	−5.58 (−9.59)
Borderline as negatives (Δ)		+0.01 (−0.02)	−0.04 (−0.09)	−0.00 (−0.04)

C.2 REFINEMENTS NEAR THE CUT ARE NEARLY NEUTRAL

The first refinement is online hard-negative mining. A non-union key drawn at random usually scores far below the cut and teaches the surrogate little about the keys that actually compete with the union, so the contenders are the keys that the current Surrogate Indexer scores highest within a random pool (Table 9). With the pointwise BCE, mined negatives and the whole random pool reach the same coverage (Table 5).

The second is borderline exemption. A key just past a position's top-k can carry the same native score as the last key selected, and at the training budget the band fits beside the union, so labeling it negative would teach a separation that the native indexer does not make. Counting the band as negatives instead barely moves coverage, as the last row of Table 5 shows.

At the reported budget both refinements are nearly neutral. Each changes top-p coverage by at most 0.04 points at $c = 4096$, close to the difference of at most 0.01 between two runs of the same configuration. Both stay far below the joint-training gains of Table 3, so they are safe choices rather than sources of the gain.

Mining matters more as the budget tightens. At $c = 2048$ it changes top-p coverage by up to 0.19 points, above the difference of at most 0.04 between two runs at that budget, and it gains more again

in contexts beyond 256K, while borderline exemption stays near that noise. We read this as the cut mattering more when it is crowded, since the contenders that mining selects then compete for fewer slots. The pattern fits its rationale, but the gain stays within a few tenths of a point at either budget.

C.3 Full-field distillation matches at a higher cost

Full-field distillation is the natural alternative to membership labels, since it passes the surrogate the dense indexer's graded scores rather than a binary label per key. It distills (Hinton et al., 2015) that score field into the surrogate by regressing the teacher's normalized scores over every visible key, more than 100K per window, whereas the losses above score about 7K sampled keys per window. Under matched conditions, with the same surrogate parameterization, validation windows, steps, batch size and learning rate, it differs by -0.55 to $+0.33$ points of top-k coverage across three layers from a membership-label arm that adds a pairwise hinge over incumbent and contender pairs to the BCE, within the paired 95% interval (Table 6). The distillation variant trains on far fewer documents, which does not change coverage at this step budget.

Table 6: **Under matched conditions, full-field distillation and BCE + pair are within noise.** Top-k coverage (%), where Δ is BCE + pair minus distillation and the paired 95% interval is ± 1.06.

Layer	Step 0	BCE + pair	Full-field distillation	Δ
L10	92.17	97.27	96.98	+0.29
L11	91.83	97.13	96.58	+0.55
L14	93.23	96.94	97.27	−0.33

Distillation is also costlier and more sensitive to the learning rate. Scoring every visible key makes it train about an order of magnitude slower, and at the lower learning rate it gains 0.59 points on L10 against 1.28 for BCE + pair (Table 7). That comparison is shorter, 3,000 steps evaluated on 32 windows of one held-out document with different training documents for the two variants, and at the higher learning rate the two stay close.

Table 7: **At the lower learning rate, distillation falls behind on L10.** Top-k coverage (%), with the gain over step 0 in parentheses.

Loss	Learning rate	L0	L10
Step 0	–	84.56	96.47
BCE + pair	10^{-5}	95.00 (+10.43)	97.75 (+1.28)
	10^{-4}	96.31 (+11.75)	98.06 (+1.59)
Full-field distillation	10^{-5}	94.99 (+10.43)	97.07 (+0.59)
	10^{-4}	96.41 (+11.86)	98.30 (+1.81)

As an auxiliary term, distillation helps little. Added on top of that arm it gains less than a point of coverage, and the dense signal falls mostly on keys far from the cut, so it buys no coverage that membership labels miss. The reported runs train without distillation.

D Experimental protocol

D.1 Benchmarks and baselines

Run-to-run noise decides what each benchmark can separate (Table 8). Only RULER resolves differences among methods that stay close to dense accuracy, and its threshold is calibrated at 512K rather than 64K. On the other two no surrogate-based method falls below the dense indexer by more than the noise, so they show only that accuracy is kept. Lengths in the headers of Table 1 are average prompt lengths after truncation, with 1K = 1024 tokens.

The baselines of §5.1 need three notes. Our IndexCache reimplementation is training-free, g counts layers there, and its Reuse variant runs at $c = 512$ with a speedup capped at 21/6, since only 6 of

Table 8: **Run-to-run noise decides what each benchmark can separate.** Settings of the three benchmarks, with the noise measured by repeated runs.

Benchmark	Content	Context and samples	Run-to-run noise
RULER	13 tasks	64K to 512K tokens, 100 samples per task	lossless if the average moves by at most 0.2 at 512K
LongBench-v2	short, medium and long by word count	middle-truncated to 512K tokens, 47 of the 108 long prompts shortened	about 2 points per length bucket
LongMemEval	S with the full history, M with the most recent sessions	within a 512K budget, accuracy over 500 questions	$\pm$2.7 points

the 21 indexer layers score. PIVOT-Refine keeps the mean-pooled surrogate, so its coverage is that of step 0. Random and Recency list the speedup of the training-free $g = 4$ pipeline whose candidate source they replace, and their own cost is lower.

D.2 Protocol shared by the coverage results

Every coverage number follows one protocol unless its caption says otherwise. Coverage is top-p coverage at $c = 4096$, with top-k in parentheses, averaged over the 21 indexer layers and measured on 600 held-out windows, 100 in each of six length buckets that double from 16K to beyond 512K tokens. Results are paired means within a group width, since $g = 32$ and $g = 64$ share a collection of validation documents while $g = 8$ and $g = 16$ each have their own, and two runs of the same configuration differ by at most 0.01 points of top-p and 0.02 of top-k coverage. Table 4 is the exception, its floors taken on 16 windows per bucket and on the harder of the two deepest buckets, so they are not comparable with the worst-layer numbers of Table 12 and Figure 1a.

Step 0 is the Surrogate Indexer before training (§5.1), and every gain is measured from it. The worst layer is the one where GroupIndex scores lowest. Coverage is not compared across budgets, since the evaluation keeps only windows with more than c visible keys, and each ablation changes one component of the reported configuration at $g = 16$ and $c = 4096$, leaving the rest at its initialization.

D.3 Training configuration and cost

Training reads only the cached data and needs little hardware. Each of the 21 Surrogate Indexers trains on its own within about 3 GB of GPU memory under the hyperparameters of Table 9, and one trajectory yields a checkpoint for each of $c = 2048$ and $c = 4096$, since the budget enters the choice of checkpoint and not the gradient. Layer specialization trains nothing further. Collecting the data costs about six times as much as training all four widths (Table 10). Each collection caches hidden states, compressed keys and native indexer scores for 1,000 long documents, 900 for training and 100 for validation, and every ablation reuses them.

E Supplementary results

Section 5 makes five statements, and the results below back them in turn. Accuracy depends mainly on the candidate set, end-to-end gains concentrate in long contexts, the width limit lies in the representation, the two parts of the Surrogate Indexer help mainly together, and layer specialization spends slack that a uniform plan leaves unused. Per-layer coverage for every setting closes the appendix as reference data (Appendix E.6).

E.1 Accuracy depends mainly on the candidates

The accuracy of GroupIndex depends mainly on the candidates it proposes rather than on the re-ranking inside them. With in-candidate re-ranking fixed and only the source of candidates changed (Table 11), learning adds 7.45 RULER points over mean pooling at $g = 16$, and reading content

Table 9: **One configuration trains every layer, width and budget.** Training hyperparameters of the reported configuration, with k=512.

Setting	Value
Initialization	native scoring projections, $a = b = 1/g$
Optimizer	AdamW per layer, no weight decay, gradient norm clipped to 1
Learning rate	10^{-4} for the scoring projections, 10^{-2} for a and b
Schedule	batches of 8 windows, 270K steps (five epochs)
Positives	the union U of the native top-k sets
Negatives	the 512 top-scored keys of a random pool of 4,096 keys outside U and $\mathcal{B}$
Borderline band	native ranks k+1 to 2k, left out of the loss
Evaluation	every 2,000 steps on the 600 validation windows
Checkpoints	one per budget $c \in \{2048, 4096\}$, the step whose top-c holds the largest share of U

Table 10: **Collecting the data costs about six times as much as training all four widths.** GPU-hours on H20 GPUs, where g=32 and g=64 share one collection, counted once. Training is given as single-GPU hours over all 270K steps and the periodic evaluation.

Stage	Width	GPU-hours	Data (TB)
Collection	$g = 8$	371	1.65
	$g = 16$	333	2.91
	$g = 32$	445	10.51
	$g = 64$		
Training	$g = 8$	30.6	–
	$g = 16$	36.9	–
	$g = 32$	50.1	–
	$g = 64$	62.0	–
Total		**1,328**	**15.07**

at all, even without training, is worth about 31.6 points over random candidates. Random picks candidates uniformly and Recency takes the keys nearest the causal boundary, yet Recency is the weakest source, below Random at every length, which argues against the view that answers mostly sit near the causal boundary. The re-ranker can only keep what the candidates contain, so the source largely decides the accuracy.

The comparison favors the content-free sources, so their gaps are lower bounds. They always fill every candidate slot, whereas the surrogate-based methods keep 918 to 2046 of the 2048 candidates after the causal cut. More candidates chosen blindly still lose to fewer chosen by reading content, so the gap measures the value of the content rather than of the budget.

E.2 End-to-end gains concentrate in long contexts

The end-to-end gain of GroupIndex concentrates in long contexts because it rests on two quantities that both grow with context. GroupIndex speeds up only the indexer's scoring stage and leaves attention and the MoE layers as they are, so the share of prefill time that the scoring stage takes caps the end-to-end gain. That share grows with the history each chunk is scored against, and the kernel speedup grows with the history as well (Figure 5a). Together they keep the gain below 1.3× up to 256K and lift it to 1.89× at 1M for $g = 16$ (Figure 7). Short prompts also gain little because their first chunks have prefixes of at most $2c$ keys, which run the native indexer directly (Appendix B.2). GroupIndex therefore pays off in long-context prefill, which is also where prefill takes longest.

The layer-specialized plan of Figure 5b stays within a few hundredths of $g = 16$ at every length and reaches 1.92× at 1M, so the figure stands for both. Each speedup is a ratio of latencies for requests

Table 11: **Candidates chosen without reading content trail GroupIndex by more than 35 RULER points.** Only the candidate source changes, at c=2048, and the content-free sources were not evaluated offline. The PIVOT-Refine and GroupIndex rows repeat Table 2.

Candidate source	**RULER**: AVG (512K)	**Coverage**: top-p (top-k)	**Speedup** @ 512K
DSA (baseline)	92.68 (88.3)	100 (100)	1.00×
g=8			
GroupIndex	**92.48 (88.2)**	**98.53 (96.78)**	5.65×
PIVOT-Refine	89.85 (83.0)	95.19 (91.60)	6.75×
Random	55.75 (43.7)	–	6.75×
Recency	48.95 (41.6)	–	6.75×
g=16			
GroupIndex	**92.38 (88.2)**	**96.74 (93.18)**	8.80×
PIVOT-Refine	84.93 (75.6)	93.04 (88.16)	11.71×
Random	53.35 (39.3)	–	11.71×
Recency	48.15 (36.8)	–	11.71×

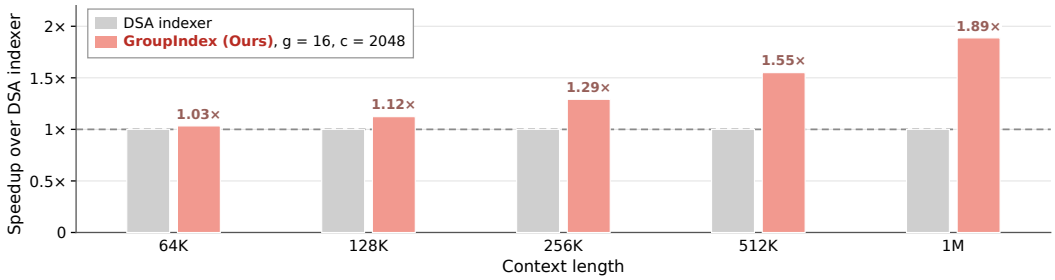


Figure 7: **End-to-end prefill gains concentrate in long contexts.** Measured speedup of the whole prefill over the DSA indexer for GroupIndex at g=16 and c=2048.

that generate a single token, whose time is almost all prefill. Latency is measured at the server on the eight H20 GPUs of Section 5.1 and taken as the median of five runs, or three at 1M.

E.3 The width limit lies in the representation

What wider groups need is a better surrogate query, not more candidates. Averaged coverage understates this, since top-p coverage separates the two surrogates of Table 2 by only about 3 points at every g, yet the keys the mean-pooled surrogate misses are the ones that multi-key retrieval depends on, so at 512K most of the RULER gap sits in the multi-key tasks. The gap also survives at equal speed, as GroupIndex at $g = 32$ runs slightly faster than PIVOT-Refine at $g = 16$ and scores 12.1 points higher at 512K (Figure 1b). The gain from learning is therefore a gain in what the surrogate retrieves, which a faster setting of the mean-pooled surrogate does not match.

The budget, by contrast, is a weak lever for both quality and speed. Doubling g accelerates the scoring stage by up to 1.56×, against at most 1.26× for halving c, and at equal speed a smaller g with a smaller budget is the more accurate choice, as $g = 32$ with $c = 2048$ beats $g = 64$ with $c = 4096$ at 512K (Table 2). Group width is therefore the knob for speed, and a larger budget is worth its cost only where the union no longer fits (Section 5.4).

No fixed pooling rule closes the gap. Because the query projection is linear up to RMSNorm, pooling the hidden states or averaging the queries both amount to mixing the members' own queries at preset weights, and the union asks for a maximum over members, so max pooling is the natural fixed rule to try in place of the mean. It does worse at every group width, and the shortfall grows with g (Table 12), with the same ordering when coverage is averaged over layers and at $c = 2048$.

A coordinate-wise maximum assembles one query from different members' extreme coordinates, which is not a maximum over members, so it does not approximate the score the union needs. Learning is what closes the gap, and at $g = 64$ GroupIndex closes 61% of the gap between the mean and the capacity ceiling. Both pooling rules keep the native weights, and the query variant averages the head weights, since max-pooling those as well scores lower still.

Table 12: **Max pooling is worse than the mean.** Worst-layer top-p coverage, top-k in parentheses, in the 256K–512K bucket at c=4096, where the worst layer is L0 in every column.

Surrogate	g=8	g=16	g=32	g=64
Capacity ceiling	100.00 (100.00)	99.98 (99.97)	95.40 (93.54)	90.01 (87.52)
GroupIndex	99.02 (98.69)	96.07 (94.77)	90.81 (88.42)	84.23 (81.77)
Mean pooling (PIVOT-Refine)	88.53 (86.98)	80.95 (79.00)	77.87 (75.91)	75.08 (73.25)
Max pooling, hidden states	85.52 (83.56)	77.05 (74.75)	72.00 (69.51)	68.30 (65.86)
Max pooling, queries	79.47 (76.89)	65.80 (62.17)	56.18 (52.10)	46.41 (42.50)

E.4 The two parts help mainly together

The head-wise query aggregation and the scoring projections pay off mainly together, as Section 4.1 argues. Their interaction, the joint gain minus both single-part gains, measures how much of the joint gain neither part delivers alone. Under top-k coverage it reaches 4.62 points at L10 and turns negative at L0, at -1.17, the same pattern as under top-p coverage (Table 3). Deep layers thus need both parts, whereas the shallowest layer does not.

The interaction grows with context in the deep layers, while in the shallowest one it stays negative and falls further (Table 13). Beyond 256K the joint gain at L10 reaches 7.03 points against at most 1.85 for either part alone, whereas at L0, where neighboring queries differ mostly in lexical and positional cues that reweighting can capture, either part alone recovers more than half of the joint gain. Training both parts matters most in deep layers at long context, and a recipe that trains only one part would lose most of the gain exactly there.

Table 13: **Beyond 256K the interaction grows in every deep layer.** Top-p gain over step 0 at g=16 and c=4096, over all 600 windows as in Table 3 and over the 200 windows beyond 256K. Inter. = Both − Agg. only − Proj. only.

	Shallow		**Deep**		
	L0	L4	L10	L11	L19
All windows					
Agg. only	+5.43	+1.76	+0.53	+0.19	+1.07
Proj. only	+4.79	+2.16	+1.11	+0.61	+1.22
Both	+8.72	+4.00	+4.24	+2.91	+3.64
Inter.	−1.50	+0.08	+2.61	+2.11	+1.35
Beyond 256K					
Agg. only	+9.46	+2.92	+0.88	+0.20	+1.84
Proj. only	+8.54	+3.62	+1.85	+1.01	+2.03
Both	+15.45	+6.63	+7.03	+4.90	+6.34
Inter.	−2.55	+0.10	**+4.31**	**+3.70**	**+2.46**

E.5 Layer specialization spends unused slack

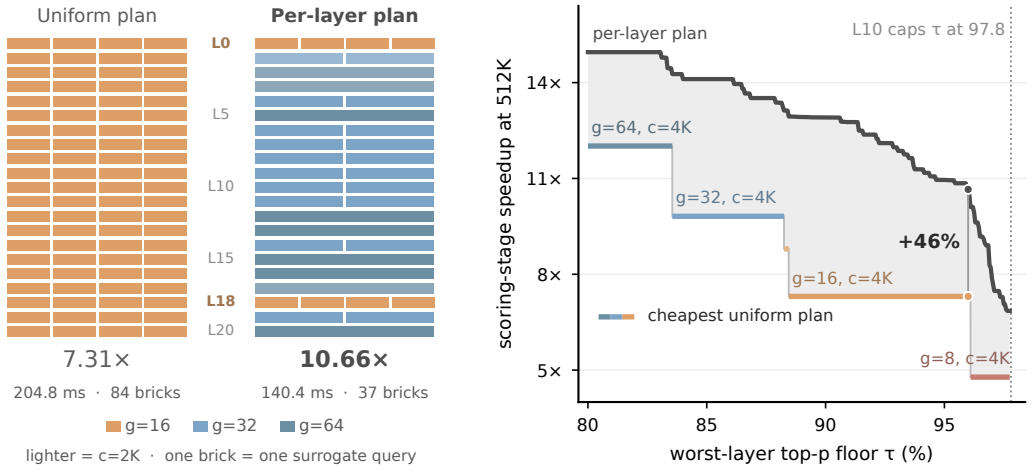
Layer specialization gains by spending the coverage that a uniform plan leaves unused on its easy layers. The per-layer plan of Table 4 takes each uniform plan's worst-layer coverage as the floor and solves the cheapest per-layer plan (g_ℓ, c_ℓ) that meets it, choosing each layer's setting from $g \in \{8, 16, 32, 64\}$ and $c \in \{2048, 4096\}$. Coverage there is top-p coverage on 16 windows per

bucket, taking the harder of the two deepest length buckets. That floor is not the 100-window, 256K–512K single-bucket worst-layer coverage of Table 12 and Figure 1a. Both speedup columns are scoring-stage speedups over the dense indexer at 512K, obtained by summing per-layer kernel latency profiled on an H20 rather than by timing the whole chain. The uniform rows use $c = 4096$, unlike the fixed-g configurations of Table 1, and the plan in Table 1 uses the round floor of 96.00% and reaches 10.66× on the same per-layer cost table. Table 14 repeats the comparison at $c = 2048$, the budget of those fixed-g configurations, and there the gain grows with the floor that the per-layer plan must meet.

Table 14: **With uniform plans at c=2K, the gain of per-layer plans grows with the floor they must meet.** Same layout as Table 4, and the g=64 row repeats its last row.

Uniform plan	Floor	Uniform	Per-layer	Gain
g=8, c=2K	92.66	5.65×	12.11×	+114.2%
g=16, c=2K	**88.43**	**8.79×**	**13.13×**	**+49.3%**
g=32, c=2K	77.75	12.14×	14.96×	+23.3%
g=64, c=2K	71.46	15.14×	15.14×	0.0%

The per-layer plan is never slower than the cheapest uniform plan that meets the same floor, at any floor (Figure 8), and no plan of either kind exists above 97.81%, the best coverage any setting reaches on L10 and the floor of the top row of Table 4. At the lowest floor every layer can take the cheapest setting, so the per-layer plan collapses onto the uniform one, as the row below the rule of Table 4 confirms. The slack it spends is the spread between layers of Appendix E.6, and at $\tau = 96\%$ the plan gives the hardest layer, L0, $g = 16$ and the easiest, L17, $g = 64$.



(a) Per-layer vs. uniform at τ = 96% **(b) Same floor, more speed**

Figure 8: **Per-layer plans stay faster than uniform ones at every floor.** Each row in (a) is one indexer layer and each brick one surrogate query per 64 positions. Speedups are for the scoring stage on an H20 at 512K.

E.6 Per-layer coverage

Per-layer coverage backs the layer-level statements above, and the mean rows of the width tables reproduce the averages of Table 2. Tables 15 and 16 give top-p and top-k coverage at the four widths, averaged over all six length buckets, and Tables 17 and 18 split the $g = 16$ columns into the four buckets from 64K upward, which cover the lengths of the RULER results. Each cell reads step 0 → checkpoint followed by the gain, with the checkpoint saved for that budget, under the shared protocol of Appendix D.2.

Difficulty is a property of the layer. The layers keep nearly the same order at every width and under both measures, with Spearman correlations of at least 0.89 between any two widths, and the lowest layer is L0 in seven of the eight top-p columns and L10 or L11 in seven of the eight top-k columns. This is the stability that O5 reads from Figure 2c, here for every width, both budgets and both measures, and these hard layers set the worst-layer floors that layer specialization must meet (Appendix E.5).

Training lifts every layer and gains most where step 0 is lowest, as the bold gains mark. The spread of top-p coverage across layers thus narrows from 10.4 points at step 0 to 1.7 at $g = 16$ and $c = 4096$, yet the spread that remains grows about fivefold from $g = 16$ to $g = 64$ at that budget. Wider groups leave more room between the easy and the hard layers, and that room is what the per-layer plan spends.

Coverage falls with context length, and training gains grow with it. The mean rows fall from bucket to bucket at both budgets, top-k faster than top-p, and at $c = 2048$ every layer except L0 has its lowest checkpoint top-p coverage in the longest bucket. Training gains most where the mean-pooled surrogate loses most, as the mean top-p coverage at $c = 4096$ rises from 93.3 to 98.5 in the longest bucket, more than twice the gain of the 64K–128K bucket. The hard layers stay hard at every length, since at $c = 2048$ L0 is the lowest layer of every bucket under top-p coverage and L0 or L11 under top-k.

Table 15: **Training lifts every layer, and L0, the hardest, gains the most at every width.** Top-p coverage (%) per layer, step 0 $\rightarrow$ checkpoint and the gain, on the 600 validation windows of each width. Bold marks gains of at least 5 points.

Layer	g=8	g=16	g=32	g=64
c=2048				
L0	88.7→97.0 **+8.3**	82.9→92.3 **+9.4**	79.9→87.7 **+7.8**	77.5→82.7 **+5.2**
L1	97.8→99.7 +1.9	96.7→99.1 +2.4	95.6→98.1 +2.5	94.6→97.0 +2.4
L2	98.4→99.8 +1.4	97.5→99.3 +1.8	96.7→98.6 +1.9	95.9→97.6 +1.7
L3	97.9→99.7 +1.8	97.1→99.1 +2.0	96.7→98.5 +1.8	96.1→97.8 +1.7
L4	94.1→98.1 +4.0	90.1→95.1 **+5.0**	87.1→91.9 +4.8	84.3→88.6 +4.3
L5	95.9→98.8 +2.9	94.1→97.2 +3.1	93.1→95.9 +2.8	91.7→94.1 +2.4
L6	94.4→97.9 +3.5	92.5→96.1 +3.6	91.4→94.7 +3.3	90.0→92.7 +2.7
L7	93.8→97.9 +4.1	91.6→95.8 +4.2	90.3→94.0 +3.7	88.7→91.7 +3.0
L8	93.5→98.0 +4.5	91.2→95.8 +4.6	89.7→93.8 +4.1	87.8→91.4 +3.6
L9	94.4→98.2 +3.8	92.5→96.5 +4.0	91.0→94.9 +3.9	89.4→92.4 +3.0
L10	92.5→97.4 +4.9	89.3→94.6 **+5.3**	87.0→91.9 +4.9	84.5→87.6 +3.1
L11	94.4→97.8 +3.4	92.1→95.6 +3.5	90.6→93.6 +3.0	89.0→90.7 +1.7
L12	96.4→99.1 +2.7	95.1→97.9 +2.8	94.6→96.9 +2.3	93.6→95.5 +1.9
L13	96.8→99.3 +2.5	95.5→98.3 +2.8	94.7→97.2 +2.5	93.7→95.9 +2.2
L14	93.6→97.5 +3.9	90.1→94.6 +4.5	87.2→91.5 +4.3	84.3→87.8 +3.5
L15	95.4→98.6 +3.2	93.9→97.2 +3.3	92.9→95.9 +3.0	91.6→94.1 +2.5
L16	96.9→99.2 +2.3	95.6→98.1 +2.5	94.8→97.2 +2.4	93.9→95.7 +1.8
L17	99.7→100.0 +0.3	99.4→99.9 +0.5	99.1→99.8 +0.7	98.8→99.6 +0.8
L18	93.2→97.7 +4.5	90.1→95.1 **+5.0**	88.0→92.6 +4.6	85.8→89.4 +3.6
L19	93.6→98.0 +4.4	90.8→95.5 +4.7	89.1→93.3 +4.2	87.3→90.7 +3.4
L20	97.7→99.6 +1.9	95.9→98.4 +2.5	94.8→97.1 +2.3	93.4→95.4 +2.0
Mean	95.2→98.5 +3.3	93.0→96.7 +3.7	91.6→95.0 +3.4	90.1→92.8 +2.7
c=4096				
L0	93.8→99.7 **+5.9**	89.5→98.3 **+8.8**	87.0→95.7 **+8.7**	85.0→91.6 **+6.6**
L1	99.4→100.0 +0.6	98.9→100.0 +1.1	98.4→99.8 +1.4	97.9→99.5 +1.6
L2	99.6→100.0 +0.4	99.3→100.0 +0.7	99.0→99.9 +0.9	98.6→99.7 +1.1
L3	99.3→100.0 +0.7	99.0→100.0 +1.0	98.8→99.8 +1.0	98.5→99.6 +1.1
L4	97.5→99.8 +2.3	95.2→99.2 +4.0	93.2→97.9 +4.7	91.1→95.5 +4.4
L5	98.3→99.9 +1.6	97.4→99.6 +2.2	96.8→99.2 +2.4	96.0→98.2 +2.2
L6	97.5→99.7 +2.2	96.5→99.2 +2.7	95.8→98.6 +2.8	95.0→97.3 +2.3
L7	97.1→99.7 +2.6	95.9→99.2 +3.3	95.1→98.3 +3.2	94.1→96.9 +2.8
L8	96.9→99.8 +2.9	95.7→99.3 +3.6	94.8→98.4 +3.6	93.6→97.0 +3.4
L9	97.2→99.8 +2.6	96.1→99.3 +3.2	95.2→98.5 +3.3	94.2→97.0 +2.8
L10	96.4→99.6 +3.2	94.5→98.7 +4.2	93.1→97.5 +4.4	91.4→94.4 +3.0
L11	97.4→99.7 +2.3	96.1→99.0 +2.9	95.2→98.1 +2.9	94.1→96.0 +1.9
L12	98.3→99.9 +1.6	97.7→99.7 +2.0	97.4→99.3 +1.9	96.9→98.6 +1.7
L13	98.6→100.0 +1.4	98.0→99.8 +1.8	97.6→99.5 +1.9	96.9→98.8 +1.9
L14	97.3→99.7 +2.4	95.5→99.0 +3.5	93.9→97.8 +3.9	91.9→95.4 +3.5
L15	97.8→99.8 +2.0	97.0→99.5 +2.5	96.4→99.0 +2.6	95.7→98.0 +2.3
L16	98.5→99.9 +1.4	97.8→99.7 +1.9	97.4→99.4 +2.0	96.8→98.5 +1.7
L17	99.9→100.0 +0.1	99.9→100.0 +0.1	99.8→100.0 +0.2	99.7→100.0 +0.3
L18	96.9→99.7 +2.8	95.1→99.0 +3.9	93.7→97.9 +4.2	92.1→95.7 +3.6
L19	97.1→99.8 +2.7	95.5→99.1 +3.6	94.4→98.2 +3.8	93.2→96.4 +3.2
L20	99.4→100.0 +0.6	98.5→99.9 +1.4	97.9→99.6 +1.7	97.1→99.0 +1.9
Mean	97.8→99.8 +2.0	96.6→99.4 +2.8	95.7→98.7 +3.0	94.8→97.3 +2.5

Table 16: **Top-k coverage orders the layers almost as top-p does, with L10 and L11 lowest.** Same layout and windows as Table 15, with top-k coverage (%). Bold marks gains of at least 7 points.

Layer	g=8	g=16	g=32	g=64
c=2048				
L0	86.4→95.7 **+9.3**	80.2→89.4 **+9.2**	77.1→83.8 +6.7	75.0→79.0 +4.0
L1	97.2→99.7 +2.5	95.8→98.9 +3.1	94.5→97.5 +3.0	93.4→96.1 +2.7
L2	97.8→99.8 +2.0	96.6→99.0 +2.4	95.5→97.9 +2.4	94.4→96.6 +2.2
L3	96.5→99.5 +3.0	95.2→98.4 +3.2	94.5→97.2 +2.7	93.5→96.0 +2.5
L4	89.5→96.1 +6.6	83.1→90.2 **+7.1**	78.6→84.2 +5.6	74.7→79.4 +4.7
L5	92.3→97.4 +5.1	89.1→94.1 +5.0	87.3→91.4 +4.1	84.9→88.4 +3.5
L6	90.1→95.6 +5.5	87.0→91.9 +4.9	85.0→89.1 +4.1	82.6→86.0 +3.4
L7	89.4→95.5 +6.1	85.8→91.3 +5.5	83.6→87.9 +4.3	81.0→84.4 +3.4
L8	89.7→96.1 +6.4	86.2→92.1 +5.9	83.8→88.8 +5.0	80.9→85.0 +4.1
L9	89.9→95.8 +5.9	86.5→91.9 +5.4	83.7→88.6 +4.9	80.8→84.4 +3.6
L10	86.6→93.6 **+7.0**	81.0→87.3 +6.3	77.7→82.6 +4.9	74.0→77.1 +3.1
L11	86.5→93.2 +6.7	81.7→87.1 +5.4	78.8→82.4 +3.6	75.7→77.7 +2.0
L12	93.1→97.6 +4.5	90.9→94.9 +4.0	89.6→92.7 +3.1	87.7→90.1 +2.4
L13	94.0→98.5 +4.5	91.5→96.4 +4.9	90.0→94.1 +4.1	88.2→91.6 +3.4
L14	90.0→95.4 +5.4	84.8→90.4 +5.6	81.1→85.5 +4.4	77.3→80.7 +3.4
L15	91.2→96.4 +5.2	88.3→93.2 +4.9	86.5→90.3 +3.8	84.2→87.2 +3.0
L16	92.5→97.4 +4.9	89.8→94.5 +4.7	88.1→91.9 +3.8	86.1→88.9 +2.8
L17	99.4→100.0 +0.6	98.8→99.8 +1.0	98.2→99.5 +1.3	97.4→99.0 +1.6
L18	87.3→94.6 **+7.3**	82.1→88.7 +6.6	78.8→83.5 +4.7	75.5→79.1 +3.6
L19	87.9→95.2 **+7.3**	83.2→89.7 +6.5	80.1→85.1 +5.0	77.2→81.3 +4.1
L20	96.6→99.4 +2.8	94.0→97.6 +3.6	92.5→95.6 +3.1	90.6→93.1 +2.5
Mean	91.6→96.8 +5.2	88.2→93.2 +5.0	85.9→90.0 +4.1	83.6→86.7 +3.1
c=4096				
L0	92.6→99.5 +6.9	88.0→97.6 **+9.6**	85.4→94.3 **+8.9**	83.4→89.7 +6.3
L1	99.2→100.0 +0.8	98.6→100.0 +1.4	98.0→99.8 +1.8	97.4→99.4 +2.0
L2	99.5→100.0 +0.5	99.0→100.0 +1.0	98.7→99.9 +1.2	98.2→99.6 +1.4
L3	98.9→100.0 +1.1	98.3→99.9 +1.6	98.0→99.7 +1.7	97.6→99.3 +1.7
L4	95.6→99.6 +4.0	91.8→98.3 +6.5	88.7→95.7 **+7.0**	85.7→91.8 +6.1
L5	97.0→99.8 +2.8	95.4→99.2 +3.8	94.3→98.3 +4.0	93.0→96.5 +3.5
L6	95.8→99.4 +3.6	94.0→98.4 +4.4	92.8→97.1 +4.3	91.5→95.0 +3.5
L7	95.4→99.5 +4.1	93.3→98.4 +5.1	92.0→96.8 +4.8	90.4→94.3 +3.9
L8	95.2→99.6 +4.4	93.4→98.7 +5.3	92.0→97.2 +5.2	90.2→94.9 +4.7
L9	95.2→99.5 +4.3	93.3→98.4 +5.1	91.7→96.8 +5.1	90.0→94.0 +4.0
L10	93.7→99.1 +5.4	90.4→97.0 +6.6	88.4→94.5 +6.1	85.7→89.6 +3.9
L11	93.9→99.1 +5.2	91.1→97.1 +6.0	89.2→94.4 +5.2	87.1→90.3 +3.2
L12	97.0→99.8 +2.8	95.8→99.2 +3.4	95.2→98.3 +3.1	94.3→96.9 +2.6
L13	97.5→99.9 +2.4	96.3→99.6 +3.3	95.5→99.0 +3.5	94.4→97.7 +3.3
L14	95.8→99.4 +3.6	93.2→98.2 +5.0	90.9→96.2 +5.3	88.2→92.7 +4.5
L15	96.1→99.6 +3.5	94.6→98.8 +4.2	93.5→97.6 +4.1	92.1→95.6 +3.5
L16	96.6→99.8 +3.2	95.1→99.1 +4.0	94.3→98.1 +3.8	93.0→96.1 +3.1
L17	99.9→100.0 +0.1	99.7→100.0 +0.3	99.5→100.0 +0.5	99.3→99.9 +0.6
L18	94.4→99.3 +4.9	91.3→97.7 +6.4	89.1→95.2 +6.1	86.7→91.4 +4.7
L19	94.8→99.5 +4.7	92.0→98.1 +6.1	90.0→95.9 +5.9	87.9→92.7 +4.8
L20	99.1→100.0 +0.9	97.8→99.8 +2.0	97.1→99.4 +2.3	96.0→98.5 +2.5
Mean	96.3→99.6 +3.3	94.4→98.7 +4.3	93.1→97.3 +4.2	91.5→95.0 +3.5

Table 17: **Coverage falls with context length, and training gains most in the longest contexts.** Top-p coverage (%) per layer at g=16 in the length buckets from 64K upward, step 0 $\rightarrow$ checkpoint and the gain, on 100 windows per bucket. Bold marks gains of at least 5 points.

Layer	64K–128K	128K–256K	256K–512K	>512K
c=2048				
L0	86.5→94.7 **+8.2**	79.6→90.3 **+10.7**	73.0→86.1 **+13.1**	72.2→86.6 **+14.4**
L1	96.9→99.3 +2.4	95.9→98.8 +2.9	95.4→98.6 +3.2	95.3→98.6 +3.3
L2	97.9→99.6 +1.7	97.0→99.1 +2.1	96.4→98.8 +2.4	96.1→98.7 +2.6
L3	97.4→99.4 +2.0	96.9→99.0 +2.1	96.1→98.7 +2.6	95.0→98.1 +3.1
L4	91.2→95.8 +4.6	88.8→94.2 **+5.4**	85.6→92.6 **+7.0**	85.7→92.2 **+6.5**
L5	94.4→97.5 +3.1	93.4→96.8 +3.4	92.0→95.9 +3.9	90.0→94.5 +4.5
L6	93.3→96.7 +3.4	92.1→95.9 +3.8	90.5→94.5 +4.0	86.5→91.9 **+5.4**
L7	92.4→96.4 +4.0	90.9→95.4 +4.5	88.9→94.0 **+5.1**	85.5→91.7 **+6.2**
L8	92.0→96.2 +4.2	90.3→95.3 **+5.0**	88.5→94.2 **+5.7**	84.9→91.7 **+6.8**
L9	93.1→97.1 +4.0	92.4→96.6 +4.2	89.9→95.0 **+5.1**	86.5→92.7 **+6.2**
L10	89.9→95.3 **+5.4**	89.0→94.4 **+5.4**	84.8→91.6 **+6.8**	83.1→90.6 **+7.5**
L11	92.8→96.2 +3.4	91.6→95.3 +3.7	88.9→93.2 +4.3	87.1→92.3 **+5.2**
L12	95.3→98.0 +2.7	95.6→98.1 +2.5	94.1→97.2 +3.1	91.3→95.5 +4.2
L13	95.8→98.4 +2.6	94.9→98.0 +3.1	94.7→98.0 +3.3	92.4→96.5 +4.1
L14	90.7→95.1 +4.4	88.4→93.5 **+5.1**	87.1→92.5 **+5.4**	85.4→91.7 **+6.3**
L15	94.1→97.4 +3.3	93.7→97.1 +3.4	92.1→96.1 +4.0	90.1→95.1 **+5.0**
L16	96.2→98.5 +2.3	95.4→98.1 +2.7	93.8→97.3 +3.5	92.5→96.4 +3.9
L17	99.5→99.9 +0.4	99.3→99.9 +0.6	99.1→99.9 +0.8	99.1→99.8 +0.7
L18	91.0→95.7 +4.7	88.6→94.3 **+5.7**	86.4→92.8 **+6.4**	84.4→91.8 **+7.4**
L19	92.0→96.4 +4.4	90.0→95.2 **+5.2**	87.3→93.4 **+6.1**	83.9→90.9 **+7.0**
L20	96.7→99.0 +2.3	95.5→98.2 +2.7	93.9→97.4 +3.5	92.6→96.6 +4.0
Mean	93.8→97.3 +3.5	92.3→96.4 +4.1	90.4→95.1 +4.7	88.6→94.0 **+5.4**
c=4096				
L0	93.1→99.5 **+6.4**	87.2→98.2 **+11.0**	81.0→96.1 **+15.1**	80.2→96.0 **+15.8**
L1	99.1→100.0 +0.9	98.5→100.0 +1.5	98.1→99.9 +1.8	98.2→99.9 +1.7
L2	99.5→100.0 +0.5	99.1→100.0 +0.9	98.7→100.0 +1.3	98.7→99.9 +1.2
L3	99.2→100.0 +0.8	98.8→100.0 +1.2	98.4→99.9 +1.5	97.8→99.9 +2.1
L4	96.3→99.6 +3.3	94.3→99.1 +4.8	91.6→98.4 **+6.8**	91.7→98.1 **+6.4**
L5	97.8→99.8 +2.0	97.1→99.6 +2.5	96.0→99.3 +3.3	94.7→99.0 +4.3
L6	97.1→99.6 +2.5	96.2→99.3 +3.1	94.9→98.6 +3.7	92.5→97.8 **+5.3**
L7	96.5→99.5 +3.0	95.4→99.2 +3.8	93.9→98.6 +4.7	91.6→97.8 **+6.2**
L8	96.4→99.6 +3.2	95.1→99.3 +4.2	93.6→98.9 **+5.3**	91.2→98.0 **+6.8**
L9	96.7→99.6 +2.9	96.1→99.4 +3.3	94.2→98.8 +4.6	91.8→98.0 **+6.2**
L10	95.2→99.3 +4.1	94.4→98.9 +4.5	90.9→97.6 **+6.7**	89.5→96.9 **+7.4**
L11	96.8→99.4 +2.6	95.9→99.1 +3.2	93.7→98.2 +4.5	92.3→97.6 **+5.3**
L12	97.7→99.7 +2.0	97.9→99.8 +1.9	96.9→99.5 +2.6	95.1→99.1 +4.0
L13	98.3→99.9 +1.6	97.6→99.8 +2.2	97.3→99.8 +2.5	96.0→99.4 +3.4
L14	96.1→99.4 +3.3	94.6→98.9 +4.3	93.2→98.1 +4.9	91.9→97.9 **+6.0**
L15	97.3→99.6 +2.3	96.8→99.5 +2.7	95.8→99.2 +3.4	94.2→98.9 +4.7
L16	98.2→99.8 +1.6	97.7→99.7 +2.0	96.5→99.5 +3.0	95.5→99.2 +3.7
L17	99.9→100.0 +0.1	99.8→100.0 +0.2	99.7→100.0 +0.3	99.8→100.0 +0.2
L18	96.1→99.4 +3.3	94.2→99.0 +4.8	92.1→98.3 **+6.2**	90.6→97.8 **+7.2**
L19	96.6→99.6 +3.0	95.1→99.3 +4.2	92.9→98.6 **+5.7**	90.5→97.6 **+7.1**
L20	99.1→100.0 +0.9	98.4→99.9 +1.5	97.3→99.8 +2.5	96.6→99.7 +3.1
Mean	97.3→99.7 +2.4	96.2→99.4 +3.2	94.6→98.9 +4.3	93.3→98.5 **+5.2**

Table 18: **Top-k coverage falls faster with length than top-p, and its training gain grows with length too.** Same layout and windows as Table 17, with top-k coverage (%). Bold marks gains of at least 7 points.

Layer	64K–128K	128K–256K	256K–512K	>512K
c=2048				
L0	83.6→92.0 **+8.4**	76.8→86.7 **+9.9**	70.1→81.8 **+11.7**	68.1→81.9 **+13.8**
L1	96.1→99.1 +3.0	94.9→98.4 +3.5	94.3→98.2 +3.9	94.2→98.2 +4.0
L2	97.0→99.4 +2.4	95.9→98.7 +2.8	95.1→98.3 +3.2	94.8→98.2 +3.4
L3	95.5→98.8 +3.3	94.7→98.2 +3.5	93.5→97.7 +4.2	92.1→96.6 +4.5
L4	84.4→91.1 +6.7	80.9→88.6 **+7.7**	76.6→85.7 **+9.1**	76.0→84.8 **+8.8**
L5	89.5→94.6 +5.1	87.7→93.1 +5.4	85.4→91.5 +6.1	82.9→89.4 +6.5
L6	88.2→92.9 +4.7	85.9→91.3 +5.4	83.4→89.0 +5.6	77.9→84.7 +6.8
L7	86.8→92.0 +5.2	84.2→90.3 +6.1	81.5→88.1 +6.6	76.9→84.1 **+7.2**
L8	87.1→92.6 +5.5	84.6→91.2 +6.6	82.1→89.6 **+7.5**	77.4→85.4 **+8.0**
L9	87.5→92.6 +5.1	85.6→91.6 +6.0	82.4→89.1 +6.7	77.4→84.9 **+7.5**
L10	82.0→88.1 +6.1	79.9→86.8 +6.9	74.8→82.4 **+7.6**	71.1→79.5 **+8.4**
L11	82.6→87.8 +5.2	80.3→86.3 +6.0	76.0→82.3 +6.3	71.9→79.0 **+7.1**
L12	91.2→95.0 +3.8	91.3→95.3 +4.0	89.2→93.7 +4.5	84.2→90.1 +5.9
L13	92.0→96.6 +4.6	90.4→95.8 +5.4	90.2→95.8 +5.6	86.5→93.2 +6.7
L14	85.6→91.0 +5.4	82.7→88.8 +6.1	81.3→87.5 +6.2	77.8→85.1 **+7.3**
L15	88.4→93.1 +4.7	87.6→92.7 +5.1	85.7→91.4 +5.7	82.4→88.8 +6.4
L16	90.5→94.9 +4.4	89.0→94.0 +5.0	87.0→92.6 +5.6	83.7→90.3 +6.6
L17	98.9→99.9 +1.0	98.6→99.8 +1.2	98.2→99.7 +1.5	98.0→99.5 +1.5
L18	83.3→89.7 +6.4	79.5→87.1 **+7.6**	76.3→84.1 **+7.8**	73.3→82.1 **+8.8**
L19	84.4→90.8 +6.4	81.2→88.8 **+7.6**	77.4→85.4 **+8.0**	73.5→81.5 **+8.0**
L20	95.1→98.3 +3.2	93.3→97.1 +3.8	91.2→96.1 +4.9	89.8→95.1 +5.3
Mean	89.0→93.8 +4.8	86.9→92.4 +5.5	84.4→90.5 +6.1	81.4→88.2 +6.8
c=4096				
L0	91.7→99.2 **+7.5**	85.5→97.5 **+12.0**	79.0→94.8 **+15.8**	77.1→94.6 **+17.5**
L1	98.9→100.0 +1.1	98.2→99.9 +1.7	97.7→99.9 +2.2	97.7→99.9 +2.2
L2	99.3→100.0 +0.7	98.8→100.0 +1.2	98.3→100.0 +1.7	98.3→99.9 +1.6
L3	98.6→100.0 +1.4	98.1→99.9 +1.8	97.5→99.9 +2.4	96.6→99.8 +3.2
L4	93.5→99.0 +5.5	90.3→98.2 **+7.9**	86.2→96.9 **+10.7**	86.0→96.2 **+10.2**
L5	96.0→99.5 +3.5	94.6→99.2 +4.6	92.8→98.6 +5.8	91.1→98.0 +6.9
L6	95.2→99.1 +3.9	93.4→98.4 +5.0	91.2→97.2 +6.0	87.7→95.7 **+8.0**
L7	94.3→99.0 +4.7	92.2→98.4 +6.2	90.1→97.3 **+7.2**	86.8→95.9 **+9.1**
L8	94.5→99.2 +4.7	92.3→98.7 +6.4	90.3→97.9 **+7.6**	86.9→96.5 **+9.6**
L9	94.3→98.9 +4.6	92.9→98.5 +5.6	90.2→97.5 **+7.3**	86.5→95.8 **+9.3**
L10	91.7→97.9 +6.2	89.7→97.3 **+7.6**	85.2→94.8 **+9.6**	82.0→93.1 **+11.1**
L11	92.3→97.9 +5.6	90.4→97.2 +6.8	86.5→95.0 **+8.5**	83.0→93.0 **+10.0**
L12	96.0→99.2 +3.2	96.0→99.4 +3.4	94.5→98.9 +4.4	91.3→98.0 +6.7
L13	96.9→99.8 +2.9	95.6→99.6 +4.0	95.1→99.5 +4.4	92.9→98.9 +6.0
L14	94.1→98.8 +4.7	91.9→98.0 +6.1	90.2→96.9 +6.7	87.6→96.1 **+8.5**
L15	94.9→99.0 +4.1	94.0→98.8 +4.8	92.4→98.2 +5.8	89.7→97.3 **+7.6**
L16	95.8→99.3 +3.5	94.6→99.1 +4.5	92.9→98.6 +5.7	90.4→97.7 **+7.3**
L17	99.8→100.0 +0.2	99.7→100.0 +0.3	99.4→100.0 +0.6	99.5→100.0 +0.5
L18	92.9→98.6 +5.7	89.7→97.6 **+7.9**	86.4→95.9 **+9.5**	83.9→94.9 **+11.0**
L19	93.6→98.9 +5.3	90.9→98.2 **+7.3**	87.4→96.7 **+9.3**	84.3→94.8 **+10.5**
L20	98.6→99.9 +1.3	97.6→99.8 +2.2	96.2→99.7 +3.5	95.4→99.5 +4.1
Mean	95.4→99.2 +3.8	93.6→98.8 +5.2	91.4→97.8 +6.4	89.3→96.9 **+7.6**