LITETOPK: EXPLOITING THE CURSE OF DIMENSIONALITY FOR A FUSED INDEXER-TOPK KERNEL IN LONG-CONTEXT SPARSE ATTENTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Indexer-TopK, which computes scores and selects the top-k candidates, is a key primitive in long-context sparse attention for large language models and vector retrieval for recommendation systems and vector databases. However, existing GPU-based Indexer-TopK kernels like DeepSeek Sparse Attention (DSA) typically separate scoring from selection and materialize the full score matrix in high-bandwidth memory. This strategy is inefficient for large score matrices, such as those encountered in long-context sparse attention, due to excessive global-memory traffic, costly synchronization, and prohibitive memory overhead. In this study, motivated by the curse of dimensionality, we observe that sparse attention scores exhibit a concentration phenomenon, with scores falling within a narrow range. Based on this observation, we propose LiteTopK, an efficient fused Indexer-TopK kernel. LiteTopK samples a small subset of data to estimate query-data score ranges, then partitions candidates into bins accordingly. This design allows LiteTopK to maintain a slightly relaxed threshold online and write back only promising candidates, reducing unnecessary I/O and memory overhead while preserving exact top-k correctness. We integrate LiteTopK into three widely used LLM training and inference systems: vLLM, SGLang, and Megatron-LM. Experiments in a production environment with eight NVIDIA B200 GPUs demonstrate prefill speedups of up to 1.49$\times$ across GLM-5.2, DeepSeek-V4-Flash, and Hy-4-Preview at a context length of 1M tokens, without compromising model capabilities.

1 INTRODUCTION

Indexer-TopK, which computes scores under a specific scoring function and selects the per-row top-k elements, has become a core primitive in modern machine learning systems, with applications in sparse attention of large language model (LLM) (Tang et al., 2024; Liu et al., 2025; Synk et al., 2025), recommendation systems (Khandagale et al., 2025; Gao et al., 2021), and information retrieval (Lee et al., 2023). Representative examples include:

1. As context lengths grow to millions of tokens, attention computation becomes prohibitively expensive, motivating sparse-attention systems (Ribar et al., 2024; Chen et al., 2024; Liu et al., 2025; Synk et al., 2025; Tang et al., 2024) such as DeepSeek Sparse Attention (DSA) (DeepSeek-AI, 2026) to approximate full attention using only a small subset of tokens. Specifically, DSA treats the current text chunk as queries and the historical context as candidate keys and values. It computes and stores lightweight relevance scores between the preceding context and the chunk text, then applies top-k selection to identify the most relevant historical tokens for constructing sparse attention. While this design reduces the cost of full attention, its score-computation and selection pipeline itself becomes a major performance bottleneck. As shown in Figure 1, the DSA kernel accounts for 70.9% of the total prefill runtime of GLM-5.2 (GLM-5-Team et al., 2026) in SGLang Zheng et al. (2024) at a 1M-token context length.
2. Recommendation systems and information retrieval systems typically encode queries/users and candidate items into high-dimensional embeddings using deep learning models (Khandagale et al., 2025; Gao et al., 2021). During candidate retrieval, they use Indexer-TopK to identify the top-k candidates with the highest similarity scores, where k can be on the order of thousands. The retrieved candidates are then re-ranked by more sophisticated models.

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.

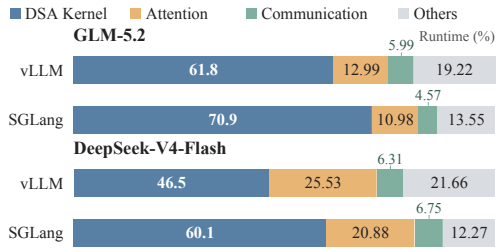


Figure 1: Prefill runtime breakdown at a 1M-token context length for GLM-5.2 and DeepSeek-V4-Flash on eight NVIDIA B200 GPUs using tensor and expert parallelism.

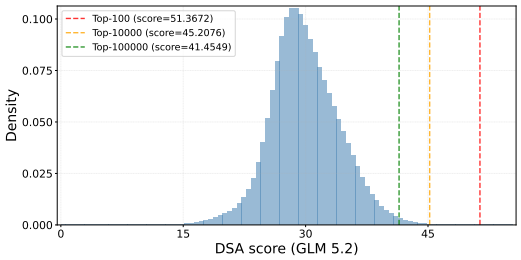


Figure 2: DSA score distribution in the first layer of GLM-5.2 during prefill on the Wiki dataset.

Despite the broad applications, existing Indexer-TopK implementations remain inefficient on GPUs because they typically decouple score computation and selection into two stages: first, computing and materializing the full score matrix in high-bandwidth memory (HBM) and then performing a top-k selection, such as radix-select (Alabi et al., 2012). This strategy introduces two major overheads. First, it incurs substantial memory overhead. For example, the DSA kernel incurs a runtime memory overhead of 32 GB under a 1M-token prefill with a moderate chunk size (e.g., 8192 in vLLM Kwon et al. (2023)). This is because the kernel needs to keep the full score matrix before the top-k selection. Such runtime memory overhead reduces the portion of HBM for storing the KV cache, which is already tight in existing LLM serving systems. To mitigate this, vLLM and SGLang further split the 8192-token chunk into smaller sub-chunks during prefill, such as 512, to reduce the HBM footprint, but at the cost of lower prefill throughput. Second, it incurs significant latency overhead, since the score matrix must be written back to HBM and subsequently read multiple times during Top-k selection, leading to significant latency. The same issue is also noted in the vLLM blog[1], which observes that "a clear challenge is that at high batch size with long context, the logits tensor is materialized before running a row-wise top-k."

To address this, we study the distribution of the scores in sparse attention and observe that these scores consistently show a concentration phenomenon, which is brought by the curse of dimensionality in high-dimensional spaces (Weber et al., 1998; Indyk & Motwani, 1998). In particular, similarity scores between high-dimensional vectors, including Euclidean distance and inner product, tend to concentrate within a narrow range while exhibiting an extremely long-tailed distribution, which is widely observed in real-world datasets (Bruch, 2024). Since sparse attention scores are computed from the similarity scores between high-dimensional vectors, this concentration effect also appears in sparse attention. For example, DSA applies ReLU-based weighting to inner-product scores, resulting in a distribution that exhibits the same narrow-range concentration and long-tail behavior. As shown in Figure 2, for the first layer of GLM-5.2 under a 1M-token context, DSA scores are concentrated between 25 and 35, while scores greater than or equal to 45 are extremely rare.

Building on this pattern, we propose LiteTopK, a kernel that follows a sample–filter–select framework to initialize and dynamically update a slightly relaxed pruning threshold, reducing memory overhead and I/O costs. During sampling, LiteTopK exploits score concentration by sampling a small subset of candidates and computing their scores against a query chunk to estimate the score distribution for each query token. It then applies equal-width quantization to divide the estimated score range into non-overlapping intervals, or called bins, and constructs a histogram of candidate counts across these bins. The bin containing the local k-th largest score serves as the initial threshold bin. During filtering, LiteTopK computes scores for all candidates and writes back only those with scores at or above the threshold bin's lower distance bound as potential top-k candidates, discarding the rest. However, two challenges remain: (1) writing all candidates to a shared buffer causes write contention that disrupts the computation pipeline; and (2) updating the threshold in real time incurs frequent cache reads that slow overall execution. We address these challenges with two techniques: (1) maintaining a local queue for each lane within each warp and flushing it in batches once it reaches a predefined size to reduce write contention; and (2) using idle warps to lazily update the threshold bin. During selection, only candidates within the threshold bin require further selection, while candidates in higher-scoring bins are written directly to the output, thereby reducing selection overhead.

In summary, this work makes the following contributions:

[1] https://vllm.ai/blog/2025-09-29-deepseek-v3-2

- To the best of our knowledge, we are the first to observe and characterize the score concentration phenomenon in sparse attention, trace its origin to the well-known curse of dimensionality in high-dimensional vector spaces, and further exploit this property to improve sparse attention kernel design.
- We propose LiteTopK, to the best of our knowledge the first sparse-attention Indexer-TopK kernel that fuses exact top-k selection into indexer scoring without materializing the full score matrix in HBM. This substantially reduces write-back memory pressure while preserving exact top-k results. Compared with the existing DSA kernel, LiteTopK delivers kernel-level speedups of up to $1.72\times$ during prefill and $1.88\times$ during decoding. Its design principles also can extend to other Indexer-TopK kernels.
- In a real-world deployment on 8 B200 GPUs, LiteTopK achieves up to a $1.49\times$ end-to-end speedup in the prefill stage across GLM-5.2, DeepSeek-V4-Flash, and Hy-4-Preview, while using less memory. The source code of our implementation is available at `https://anonymous.4open.science/r/LiteTopK-7822/`.

2 BACKGROUND

Sparse Attention. As million-token context windows are increasingly common in real-world LLM production environments, sparse attention algorithms (Lai et al., 2026; Yuan et al., 2025; Liu et al., 2025; DeepSeek-AI, 2026; Tang et al., 2024; Ribar et al., 2024; GLM-5-Team et al., 2026; Tencent Hy Team, 2026) has become indispensable for reducing the I/O and computational costs of full attention. A representative example is DeepSeek Sparse Attention (DSA) (Liu et al., 2025), a natively trained sparse attention module, which has become the foundation for massively deployed models such as DeepSeek-V3.2/V4 Liu et al. (2025); DeepSeek-AI (2026), GLM-5 family (GLM-5-Team et al., 2026), Hy-4-Preview Tencent Hy Team (2026), Qwen3.8-Flash-Next Qwen Team (2026), and Longcat-2.0 Meituan LongCat Team (2026). DSA follows a score-then-select paradigm. It develops a lightweight indexer, which stores an indexer key vector $k_s \in \mathbb{R}^d$ for every preceding token s. For a query token t, it derives H indexer heads $q_{t,j} \in \mathbb{R}^d, j = 1, ..., H$ and aggregates their head-wise scores to compute the index score as follows:

$$I_{t,s} = \sum_{j}^{H} w_{t,j} \, \mathrm{ReLU}\big((q_{t,j})^\top k_s\big) \, . \tag{1}$$

where $w_{t,j}$ is a query-dependent weight used to aggregate the per-head scores. A per-query top-k selection with $k = 2,048$ is then applied to the aggregated score $I_{t,s}$, and full attention is computed over the Key/Value vectors of the selected top-2,048 tokens.

Another class of sparse-attention methods is training-free, meaning that they can be directly applied to off-the-shelf LLM checkpoints without modifying model weights or requiring sparse-attention pretraining Yang et al. (2025b;a); Li et al. (2024). These methods typically follow the same design: performing score-then-select independently for each attention head. They mainly differ in how per-head relevance scores are computed, ranging from exact query-key inner products (Gupta et al., 2021; Yang et al., 2025a) to approximate scores derived from reduced representations, such as salient query channels (Ribar et al., 2024) and product quantization (Zhang et al., 2025). However, these training-free methods typically introduce non-negligible accuracy degradation Nawrot et al. (2026); Yuan et al. (2025). In this study, we integrate LiteTopK with DSA, while LiteTopK is broadly applicable to other score-then-select sparse-attention paradigms as well. It is also worth noting that LiteTopK is compatible with both training and inference systems (See Section 3.2).

LLM System. To support the training and inference of large language models, LLM systems have become essential infrastructure, with representative systems including vLLM Kwon et al. (2023), SGLang Zheng et al. (2024), Megatron-LM Shoeybi et al. (2019), and FlashInfer Ye et al. (2025). These systems aim to improve efficiency, including throughput and latency, as well as GPU memory utilization, while preserving algorithmic correctness. Existing efforts can be broadly categorized along four main dimensions: (1) reducing GPU memory traffic, as in FlashAttention Dao et al. (2022), which tiles attention and applies online softmax so that the attention score matrix is never materialized in HBM; (2) increasing parallelism, as in FlashDecoding Hong et al. (2024), which splits the KV dimension across CTAs to better saturate GPU SMs when the number of queries is small; (3) overlapping computation with data movement, as in FlashAttention-3 Shah et al. (2024), which uses

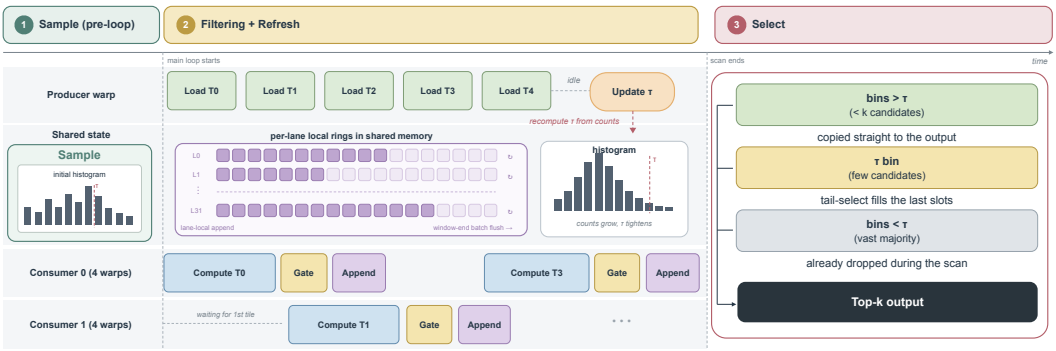


Figure 3: Illustration of the LiteTopK.

warp-specialized pipelines and asynchronous copies to hide memory-access latency; and (4) reducing redundant memory overhead, as in PagedAttention Kwon et al. (2023) and RadixAttention Zheng et al. (2024), which respectively manage KV cache with paged memory allocation and reuse shared prefix states across requests. Despite the importance of GPU memory management, to the best of our knowledge, no prior LLM serving system has explored fusing indexer scoring with top-k selection to avoid materializing intermediate scores in HBM. To fill this gap, we propose LiteTopK, which not only reduces the latency but also significantly reduces memory overhead.

GPU Architecture. Parallelism is fundamental to modern high-performance GPU computation. One source of such parallelism comes from the heterogeneous computational units: modern NVIDIA GPUs use Tensor Cores for matrix-matrix multiplication and CUDA cores for scalar and element-wise operations. Since these units have separate pipelines and can execute concurrently, per-score scalar operations can overlap with Tensor Core score computation. For example, in DSA, the dot products are computed on Tensor Cores, while the ReLU and weighted aggregation are performed on CUDA cores (see Equation 1). Another source is thread-level parallelism: GPU kernels are organized into Cooperative Thread Arrays (CTAs), each containing multiple warps. Beginning with Hopper, warps within a CTA can assume specialized roles: producer warps load tiles from HBM into shared memory, while consumer warps process previously loaded tiles to compute scores. This forms a software pipeline that overlaps data movement with computation (see Figure 3). LiteTopK exploits these pipeline gaps to hide its additional operations.

3 Methodology

3.1 Problem Definition and Overview

Problem Definition. Given a query set Q and a candidate set X, our objective is to identify, for each query q, the k candidates with the highest scores under a scoring function f:

$$\mathrm{TopK}(q) = \underset{x \in X}{\arg\operatorname{topk}} f(q, x), \qquad q \in Q. \tag{2}$$

This paper focuses on sparse attention, where f is an indexer score, such as the DSA score defined in Equation 1. We also consider other settings, including k nearest neighbor (KNN) search.

Overview. Figure 3 presents a CTA-level overview of our proposed LiteTopK. Specifically, LiteTopK follows a sample-filter-select workflow. Before the main loop starts, LiteTopK performs the sample phase. It samples a small subset of the dataset, computes the score between the query and each sampled data point, and partitions the interval from the minimum to the maximum sampled score into equal-width bins. The bin containing the local top-k candidate is then identified as the threshold bin (τ). Since the global top-k threshold is guaranteed to be no lower than the local top-k threshold, filtering out candidates whose scores fall below τ's lower distance preserves correctness.

After the sample phase, LiteTopK enters the main loop, fusing its filtering operations with existing indexer kernels. In particular, modern indexers typically follow a warp-specialized pipeline, as described in Section 2 and illustrated in Figure 3. Within each CTA, different warps are assigned specialized roles for either loading data or computing scores: the producer warp loads a sequence of data tiles, while the two consumer warpgroups, each consisting of four warps, compute the corresponding scores. LiteTopK implements filtering as a lightweight integer gate embedded in the existing indexer pipeline. The lightweight gate is evaluated on the CUDA core where scores are computed and therefore does not interrupt the original pipeline execution. Then the passing

candidates are appended into the local queue in the shared memory, as shown in Figure 3, which are flushed in batches to HBM later. Regarding the refresh phase, we periodically use the idle warps to recompute the threshold bin from the histogram, so the gate tightens as the scan proceeds.

In the selection phase, candidates in bins above the final threshold bin are written directly to the output. Tail selection is performed only within the threshold bin, while candidates in all remaining bins are discarded without affecting accuracy. We next describe our proposed LiteTopK in detail.

3.2 LITETOPK

Sampling. The sample serves two purposes: it must span the score range so that the bins are well scaled, and it should contain scores close to the true top-k threshold so that the initial gate is tight. An overly coarse sample degrades the effectiveness of filtering. For sparse attention, we exploit the following observation.

> ***Observation 1.*** *Adjacent query tokens tend to select substantially overlapping sets of prefix tokens in their top-k attention results.*

During sparse-attention prefill, we leverage Observation 1 to construct the sample for the current chunk. Specifically, we select the k' prefix tokens (e.g., $k' = 3k$ for DSA) that appear most frequently in the top-k results of queries in the previous chunk. These tokens are likely to remain high-scoring, yielding a tight initial threshold. A prefill chunk contains thousands of queries (e.g., 8,192), so the cost of selecting the k' tokens is amortized over them. We also perform the selection asynchronously, making its overhead negligible in practice. For k-NN search, which lacks this temporal structure, random sampling suffices.

After obtaining the sample set, we score its k' candidates against each query to determine the minimum and maximum sample scores, $s_{\min}$ and $s_{\max}$. We then partition $[s_{\min}, s_{\max}]$ into m equal-width bins, with $\Delta = (s_{\max} - s_{\min})/m$ and $\delta = 1/\Delta$. A score s is assigned to bin

$$b(s) = \min\{m-1,\, \max\{0, \lfloor (s - s_{\min})\delta \rfloor\}\}, \qquad \delta = \frac{1}{\Delta}.$$

We initialize the histogram with the sample's bin counts and set the threshold bin to the bin containing the k-th largest sampled score. During filtering, a candidate passes the gate if its bin ID is at least the threshold bin ID. Since bin assignment is monotone in the score, every true top-k candidate lies in the threshold bin or a higher bin and therefore passes the gate.

Filtering. We fuse filtering into the scoring pipeline: once each score is computed, we map it to a floating-point bin coordinate $B_{t,s}$, and retain the candidate only if its bin ID meets the current threshold. The implementation focuses on two operations: computing bin IDs and writing surviving candidates to the shared-memory buffer. For DSA, we incorporate the affine bin mapping directly into the existing weighted sum:

$$\begin{aligned} B_{t,s} &= (I_{t,s} - s_{\min})\delta \\ &= \sum_j w'_{t,j}\, \mathrm{ReLU}\big(q_{t,j}^\top k_s\big) - s_{\min}\delta, \qquad w'_{t,j} = w_{t,j}\delta. \end{aligned} \tag{3}$$

Since $s_{\min}$ and δ remain fixed throughout the scan for each query, we pre-scale the weights once before loading them into registers. We then initialize the weighted-sum accumulator to $-s_{\min}\delta$ instead of zero. The existing FFMA chain thus produces $B_{t,s}$ directly, without a separate affine transform for each score. We convert this value to a bin ID and retain the candidate if its bin ID is at least the current threshold bin ID.

Storing the floating-point value $B_{t,s}$ preserves the within-bin score information needed for final ranking. Because $\delta > 0$, ranking by $B_{t,s}$ is equivalent to ranking by $I_{t,s}$ in exact arithmetic. At the final output stage, scores are converted back to their original scale using $I_{t,s} = B_{t,s}/\delta + s_{\min}$.

For the write-back of promising candidates, each lane within a warp maintains a small unordered staging list, as illustrated in Figure 3. Every lane appends the candidates that pass the gate to its own list, which substantially reduces write contention. Once a list reaches a predefined length (e.g., 32 or 64 entries), the warp flushes it to the shared-memory candidate buffer: the warp reserves a contiguous region with a single atomic update to the buffer tail, and all 32 lanes then cooperatively write their

entries into that region. This scheme requires only one tail update per batch, thereby reducing atomic contention while improving lane utilization during write-back.

Each candidate that passes the gate also atomically increments its histogram bin in a fire-and-forget manner. Since subsequent score computations never depend on the return values of these increments, they do not stall the computation pipeline. The accumulated counts are read only during the periodic threshold refresh described next.

Threshold Refresh. As the histogram grows, the gate can tighten beyond its initial setting. Periodically, an idle warp, such as the producer once it has issued all fetch commands or any other idle warp, recomputes the threshold bin by accumulating histogram counts from the highest bin downward until the cumulative count reaches k. The new value is published with a single plain store to one shared-memory slot, with no atomic operation and no barrier. Computing warps independently refresh their cached threshold at fixed intervals and continue using the previous value between reads, so the update does not disrupt computation execution. A stale threshold only loosens the gate, rarely admitting extra candidates under score concentration, but does not affect accuracy. Threshold maintenance therefore remains off the critical path, and does not stall the pipeline.

Top-k Selection. Once scoring is complete, the candidate buffer contains the surviving candidates and their floating-point bin-space scores. Let r denote the number of candidates in bins above the final threshold bin. The refresh rule guarantees that $r < k$, and these candidates are written directly to the output. We fill the remaining $k - r$ slots by selecting the highest-scoring candidates within the threshold bin based on their stored bin-space scores. The selected scores are converted back to their original score at the same time. Thus, LiteTopK restricts the final selection to candidates in a single bin, reducing the selection workload compared with decoupled designs that apply a full top-k procedure, such as radix selection, to all materialized scores. Because LiteTopK returns the exact original-scale scores $I_{t,s}$ of the selected candidates alongside their indices, it preserves the information required by the backward pass and thus serves as a drop-in replacement for the native operator in both inference and training systems.

Extension to Decoding. Decoding has a different cost profile from prefill. Prefill processes large token chunks and therefore incurs the costs of both (1) writing a large score matrix back to HBM and (2) performing selection over that matrix. By contrast, decoding generates only one token per step, largely eliminating the write-back cost and leaving top-k selection as the bottleneck. We therefore fuse selection directly into score computation. Each CTA maintains a local histogram whose bins are derived from the intrinsic bit representation of the scores rather than from sampled value ranges, and each computed score triggers a fire-and-forget atomic increment to the corresponding bin. After all scores have been computed, we aggregate the per-CTA histograms to identify the threshold bin and then restrict the final selection to elements within that bin.

4 Experiments

We organize our evaluation around four research questions: (1) does LiteTopK accelerate long-context prefilling end-to-end across vLLM, SGLang, and Megatron-LM? (2) does LiteTopK preserve exact Top-k outputs and the long-context capability of the underlying LLM? (3) are these end-to-end prefill speedups driven by lower kernel latency and reduced memory overhead? (4) do the benefits of LiteTopK generalize to other GPU architectures and workloads, such as decode and vector retrieval?

4.1 Experimental Setup

End-to-End Long-Context Prefilling. We integrate LiteTopK into three widely used LLM systems: vLLM, SGLang, and Megatron-LM. In each system, we replace only the native Indexer-TopK kernel with LiteTopK, while keeping all other framework components and settings unchanged within each paired comparison. We evaluate the FP8 checkpoints of three models with native support for contexts of up to 1M tokens: GLM-5.2 GLM-5-Team et al. (2026), DeepSeek-V4-Flash DeepSeek-AI (2026), and Hy-4-Preview Tencent Hy Team (2026). We construct the input prompts from the Salesforce WikiText dataset and use a batch size of one.

Because context parallelism (CP) support in vLLM and Megatron-LM is incomplete, we evaluate all three frameworks with expert parallelism (EP) and tensor parallelism (TP). We separately evaluate SGLang with CP enabled to examine its effect on the speedup achieved by LiteTopK. For a fair comparison, Each Native-LiteTopK pair uses the same model-specific parallel configuration. The complete configurations are provided in Appendix Table 5.

Table 1: End-to-end prefill latency on eight NVIDIA B200 GPUs with TP and EP. Latencies are reported in seconds. Speedup is calculated as Native latency divided by LiteTopK latency. Megatron-LM results for Hy-4-Preview are unavailable because the framework did not yet support this newly released model at the time of evaluation.

Model	Context	vLLM			SGLang			Megatron-LM		
		Native	LiteTopK	Speedup	Native	LiteTopK	Speedup	Native	LiteTopK	Speedup
GLM-5.2	256K	18.61	17.57	1.06×	20.18	17.93	1.13×	17.56	17.13	1.03×
	512K	48.00	42.59	1.13×	51.94	40.67	1.28×	45.53	41.85	1.09×
	768K	90.51	74.95	1.21×	94.03	72.39	1.30×	82.66	73.86	1.12×
	1M	145.81	113.27	1.29×	148.87	108.56	1.37×	136.29	113.18	1.20×
DeepSeek-V4-Flash	256K	6.92	6.63	1.04×	7.53	7.08	1.07×	8.34	8.11	1.03×
	512K	18.02	17.18	1.05×	21.60	18.27	1.18×	20.90	19.80	1.06×
	768K	33.49	31.13	1.08×	44.36	33.63	1.32×	40.37	35.14	1.15×
	1M	53.12	48.50	1.10×	73.06	53.12	1.38×	63.89	54.75	1.17×
Hy-4-Preview	256K	24.38	23.50	1.04×	23.57	22.76	1.04×	N/A	N/A	N/A
	512K	59.37	54.33	1.09×	58.71	51.29	1.14×	N/A	N/A	N/A
	768K	106.11	92.75	1.14×	105.61	86.94	1.21×	N/A	N/A	N/A
	1M	164.91	138.63	1.19×	164.92	130.29	1.27×	N/A	N/A	N/A

Table 2: End-to-end prefill latency in SGLang on eight NVIDIA B200 GPUs with TP=8 and CP=8. Latencies are reported in seconds. Speedup = Native / LiteTopK.

Context	GLM-5.2			DeepSeek-V4-Flash			Hy-4-Preview		
	Native	LiteTopK	Speedup	Native	LiteTopK	Speedup	Native	LiteTopK	Speedup
256K	4.17	4.07	1.02×	2.18	2.16	1.01×	5.62	5.44	1.03×
512K	9.45	8.83	1.07×	5.13	4.81	1.07×	13.20	12.04	1.10×
768K	16.30	14.62	1.11×	8.87	7.85	1.13×	24.57	19.81	1.24×
1M	24.62	21.11	1.17×	13.61	11.60	1.17×	41.71	28.02	1.49×

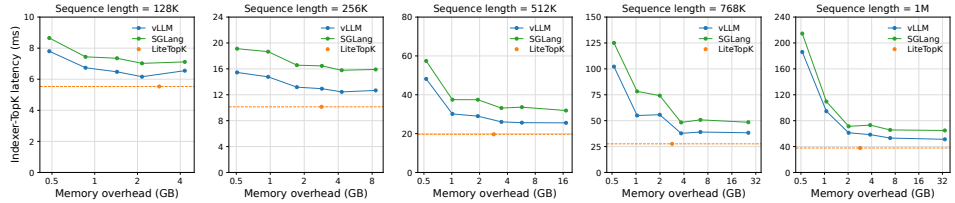
Exactness and Model Quality. We assess whether LiteTopK preserves model-level quality and kernel-level exactness. At the model level, we use a two-needle retrieval test based on the needle-in-a-haystack protocol, which is widely adopted to evaluate the long-context capabilities of large language models (Hsieh et al., 2024). The task inserts two target facts at controlled positions within a long distractor context and poses a question that can be answered correctly only by retrieving both facts. We use identical prompts and greedy-decoding settings for the native implementation and LiteTopK, vary the context length up to one million tokens, and report joint exact-match accuracy and output-token agreement. At the kernel level, we compare the Top-k indices produced by LiteTopK with those of the native implementation and report Top-k recall.

Sparse-Attention Kernel Benchmark. We next isolate the DSA workload to examine the source of the end-to-end improvements. We compare LiteTopK separately against two implementations from vLLM and SGLang. The benchmark inputs is constructed from the first-layer attention activations of GLM-5.2, using text sampled from Wikipedia (Merity et al., 2016). We consider context lengths of 256K, 512K, 768K, and 1M tokens, vary the prefill chunk size from 128 to 8,192 tokens, and fix k to 2,048. For each configuration, we report the aggregate kernel latency for processing 8,192 tokens. When the queries are divided into multiple chunks, the reported latency is the sum of the latency across all chunks. We additionally measure peak auxiliary memory consumption. Although peak memory overheads are impractical for production deployment, we retain these configurations to characterize the best latency achievable by the baselines when sufficient memory is available.

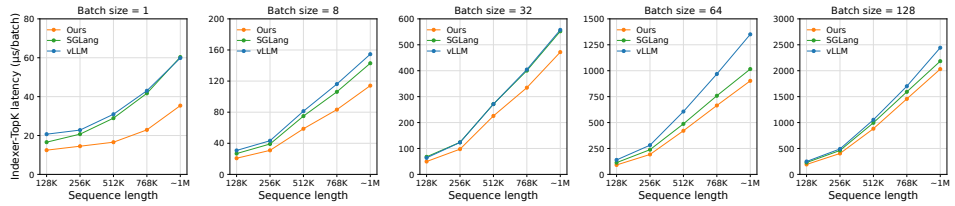
Large-Scale Retrieval Benchmark. Finally, we evaluate LiteTopK on large-scale vector retrieval to determine whether its efficiency gains generalize beyond sparse attention. We construct a corpus of five million passages sampled from MSMARCO-V2.1, a widely used text-retrieval dataset. Each passage is encoded as a 768-dimensional vector using the Snowflake Arctic-embed-m-v1.5 model (Merrick, 2024). We construct the query set Q by randomly sampling 1,000 passages and evaluating corpus sizes of 1M, 2M, 4M, and 5M vectors. We vary k over 128, 512, 1,024, 4,096, and 8,192 and use a batch size of 64 to emulate concurrent retrieval requests. We compare LiteTopK with Torch on this task.

Table 3: Joint exact-match accuracy (%) on the two-needle retrieval test, averaged over all needle positions. The four context-length buckets contain 50, 20, 10, and 5 test instances, respectively.

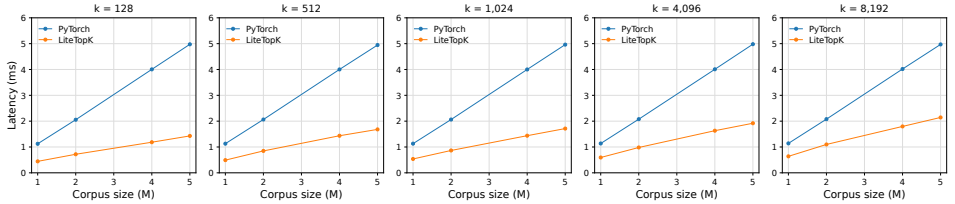
Model	Method	64K–128K	128K–256K	256K–512K	512K–1M
GLM-5.2	Native	96.10	90.86	90.25	63.16
	LiteTopK	96.11	90.89	90.25	63.12
DeepSeek-V4-Flash	Native	98.10	95.37	91.44	86.51
	LiteTopK	98.10	95.37	91.44	86.50
Hy-4-Preview	Native	94.42	81.98	47.11	42.70
	LiteTopK	94.45	81.99	47.40	42.70



(a) DSA Prefill Indexer-TopK Kernel



(b) DSA Decode Indexer-TopK Kernel



(c) Vector Retrieval

Figure 4: Runtime comparison between LiteTopK and existing methods on sparse-attention and retrieval workloads.

Implementation. All our implementations are written in CUDA C++ using CuTe/CUTLASS, while the baselines are evaluated base on their official implementations. The experiments are conducted on an NVIDIA B200 GPU with 179 GB of memory using CUDA 12.8, and we additionally conduct experiments on H100 GPUs. All reported running times are averaged over 20 runs.

4.2 Experimental Results

End-to-End Long-Context Prefilling. Table 1 reports end-to-end prefill latency on eight B200 GPUs. LiteTopK reduces latency across all evaluated models, systems, and context length configurations, with speedups increasing monotonically with context length. In particular, at a context length of 1M tokens, it achieves speedups of 1.21–1.37× for GLM 5.2, 1.10–1.38× for DeepSeek-V4-Flash, and 1.19–1.27× for Hy-4-Preview. Table 2 further shows that LiteTopK continues to provide end-to-end prefill speedups with context parallelism enabled. At a 1M-token context length, it achieves speedups of 1.17× for both GLM-5.2 and DeepSeek-V4-Flash, and 1.49× for Hy-4-Preview. The larger gain for Hy-4-Preview is attributable to its use of BF16 in attention for numerical stability and CP configuration, which leaves less HBM capacity available for the Indexer-TopK workspace and makes LiteTopK's memory savings particularly beneficial.

Exactness and Model Quality. We first evaluate whether LiteTopK preserves the outputs of the native Indexer-TopK operator. Across all evaluated configurations, LiteTopK achieves 100% recall relative to the native implementation. At the model level, Table 3 compares joint exact-match accuracy on the two-needle retrieval task. Across all three models and four context-length ranges, The maximum absolute difference is 0.29 percentage points. Small fluctuations also occur across repeated runs of the native vLLM and SGLang implementations, suggesting that the observed differences reflect run-to-run variability rather than a systematic degradation caused by LiteTopK. Overall, LiteTopK maintains nearly identical long-context retrieval accuracy.

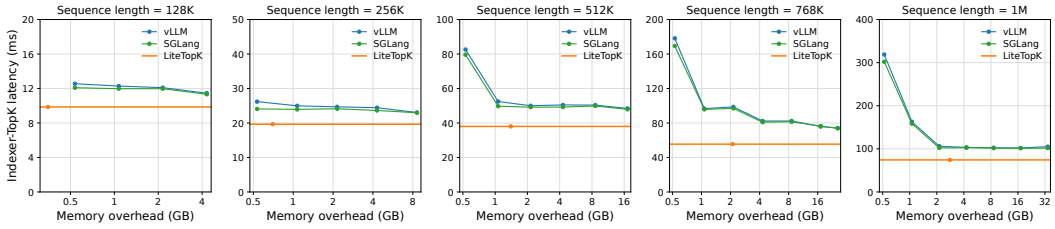


Figure 5: Runtime comparison of LiteTopK and existing DSA prefill Indexer-TopK kernels on an NVIDIA H100 GPU.

Sparse-Attention Indexer-TopK Kernel Performance. Figure 4a compares the Indexer-Topk kernel latency of LiteTopK, vLLM, and SGLang on a single NVIDIA B200 GPU. With a fixed query batch of 8,192 tokens and a context length of 1M, LiteTopK achieves a latency of 37.78 ms, compared with 51.25 ms for vLLM and 64.85 ms for SGLang, corresponding to speedups of $1.36\times$ and $1.72\times$, respectively. At 256K, the corresponding speedups are $1.25\times$ and $1.57\times$, respectively. These results show that LiteTopK's relative advantage increases with context length in the evaluated configurations.

LiteTopK also substantially reduces memory overhead. At 1M, processing all 8,192 query tokens in a single subcall requires approximately 32 GB of auxiliary workspace for the baselines, compared with 2.84 GB for LiteTopK. Reducing the baseline query chunk size lowers its memory footprint but increases the total latency. For example, vLLM takes 60.15 ms and 54.69 ms with query chunks of 1,024 and 1,536 tokens, using about 4 GB and 7 GB of workspace, respectively. LiteTopK can process the full query batch with a smaller memory footprint.

Decode Indexer-TopK Kernel Performance. Figure 4b compares the decode Indexer-TopK latency of LiteTopK, vLLM, and SGLang on a single NVIDIA B200 GPU. Across all evaluated configurations, LiteTopK achieves the lowest latency, delivering speedups of $1.08\times$–$1.82\times$ over SGLang and $1.13\times$–$1.88\times$ over vLLM. At batch size 1, the speedup over SGLang increases from $1.33\times$ at a context length of 128K to $1.82\times$ at 768K. At batch size 128, the corresponding speedup decreases from $1.19\times$ at 128K to $1.08\times$ at approximately 1M. The smaller gains at larger batch sizes are expected: indexer score computation accounts for a growing share of total decode latency, reducing the fraction attributable to TopK selection and limiting the speedup achievable with LiteTopK. Table 4 reports end-to-end decode speedups of up to $1.04\times$. These gains are modest because decoding is primarily memory-bound, and reducing Indexer–TopK overhead addresses only a small fraction of the overall memory-access cost.

Large-Scale Retrieval Performance. Figure 4c compares LiteTopK and PyTorch kernel latency for inner-product computation and top-k selection on MS MARCO passage embeddings, covering 1M–5M vectors and $k \in \{128, 512, 1024, 4096, 8192\}$. LiteTopK achieves $1.79\times$–$3.48\times$ speedups across all configurations. In particular, at 5M vectors, it reduces latency from 4.97 ms to 1.43 ms for $k = 128$, leading to $3.48\times$ speedup.

Table 4: Decode latency (s) of GLM-5.2 with SGLang on eight B200 GPUs.

256K			512K			768K			1M		
Native	LiteTopK	Speedup	Native	LiteTopK	Speedup	Native	LiteTopK	Speedup	Native	LiteTopK	Speedup
1.20	1.17	1.03×	1.24	1.21	1.03×	1.30	1.26	1.03×	1.36	1.31	1.04×

Portability to NVIDIA H100. Figure 5 compares the Indexer-TopK latency of LiteTopK, vLLM, and SGLang on a single NVIDIA H100 GPU with an 8,192-token prefill batch. At a 1M context length, LiteTopK completes in 74.37 ms, compared with 103.46 ms for vLLM and 103.56 ms for SGLang, yielding a $1.39\times$ speedup over both. At 256K, LiteTopK takes 19.64 ms, versus 24.25 ms for vLLM and 24.64 ms for SGLang, corresponding to $1.23\times$ and $1.25\times$ speedups, respectively. These results show that the acceleration observed on Blackwell GPUs carries over to the Hopper architecture.

5 Conclusion and Discussion

This paper presents LiteTopK, a fused Indexer-TopK kernel that exploits the concentration of sparse attention scores to avoid materializing the full score matrix, which accelerates both the prefill and decode stages of sparse attention in long-context inference. A promising direction for future work is to further exploit Observation 1, i.e., adjacent query tokens select largely overlapping top-k sets, in downstream sparse attention by merging their selected key/value entries into a shared, deduplicated scan list, to reduce I/O cost, thereby accelerating sparse attention computation.

REFERENCES

Tolu Alabi, Jeffrey D Blanchard, Bradley Gordon, and Russel Steinbach. Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)*, 17:4–1, 2012.

Sebastian Bruch. *Foundations of vector retrieval*, volume 1. Springer, 2024.

Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. Arkvale: Efficient generative llm inference with recallable key-value eviction. *Advances in Neural Information Processing Systems*, 37:113134–113155, 2024.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35: 16344–16359, 2022.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Weihao Gao, Xiangjun Fan, Chong Wang, Jiankai Sun, Kai Jia, Wenzi Xiao, Ruofan Ding, Xingyan Bin, Hui Yang, and Xiaobing Liu. Learning an end-to-end structure for retrieval in large-scale recommendations. In *Proceedings of the 30th ACM international conference on information & knowledge management*, pp. 524–533, 2021.

GLM-5-Team, :, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang, Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, Huilong Chen, Jiajie Zhang, Jian Jiao, Jiaqi Guo, Jingsen Wang, Jingzhao Du, Jinzhu Wu, Kedong Wang, Lei Li, Lin Fan, Lucen Zhong, Mingdao Liu, Mingming Zhao, Pengfan Du, Qian Dong, Rui Lu, Shuang-Li, Shulin Cao, Song Liu, Ting Jiang, Xiaodong Chen, Xiaohan Zhang, Xuancheng Huang, Xuezhen Dong, Yabo Xu, Yao Wei, Yifan An, Yilin Niu, Yitong Zhu, Yuanhao Wen, Yukuo Cen, Yushi Bai, Zhongpei Qiao, Zihan Wang, Zikang Wang, Zilin Zhu, Ziqiang Liu, Zixuan Li, Bojie Wang, Bosi Wen, Can Huang, Changpeng Cai, Chao Yu, Chen Li, Chengwei Hu, Chenhui Zhang, Dan Zhang, Daoyan Lin, Dayong Yang, Di Wang, Ding Ai, Erle Zhu, Fangzhou Yi, Feiyu Chen, Guohong Wen, Hailong Sun, Haisha Zhao, Haiyi Hu, Hanchen Zhang, Hanrui Liu, Hanyu Zhang, Hao Peng, Hao Tai, Haobo Zhang, He Liu, Hongwei Wang, Hongxi Yan, Hongyu Ge, Huan Liu, Huanpeng Chu, Jia'ni Zhao, Jiachen Wang, Jiajing Zhao, Jiamin Ren, Jiapeng Wang, Jiaxin Zhang, Jiayi Gui, Jiayue Zhao, Jijie Li, Jing An, Jing Li, Jingwei Yuan, Jinhua Du, Jinxin Liu, Junkai Zhi, Junwen Duan, Kaiyue Zhou, Kangjian Wei, Ke Wang, Keyun Luo, Laiqiang Zhang, Leigang Sha, Liang Xu, Lindong Wu, Lintao Ding, Lu Chen, Minghao Li, Nianyi Lin, Pan Ta, Qiang Zou, Rongjun Song, Ruiqi Yang, Shangqing Tu, Shangtong Yang, Shaoxiang Wu, Shengyan Zhang, Shijie Li, Shuang Li, Shuyi Fan, Wei Qin, Wei Tian, Weining Zhang, Wenbo Yu, Wenjie Liang, Xiang Kuang, Xiangmeng Cheng, Xiangyang Li, Xiaoquan Yan, Xiaowei Hu, Xiaoying Ling, Xing Fan, Xingye Xia, Xinyuan Zhang, Xinze Zhang, Xirui Pan, Xu Zou, Xunkai Zhang, Yadi Liu, Yandong Wu, Yanfu Li, Yidong Wang, Yifan Zhu, Yijun Tan, Yilin Zhou, Yiming Pan, Ying Zhang, Yinpei Su, Yipeng Geng, Yong Yan, Yonglin Tan, Yuean Bi, Yuhan Shen, Yuhao Yang, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yurong Wu, Yutao Zhang, Yuxi Duan, Yuxuan Zhang, Zezhen Liu, Zhengtao Jiang, Zhenhe Yan, Zheyu Zhang, Zhixiang Wei, Zhuo Chen, Zhuoer Feng, Zijun Yao, Ziwei Chai, Ziyuan Wang, Zuzhou Zhang, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Ankit Gupta, Guy Dar, Shaya Goodman, David Ciprut, and Jonathan Berant. Memory-efficient transformers via top-k attention. In *Proceedings of the Second Workshop on Simple and Efficient Natural Language Processing*, pp. 39–52, 2021.

Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. *Proceedings of Machine Learning and Systems*, 6:148–161, 2024.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: what's the real context size of your long-context language models? *CoRR*, abs/2404.06654, 2024. doi: 10.48550/ARXIV.2404.06654.

Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pp. 604–613, 1998.

Sujay Khandagale, Bhawna Juneja, Prabhat Agarwal, Aditya Subramanian, Jaewon Yang, and Yuting Wang. Interactrank: Personalized web-scale search pre-ranking with cross interaction features. In *Companion Proceedings of the ACM on Web Conference 2025, WWW 2025, Sydney, NSW, Australia, 28 April 2025 - 2 May 2025*, pp. 287–295, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Xunhao Lai, Weiqi Xu, Yufeng Yang, Qiaorui Chen, Yang Xu, Lunbin Zeng, Xiaolong Li, Haohai Sun, Haichao Zhu, Vito Zhang, et al. Minimax sparse attention. *arXiv preprint arXiv:2606.13392*, 2026.

Jinhyuk Lee, Zhuyun Dai, Sai Meher Karthik Duddu, Tao Lei, Iftekhar Naim, Ming-Wei Chang, and Vincent Zhao. Rethinking the role of token retrieval in multi-vector retrieval. *Advances in Neural Information Processing Systems*, 36:15384–15405, 2023.

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

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Meituan LongCat Team. Introducing longcat-2.0. `https://longcat.ai/blog/longcat-2.0`, June 2026. GitHub: `https://github.com/meituan-longcat/LongCat-2.0`.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

Luke Merrick. Embedding and clustering your data can improve contrastive pretraining. *arXiv preprint arXiv:2407.18887*, 2024.

Piotr Nawrot, Jianing Li, Renjie Huang, Sebastian Ruder, Kelly Marchisio, and Edoardo Maria Ponti. The sparse frontier: Sparse attention trade-offs in transformer llms. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 38667–38701, 2026.

Qwen Team. Qwen3.8-Flash-Next: A new architecture, towards ultimate cost-efficiency, August 2026. URL `https://qwen.ai/blog?id=qwen3.8-flash-next`.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. Sparq attention: Bandwidth-efficient LLM inference. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 42558–42583, 2024.

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

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Ryan Synk, Monte Hoover, John Kirchenbauer, Neel Jain, Alex Stein, Manli Shu, Josue Melendez Sanchez, Ramani Duraiswami, and Tom Goldstein. Exploiting sparsity for long context inference: Million token contexts on commodity gpus. *arXiv preprint arXiv:2502.06766*, 2025.

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

Tencent Hy Team. Hy4 preview. `https://github.com/Tencent-Hunyuan/Hy4-preview`, 2026. Open-weight Mixture-of-Experts language model, accessed 2026-08-30.

Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pp. 194–205, 1998.

Lijie Yang, Zhihao Zhang, Zhuofu Chen, Zikun Li, and Zhihao Jia. Tidaldecode: Fast and accurate LLM decoding with position persistent sparse attention. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*, 2025a.

Lijie Yang, Zhihao Zhang, Arti Jain, Shijie Cao, Baihong Yuan, Yiwei Chen, Zhihao Jia, and Ravi Netravali. Less is more: Training-free sparse attention with global locality for efficient reasoning. *arXiv preprint arXiv:2508.07101*, 2025b.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *Proceedings of Machine Learning and Systems*, 7, 2025.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, et al. 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, 2025.

Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. Pqcache: Product quantization-based kvcache for long context llm inference. *Proceedings of the ACM on Management of Data*, 3(3):1–30, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Table 5: Model-specific parallel configurations used in the end-to-end experiments. Each paired Native–LiteTopK comparison uses an identical configuration. A dash indicates that no qualified result was available.

Model	vLLM	SGLang	Megatron-LM
GLM-5.2	TP8 / EP8	TP8 / EP8	TP8 / EP8
DeepSeek-V4-Flash	TP8 / EP1	TP8 / EP1	TP8 / EP1
Hy-4-Preview	TP8 / EP8	TP8 / EP8	–

APPENDIX

A AI USAGE DISCLOSURE

In this work, we used generative AI tools to polish the writing and to obtain feedback on the presentation of our methodology and experiments. We have not used generative AI tools to implement methods, generate data, formulate claims, or interpret results. All AI-assisted content was reviewed and verified by the authors, who take full responsibility for the final content of this work, including text, claims, and artifacts.

B MODEL-SPECIFIC CONFIGURATIONS

Appendix Table 5 summarizes the model-specific and framework-specific parallel configurations used in our end-to-end evaluation, which are held fixed within each paired Native–LiteTopK comparison.