S2O: Early Stopping for Sparse Attention via Online Permutation

Yu Zhang[1*], Songwei Liu[1†*], Chenqian Yan[1],
Sheng Lin[1], Beichen Ning[2], Fangmin Chen[1‡], Xing Wang[1‡]
[1]ByteDance [2]Xiamen University

Abstract

Attention scales quadratically with sequence length, fundamentally limiting long-context inference. Existing block-granularity sparsification can reduce latency, but coarse blocks impose an intrinsic sparsity ceiling, making further improvements difficult even with carefully engineered designs. We present **S2O**, which performs early **s**topping for **s**parse attention via **o**nline permutation. Inspired by virtual-to-physical address mapping in memory systems, S2O revisits and factorizes FlashAttention execution, enabling inference to load non-contiguous tokens rather than a contiguous span in the original order. Motivated by fine-grained structures in attention heatmaps, we transform explicit permutation into an online, index-guided, discrete loading policy; with extremely lightweight preprocessing and index-remapping overhead, it concentrates importance on a small set of high-priority blocks. Building on this importance-guided online permutation for loading, S2O further introduces an early-stopping rule: computation proceeds from high to low importance; once the current block score falls below a threshold, S2O terminates early and skips the remaining low-contribution blocks, thereby increasing effective sparsity and reducing computation under a controlled error budget. As a result, S2O substantially raises the practical sparsity ceiling. On **Llama-3.1-8B** under a **128K** context, S2O reduces single-operator MSE by **3.82**× at matched sparsity, and reduces prefill compute density by **3.31**× at matched MSE; meanwhile, it preserves end-to-end accuracy and achieves **7.51**× attention and **3.81**× end-to-end speedups.

1 Introduction

Large language models (LLMs) have demonstrated strong general-purpose capabilities in natural language understanding, generation, reasoning, and cross-modal tasks(Zhao et al., 2025). A major driver of these gains is scaling both model size and context length.

However, as context length increases, transformer attention quickly becomes the dominant bottleneck due to its quadratic time complexity, severely limiting further progress in long-context inference. To mitigate this issue, a growing body of work has explored sparse attention. Among various sparsification schemes, block-sparse attention stands out as one of the most practical directions due to its simplicity and engineering feasibility: it integrates naturally with FlashAttention (Dao, 2024)'s tiling pipeline and maintains high GPU utilization. Under this paradigm, sparsity is typically applied to fixed-size compute blocks to satisfy system constraints such as aligned memory access and high parallelism.

The core challenge of block-sparse attention is deciding which attention blocks are worth computing under a limited budget. Existing approaches can be broadly grouped into two paradigms. **(i) Important-block selection** selects blocks under a fixed blocking scheme using predefined patterns or heuristic signals. This includes pattern-based sparsity (e.g., A-shape patterns induced by the attention-sink phenomenon (Xiao et al., 2023), Vertical/Slash patterns (Jiang et al., 2024), adaptive pattern switching (Lai et al., 2025), and richer pattern families (Li et al., 2025; He et al., 2025)) and signal-driven block importance estimation (e.g., min–max sampling (Tang et al., 2024), anti-diagonal probing (Xu et al., 2025), and similarity-based skipping (Zhang et al., 2025a)). **(ii) Permutation-based sparsification** improves efficiency by permuting token positions to cluster high-importance regions before sparse computation (Xi et al., 2025; Yang et al., 2025b). However, in autoregressive language models, causal masking imposes strict constraints, so existing methods are typically limited to local KV permutation (Wang et al., 2025), which hampers

*Equal contribution. †Project leader. ‡Corresponding author. Emails: <21831068@zju.edu.cn> (Project leader)

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 7737–7751

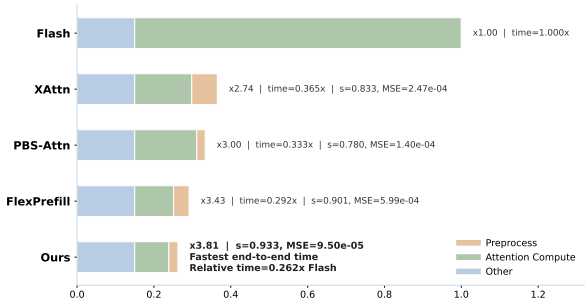


Figure 1: **Speedup at 128K context length (Llama-3-8B) with sparsity–error trade-off.** We report end-to-end latency speedup over FlashAttention and break down the major components, including sparse preprocessing time, attention compute time, and other overheads. We also show each method's sparsity ratio and the corresponding mean squared error (MSE; lower is better), highlighting that our method achieves lower error at higher sparsity.

global clustering. Moreover, permutation is often applied only once prior to computation, making it difficult to iteratively refine the ordering as the attention structure changes across segments or layers.

We make a key empirical observation: attention heatmaps in LLMs often exhibit thin, stripe-like structures rather than regular block-wise patterns (see Fig. 2). Under coarse blocking, even accurate block selection can still waste substantial computation on low-importance positions inside selected blocks, leading to significant intra-block redundancy and a practical sparsity ceiling. Therefore, achieving higher sparsity requires not only better block scoring, but also a mechanism that concentrates importance into a small set of blocks while avoiding the high overhead of physically permuting tokens.

To this end, we propose a FlashAttention-compatible solution with a global effect: **online permutation**. The key idea is to preserve the physical token layout in HBM and instead introduce an extremely lightweight permutation index as a form of logical-address mapping. Concretely, while keeping block-wise computation unchanged, we directly select and load discretely distributed query and key/value positions during the loading stage, emulating permutation benefits with negligible index-remapping overhead. This design is based on two observations: (i) on modern GPUs, high bandwidth utilization does not strictly require contiguous K/V loads; and (ii) the compute savings from higher sparsity can outweigh the extra memory-access overhead introduced by non-contiguous loading. In practice, we first permute Q at the segment level, aggregating highly correlated queries into a small number of contiguous Q segments; then, for each Q segment, we perform multiple lightweight permutation passes over its associated K/V candidates, trading a small preprocessing overhead for higher achievable sparsity. Motivated by stripe-like attention structures (Zhang et al., 2025b), we further introduce a stripe-granularity mean pooling signal that consolidates attention mass effectively (see Fig. 2c), producing reliable permutation cues at low cost.

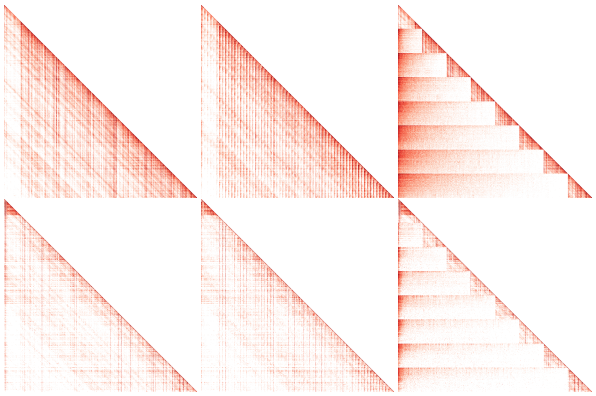
Online permutation naturally enables an **early-stopping** rule. Because candidate K/V blocks are processed in descending estimated importance under the online permutation order, once the marginal gain from newly processed blocks becomes negligible relative to the accumulated score, we terminate early and skip the remaining low-contribution blocks. This avoids rigidly committing to a fixed Top-K subset and improves effective sparsity under a controlled error budget.

Our main contributions are as follows: (1) We identify a practical sparsity ceiling under coarse block sparsity caused by a mismatch between stripe-like attention structures and block-wise computation, and propose **online permutation** to overcome this block-granularity limitation. (2) We introduce a FlashAttention-compatible, index-guided loading policy with a monotone-gain **early-stopping** rule, enabling global permutation effects without physically permuting tensors. (3) On **Llama-3-8B**(Grattafiori et al., 2024) under **128K** context, S2O improves the sparsity–error trade-off (**3.82**$\times$ lower operator MSE or **3.31**$\times$ higher sparsity), while preserving end-to-end accuracy and achieving up to **7.51**$\times$ attention and **3.81**$\times$ end-to-end speedups.

2 Motivation

2.1 How to Make the Attention Heatmap More Concentrated

Across a large set of attention heatmaps, we consistently observe a pronounced dispersed pattern, as shown in Fig. 2a: under the standard block-sparse granularity, most blocks contain only a small number of activated weights. From a block-level perspective, almost every block appears somewhat important, so even when block sparsification identifies relatively important blocks, substantial intra-block



(a) Original (b) PBS (c) Ours

Figure 2: **Attention heatmaps under different permutation strategies. (a) Original:** The heatmap exhibits abundant line-level (stripe-like) structures. **(b) PBS:** Following PBS's local K/V permutation strategy, the heatmap still contains substantial redundancy and fails to consistently emphasize salient horizontal stripes. **(c) Ours:** Our global permutation scheme (Sec. 3.1) compacts attention mass into a progressive region from dense (upper-left) to increasingly diffuse.More qualitative heatmaps are provided in Appendix D.

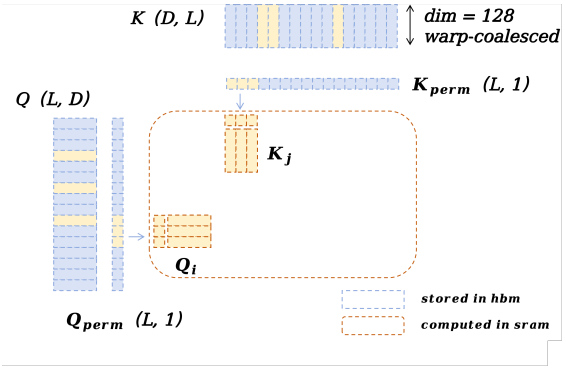


Figure 3: **Coordinate-scheduled online Q/K permutation.** Since the head dimension is typically 128, scattered token accesses can still fully utilize a warp; see Appendix B for details.

redundancy remains. In other words, the bottleneck is not whether we can quickly find the most important blocks, but that a fixed block granularity has limited capacity to represent intra-block sparsity. Therefore, rather than further debating which sparse blocks to select, it is more worthwhile to study how to reduce intra-block computation redundancy, so as to better exploit fine-grained sparsity within blocks.

Fine-grained stripes dominate the heatmap. As shown in Fig. 2a, attention patterns often exhibit finer-grained line-level structures, which have also been reported in prior work (Li et al., 2025). The heatmap is typically shaped by interleaved vertical stripes, slash-like stripes, and horizontal stripes, among which all three can be locally salient and highly concentrated. This phenomenon motivates three key premises for our method design: (i) **Local vertical stripes are salient**: positions on certain vertical stripes tend to receive higher attention scores; (ii) **Local horizontal stripes are salient**: similarly, certain horizontal stripes tend to yield higher attention scores; (iii) **Local slash-like stripes are salient**: likewise, certain slash-like stripe patterns can also carry high attention mass. Based on these observations, we perform mean pooling and importance estimation at the line level, rather than using fixed-size blocks as pooling units. Moreover, compared to the PBS strategy (Fig. 2b) that applies local permutation over K/V, our approach applies a global, index-guided permutation over both Q and K/V (Fig. 2c). This global permutation significantly concentrates attention mass toward the top-left region, yielding a more compact heatmap pattern. Interestingly, jointly permuting Q, K, and V further aligns and aggregates slash-like stripes, improving concentration. These observations motivate a fully global, index-guided online permutation to amplify concentration.

2.2 How to perform online permutation

Motivated by Section 2.1, we observe that global permutation can aggregate dispersed attention mass into a more concentrated structure; to this end, we adopt **online permutation**. We shift permutation from an *offline*, pre-attention tensor reordering to a *coordinate-addressed* loading procedure during computation. As illustrated in Fig. 3, we first construct two lightweight index arrays, Q_{perm} and K_{perm}, without moving tensors in memory. At runtime, these indices directly determine which tiles (e.g., Q_i and K_j) are materialized in on-chip SRAM for block-attention computation. Compared to offline local permutation, this strategy globally front-loads high-contribution information in the loading order, enabling earlier aggregation and improving sparsification benefits. Although token indices become non-contiguous, memory accesses within each token remain contiguous along the head-dimension, which preserves efficient GPU vectorized loads. In practice, the added overhead is typically below **10%** and is negligible compared to sparsity-induced speedups. In Appendix B, we detail how this design integrates with FlashAttention and benchmark its runtime overhead against

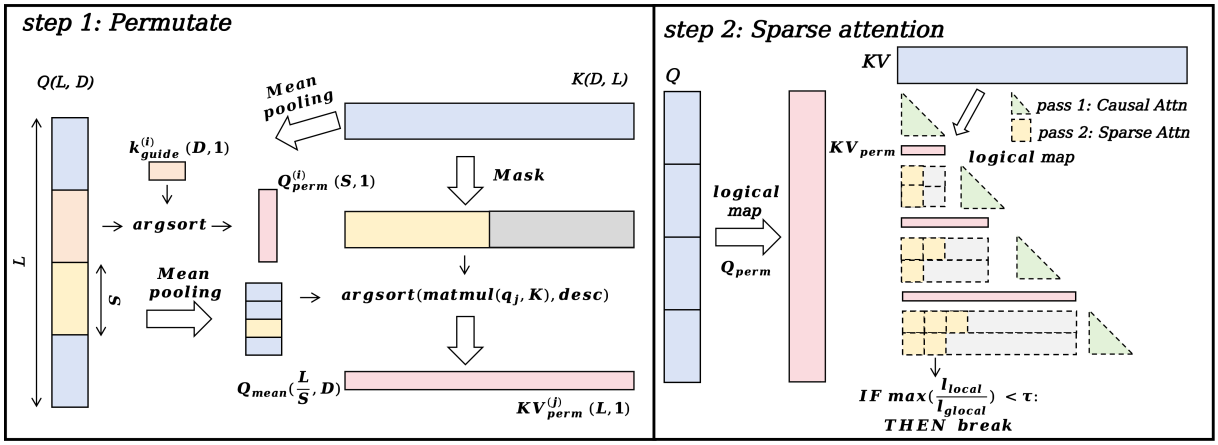


Figure 4: **S2O workflow. Step 1 (Permutate)** builds two lightweight index arrays without moving tensors: (i) an intra-segment query permutation index Q_{perm} (token-level, segment-local), and (ii) a prefix key/value logical map KV_{perm} (token-level, global indices) obtained by retrieving segment representatives. **Step 2 (Sparse attention)** runs attention in two passes: *Pass-1* computes a dense intra-segment causal window to initialize online-softmax states; *Pass-2* resumes the states and processes the historical prefix in the retrieved order with early stopping.

block-granularity sparse attention.

3 Method

Motivated by Section 2, we leverage a key empirical pattern in long-context attention: high-heat regions are often structured but dispersed, and permutation can rapidly concentrate them into a small set of positions. If we compute these high-contribution positions first, the online-softmax accumulator collects most attention mass early, enabling adaptive sparsification under a fixed approximation budget.

As shown in Figure 4, our method decomposes sparse attention into two steps: **(1) permute**—generate index arrays that permute the computation order for both Q and historical K/V; **(2) sparse attention**—run a FlashAttention-style kernel that gathers tokens by indices (no physical permutation), and skips low-gain blocks via early stopping. Crucially, we preserve the original tensor layout and memory access pattern; only lightweight index tensors are introduced.

3.1 Step 1: Intra-Segment Lightweight Ranking

As L grows, a direct token-level permutation over the full sequence becomes prohibitively expensive. We therefore introduce segments to strike a better balance between benefit and overhead. Let the sequence be partitioned into $N = L/S$ segments of length S. Within each segment, we apply a cheap, index-based permutation of Q to front-load queries that are more likely to induce prominent horizontal stripes; then we use a segment representative to score and rank historical keys for that segment, producing a logical K/V map and front-loading positions that are more likely to form salient vertical stripes within the segment.

Algorithm 1 Intra-Segment Lightweight Ranking

Input: $Q, K \in \mathbb{R}^{Z\times H\times L\times D}$; segment length S
Output: Q_{perm}, KV_{perm};
 # partition the sequence into N segments
$N \leftarrow L/S$; split $Q, K \rightarrow \{(Q_n, K_n)\}_{n=0}^{N-1}$
 # compute segment representatives
for $n = 0$ **to** $N-1$ **do**
  $q_{\text{mean}}[n] \leftarrow \text{Mean}(Q_n)$; $k_{\text{mean}}[n] \leftarrow \text{Mean}(K_n)$
end for
 # (i) permute Q within each segment (cheap scoring + ranking)
$k_{\text{guide}} \leftarrow k_{\text{mean}}[0]$
for $n = 0$ **to** $N-1$ **do**
  $s_Q[n, s] \leftarrow \langle Q_n[s],\ k_{\text{guide}}\rangle$
  $Q_{\text{perm}}[n] \leftarrow \text{Argsort}(s_Q[n, :],\ \text{desc})$
end for
 # (ii) permute historical K/V per segment (causal-prefix ranking)
for $n = 0$ **to** $N-1$ **do**
  $s_K[n, t] \leftarrow \langle q_{\text{mean}}[n],\ K[t]\rangle$
  $s_K[n,\ t \geq nS] \leftarrow -\infty$ ▷ causal prefix only
  $KV_{\text{perm}}[n] \leftarrow \text{Argsort}(s_K[n, :],\ \text{desc})$
end for
return Q_{perm}, KV_{perm}

(i) Segment-wise filtering of Q. We reshape $Q \in \mathbb{R}^{Z\times H\times L\times D}$ into $Q \in \mathbb{R}^{Z\times H\times N\times S\times D}$, where $N = L/S$. For each segment, we assign every query token a coarse importance score using a lightweight guide vector. To keep the overhead minimal, we reuse a fixed guide shared across segments (e.g., the representative key of the first

Algorithm 2 Pass-1: Dense Intra-Segment Causal Init

Input: Q, K, V; segment length S; tile sizes (B_M, B_N)
Output: buffers $(A, Lbuf, Mbuf)$ storing (acc, ℓ, m) per query tile
 # iterate over segments and heads
for $n = 0$ **to** $N - 1$ **do**
  for $h = 0$ **to** $H - 1$ **do**
 # scan queries within segment n in B_M tiles
    for all each query tile b in segment n **do**
      $q \leftarrow Q(n, h, b)$
      $(m, \ell, acc) \leftarrow (-\infty, 0, 0)$
 # scan keys/values within the same segment in B_N tiles
      for all each key tile t in segment n **do**
        $k \leftarrow K(n, h, t)$; $v \leftarrow V(n, h, t)$
        $\mathcal{M} \leftarrow \textsc{CausalMask}(q \text{ tile } b,\ k \text{ tile } t)$
        $(m, \ell, acc) \leftarrow \textsc{Softmax}(q, k, v;\ \mathcal{M})$
      end for
      $A(n, h, b) \leftarrow acc$; $Lbuf(n, h, b) \leftarrow \ell$;
      $Mbuf(n, h, b) \leftarrow m$
    end for
  end for
end for
return $(A, Lbuf, Mbuf)$

Algorithm 3 Pass-2: Coordinate-Scheduled Sparse Attention (Early Stopping)

Input: Q, K, V; buffers $(A, Lbuf, Mbuf)$ (Alg. 2); $Q_{\text{perm}}, KV_{\text{perm}}$ (Alg. 1); segment length S; local window $W \le S$; threshold τ
Output: O
for $n \leftarrow 0$ **to** $N - 1$ **do**
 # permute computation order via indices
  $\tilde{Q}_n \leftarrow \textsc{Gather}(Q_n,\ Q_{\text{perm}}[n])$
  $B_n \leftarrow (A_n, Lbuf_n, Mbuf_n)$
  $(acc, \ell, m) \leftarrow \textsc{Gather}(B_n,\ Q_{\text{perm}}[n])$
 # traverse prefix tiles in $KV_{\text{perm}}[n]$ with early stopping
  for $T \in KV_{\text{perm}}[n]$ **do**
    $s \leftarrow (m, \ell, acc)$
    $s' \leftarrow \textsc{Softmax}(\tilde{Q}_n,\ K[T],\ V[T],\ s)$
    $(m', \ell', acc') \leftarrow s'$
    if $\Delta\ell \leftarrow \ell' - \ell;\ \Delta\ell < \tau \cdot \ell$ **then**
      break
    end if
    $(m, \ell, acc) \leftarrow (m', \ell', acc')$
  end for
  $\tilde{O}_n \leftarrow acc/\ell$
  $O_n \leftarrow \textsc{Scatter}(\tilde{O}_n,\ Q_{\text{perm}}[n])$
end for
return O

segment):

$$s_Q[z, h, n, s] = \big\langle Q[z, h, n, s, :],\ k_{\text{guide}}[z, h, :] \big\rangle. \tag{1}$$

We then compute the intra-segment permutation $Q_{\text{perm}} = \text{Argsort}(s_Q, \text{desc})$, which yields segment-local offsets in $[0, S)$ and requires no tensor movement.

(ii) Segment-wise ranking of historical K/V. We next form a representative query for each segment by mean pooling: $q_{\text{mean}}[z, h, n, :] = \text{Mean}_{s \in [0,S)}\big(Q[z, h, n, s, :]\big)$. Using this representative, we score all historical keys:

$$s_K[z, h, n, t] = \big\langle q_{\text{mean}}[z, h, n, :],\ K[z, h, t, :] \big\rangle. \tag{2}$$

To respect causality, we mask out non-prefix positions so that segment n only attends to $[0, nS)$: $s_K[z, h, n, t \ge nS] \leftarrow -\infty$. Ranking s_K in descending order yields a logical causal-prefix permutation KV_{perm}, i.e., an ordered list of absolute K/V indices for each segment.

3.2 Step 2: Online permutation and early skipping

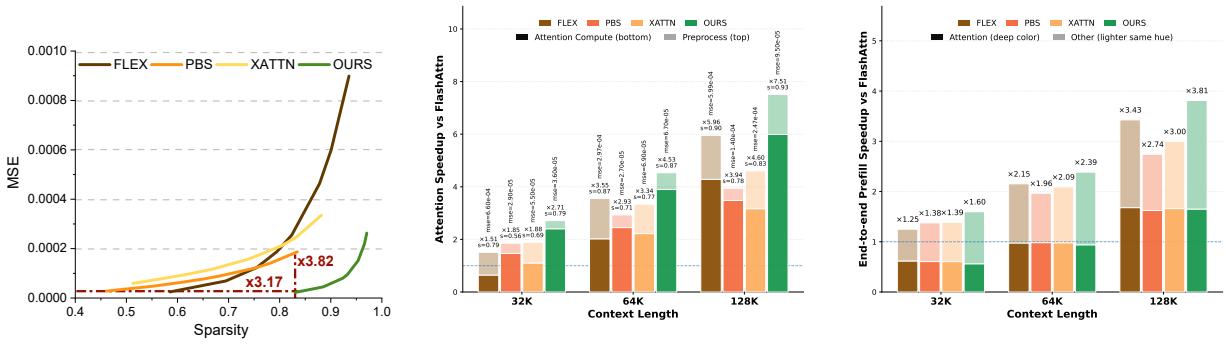
Online permutation via coordinate-indexed loading. Rather than physically permuting tensors, we permute the computation order by supplying index arrays to the attention kernel. We only materialize two lightweight index tensors: Q_{perm}, which encodes intra-segment query offsets, and KV_{perm}, which encodes the causal-prefix K/V indices. During attention, the kernel gathers the corresponding $Q/K/V$ tiles according to these indices, executes block-wise computation in the permuted order, and scatters the outputs back to their original positions.

Monotone-gain early stopping. In contrast to Top-K or Top-CDF sampling that pre-commits to a fixed subset of blocks, we adopt an online early-stopping rule. Following the order given by KV_{perm}, we track the marginal attention-mass gain contributed by each processed prefix block. Once the gain falls below a threshold τ, we stop early and skip the remaining low-contribution prefix blocks, avoiding further processing of many negligible candidates without introducing an additional complex block-retrieval procedure.

3.3 Kernel implementation

We implement Step 2 on top of a FlashAttention-style kernel. Let (m, ℓ, acc) denote the per-query online-softmax states: running maximum m, normalization accumulator ℓ, and output accumulator acc. To avoid permutation conflicts introduced by the autoregressive causal mask, and to ensure numerical stability, the kernel runs in two passes (Figure 4).

Pass 1: Dense Intra-Segment Causal Init. For each segment, we first compute a small dense causal window within the segment (Algorithm 2).



(a) Mse vs. Sparsity. (b) Attention Speedup. (c) E2e Speedup.

Figure 5: Evaluation overview on **Llama-3.1-8B**. (a) Operator-level accuracy under matched sparsity. (b) Attention speedup: dark bars denote attention time and light bars denote preprocessing time; the corresponding sparsity(s) and MSE are also reported. (c) End-to-end prefill latency breakdown across context lengths: dark bars denote total attention-related time (including preprocessing) and light bars denote all remaining overheads.

This is because, under the autoregressive causal mask, permutation may bring originally masked-out positions forward, making dense-block computation and scheduling more complicated; the intra-segment dense window initializes stable online-softmax states for each query token and caches them in lightweight buffers $(A, Lbuf, Mbuf)$.

Pass 2: Coordinate-Scheduled Sparse Attention (Early Stopping). We then permute queries inside the segment according to Q_{perm}, gather the corresponding Q tokens, and resume states from $(A, Lbuf, Mbuf)$. Next, we traverse the causal prefix keys in the order specified by KV_{perm} using tiles of size B_N, updating (m, ℓ, acc) with the standard online-softmax recurrence. After each tile, we estimate the marginal gain in the normalization mass ℓ; if $\Delta\ell$ falls below a fraction of the accumulated ℓ, we terminate early. Finally, we normalize $o = acc/\ell$ and scatter the outputs back to their original token positions.

4 Experiments

4.1 Experimental Setup

Models. We evaluate on two widely used open-source LLMs, **Qwen3-8B** (Yang et al., 2025a) and **Llama-3.1-8B** (Grattafiori et al., 2024). They are representative of mainstream deployments and capture typical attention-heatmap behaviors observed in modern LLMs.

Baselines. We compare our method against: (i) **Full** (implemented with **FlashAttention2** (Dao, 2024)), used as the dense-attention reference; (ii) **FlexPrefill** (Lai et al., 2025): it dynamically determines a per-head sparse pattern by comparing estimated vs. true score distributions, representing pattern-based block selection; (iii) **XAttn** (Xu et al., 2025): a training-free block-sparse method that scores blocks via an anti-diagonal probing pattern, representing a fine-grained block selection strategy; and (iv) **PBS** (Wang et al., 2025): it leverages attention's permutation property and performs segment-wise token permutation to concentrate high-mass regions, representing a local permutation strategy.

Implementation details. All experiments are conducted in BF16 on NVIDIA GPUs. We implement our method in Triton following a FlashAttention-style kernel structure. Unless otherwise specified, we use segment length S=2048, early-stop threshold τ=0.005, and fix the attention block size to $(B_M, B_N) = (128, 128)$ for fair comparison across methods. Latency is measured using CUDA events, and we report the mean over multiple post-warmup iterations. MSE/MAE are computed as the mean error averaged across all attention heads. All baselines use the default configurations reported in their papers (see Appendix C for details).

4.2 Main Results

Single-operator and end-to-end performance. We tune each method's hyperparameters to control its attained sparsity (i.e., compute density), and compare the resulting approximation error under approximately matched sparsity. To quantify operator-level approximation quality, we introduce mean squared error (MSE) as a direct metric for sparse-attention error: a lower MSE indicates that the attention output is closer to the full-attention reference. As shown in Fig. 5a, at matched spar-

Methods	En.Sum	En.QA	En.MC	En.Dia	Zh.QA	Code.Debug	Math.Find	Retr.PassKey	Retr.Number	Avg.
Llama-3.1-8B										
Full	0.167	0.187	0.271	0.135	0.132	0.241	0.331	0.993	0.993	0.383
	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
FlexPrefill	0.167	0.178	0.245	0.125	0.113	0.218	0.328	**1.000**	0.988	0.374
	0.8591	0.8523	0.8534	0.8416	0.8480	0.8376	0.8682	0.8760	0.8510	0.8541
XAttn	0.180	0.165	0.223	**0.135**	0.080	0.234	0.329	0.998	0.995	0.371
	0.7106	0.7132	0.7128	0.7247	0.6831	0.7303	0.7075	0.6892	0.6856	0.7063
PBS	**0.179**	**0.192**	0.245	0.115	**0.127**	0.228	0.329	0.831	0.992	0.360
	0.7539	0.7578	0.7570	0.7709	0.7292	0.7766	0.7513	0.7415	0.7376	0.7529
Ours	0.170	0.166	**0.271**	0.120	0.110	**0.244**	**0.343**	0.986	**0.998**	**0.379**
	0.9178	**0.9220**	**0.9225**	**0.9150**	**0.9106**	**0.9096**	**0.9154**	**0.9390**	**0.9382**	**0.9211**
Qwen3-8B										
Full	0.180	0.022	0.489	0.275	0.038	0.185	0.317	1.000	0.969	0.386
	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
FlexPrefill	0.149	0.019	0.450	0.160	**0.039**	0.170	**0.340**	0.998	0.990	0.368
	0.8250	0.8266	0.8494	0.8329	0.8328	0.8581	0.8164	0.8181	0.8237	0.8314
XAttn	0.164	0.019	0.415	0.165	0.037	0.200	0.306	1.000	0.810	0.346
	0.6631	0.6701	0.6697	0.6914	0.6669	0.7147	0.6189	0.6229	0.6212	0.6599
PBS	**0.180**	0.019	**0.476**	**0.305**	0.038	0.172	0.337	1.000	**1.000**	**0.392**
	0.7730	0.7468	0.7473	0.7649	0.7424	0.7806	0.7098	0.7175	0.7209	0.7504
Ours	0.175	**0.020**	0.445	0.280	0.038	**0.200**	0.323	**1.000**	0.997	0.386
	0.9085	**0.9089**	**0.8980**	**0.8669**	**0.8663**	**0.8745**	**0.8736**	**0.8554**	**0.9033**	**0.8839**

Table 1: **InfiniteBench results.** Each cell shows Accuracy (top) and Prefill-stage sparsity ratio (bottom) under identical decoding settings. **Bold** indicates the sparsity strategy with the highest score for each task.

sity, our method consistently achieves substantially lower MSE than all baselines. On **Llama-3.1-8B** with a **128K** context, compared to the prior state-of-the-art strategy, our method reduces MSE by up to **3.82**× at matched sparsity, and reduces the required compute density by up to **3.17**× at matched MSE, demonstrating superior single-operator quality.

Beyond the sparse-attention operator itself, sparse preprocessing (e.g., retrieval/indexing) may introduce non-negligible overhead in practical systems. We therefore profile the end-to-end prefill latency under the default configuration across different sequence lengths, and decompose the total time into (i) attention-operator time, (ii) sparse-preprocessing time, and (iii) end-to-end prefill time. As shown in Table 8, under long-context settings, our preprocessing overhead accounts for only a small fraction of the total time and is dominated by a single lightweight ordering step. Moreover, Fig. 5b, Fig. 5c and Fig. 1 show consistent speedups across a wide range of context lengths, indicating that operator-level improvements (as measured by MSE) effectively translate into system-level gains under realistic workloads. In particular, under a **128K** context on **Llama-3.1-8B**, our method achieves **7.53**× operator-level attention speedup and **3.81**× end-to-end speedup.

Benchmark results. We focus on long-context benchmarks and evaluate accuracy on RULER, LONGBENCH V2, and INFINITEBENCH. The main results are summarized in Tab. 1 (INFINITEBENCH), Tab. 7 (RULER (Hsieh et al., 2024)), and Tab. 9 (LONGBENCH V2 (Bai et al., 2024)). Due to space constraints, additional results are deferred to Appendix C.1. In particular, Tab. 7 shows that on RULER, our method maintains accuracy across a wide range of context lengths while achieving higher sparsity. Overall, our method attains accuracy comparable to FULL-ATTN across all benchmarks, while reducing the prefill compute density by **2–3**× compared to the original strategy, thereby providing a higher-ceiling path for sparse-attention design.

4.3 Ablation Study

Hyperparameter ablations. We conduct hyperparameter ablations on **Qwen3-8B** under a **128K** context by varying the segment length S and the early-stop threshold τ, and report their impacts on sparsity as well as mean squared error (MSE) and mean absolute error (MAE). Overall, S has only a minor effect on approximation error, whereas τ is the dominant factor governing the speed–accuracy trade-off. Based on these results, we use the recommended S in the table to balance retrieval overhead, and adjust τ to trade accuracy for sparsity gains.

Component ablation. To quantify the contribution of key design choices, we ablate two core

Hyperparameter Ablations (128K)				
S	τ	Sparsity↑	avg. MAE↓	avg. MSE↓
2048	0.001	0.721	0.00536	0.00058
2048	0.002	0.798	0.00873	0.00131
2048	0.004	0.861	0.01353	0.00265
2048	0.005	0.877	0.01535	0.00323
2048	0.010	0.918	0.02163	0.00560
2048	0.020	0.944	0.02849	0.00876
1024	0.005	0.887	0.01519	0.00322
2048	0.005	0.877	0.01535	0.00323
4096	0.005	0.862	0.01520	0.00312

Table 2: **Hyperparameter ablations.** We report mean sparsity and the average approximation error (MAE/MSE, averaged over heads) on Qwen3-8B at 128K. Top block fixes S=2048 and varies τ; bottom block fixes τ=0.005 and varies S.

Method	Sparsity ↑	MAE ↓	MSE ↓
LOCAL WINDOW ONLY	0.983	0.01790	1.8395×10^{-3}
OURS (W/O Q_{perm})	0.796	0.00133	1.86×10^{-5}
Ours (w. Q_{perm})	0.852	0.00129	1.87×10^{-5}

Table 3: **Component ablation.** We report mean sparsity and average approximation error (MAE/MSE) on Qwen3-8B at 128K, and additionally present one representative head.

components: **(i) the local dense window** for stabilizing attention near segment boundaries, and **(ii) query-side intra-segment permutation** Q_{perm}. We compare the full method with two variants (Table 3): *w/o* Q_{perm} (setting Q_{perm} to the identity order within each segment while keeping the same historical-prefix K/V ordering and early-stopping rule), and *local-window only* (retaining only dense local-window attention). On a subset of heads, enabling Q_{perm} consistently improves the speed–quality trade-off by front-loading queries that are more likely to induce horizontal stripes, allowing earlier mass accumulation and more effective early stopping at comparable error. In contrast, using the local dense window alone incurs large approximation error; its main role is to avoid special-case handling of masked positions near segment boundaries. For weaker stripe patterns, we also provide a simplified variant that performs no Q reordering and fuses pass-1 and pass-2 into a single computation (see Alg. 4). Since this variant has nearly identical runtime to the full implementation, we ultimately adopt a unified Q_{perm} permutation strategy across all heads.

5 Related Work

Sparse attention. FlashAttention (Dao, 2024), inspired by online-softmax (Milakov and Gimelshein, 2018), reduces HBM traffic through blockwise loading and incremental accumulation, providing a practical foundation for block-sparse attention. Train-free sparse attention restricts computation to structured regions without retraining, including local-window/global-token schemes (Xiao et al., 2023; Beltagy et al., 2020), pattern-based block sparsity (Jiang et al., 2024; Li et al., 2025; He et al., 2025; Lai et al., 2025), and improved block selection/sampling strategies (Tang et al., 2024; Xu et al., 2025; Zhang et al., 2025a; Gu et al., 2025). Moreover, prior work shows that reordering can concentrate attention mass and improve sparse access patterns (Xi et al., 2025; Yang et al., 2025b; Wang et al., 2025).

Efficient Transformers. **Quantization** reduces memory footprint and bandwidth via low-bit representations, improving inference throughput. Representative directions include activation-aware weight quantization (AWQ) (Lin et al., 2023), extreme compression with additive quantization (Egiazarian et al., 2024)(Liu et al., 2025), and combining quantization with sparsity for further acceleration (GQSA) (Zeng et al., 2025b); ABQ-LLM studies arbitrary-bit quantized inference to better trade accuracy for efficiency across hardware targets (Zeng et al., 2025a). **Caching** targets dominant runtime/memory bottlenecks by reusing intermediate states. For autoregressive LLMs, KV-cache quantization reduces cache memory and bandwidth (Liu et al., 2024a). For diffusion transformers (DiT), caching reuses redundant computation across timesteps, including layer caching (Ma et al., 2024)(Liu et al., 2024b), token-wise feature caching (Zou et al., 2025), adaptive caching for video generation (Kahatapitiya et al., 2024), and error-aware cache correction with timestep adjustment (Peng et al., 2025).

6 Conclusion

We present **S2O**, a fine-grained attention sparsification mechanism for long-context inference. S2O enables non-contiguous, importance-driven token loading and an online skip strategy that safely prunes low-contribution block under a controlled error budget, moving beyond conventional fixed block-granularity designs.

Limitations

First, the method introduces additional hyperparameters (e.g., segment length, thresholds, and scheduling rules), which may require re-tuning across models, context lengths, and hardware platforms, and the best configuration may not transfer reliably. Second, our evaluation primarily focuses on the prefill stage of decoder-only LLMs; extending the approach to other architectures (e.g., encoder–decoder and multimodal models) and to training-time usage requires further study. Finally, our experiments are conducted on a specific GPU and software stack; performance and memory behavior may vary across other accelerators and deployment environments.

References

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. 2024. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. *arXiv preprint arXiv:2412.15204*.

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

Tri Dao. 2024. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*.

Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. 2024. Extreme compression of large language models via additive quantization. *Preprint*, arXiv:2401.06118.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Youping Gu, Xiaolong Li, Yuhao Hu, Minqi Chen, and Bohan Zhuang. 2025. Blade: Block-sparse attention meets step distillation for efficient video generation. *Preprint*, arXiv:2508.10774.

Zhiyuan He, Yike Zhang, Chengruidong Zhang, Huiqiang Jiang, Yuqing Yang, and Lili Qiu. 2025. Trianglemix: Accelerating prefilling via decoding-time contribution sparsity. *Preprint*, arXiv:2507.21526.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*.

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. 2024. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Kumara Kahatapitiya, Haozhe Liu, Sen He, Ding Liu, Menglin Jia, Chenyang Zhang, Michael S. Ryoo, and Tian Xie. 2024. Adaptive caching for faster video generation with diffusion transformers. *Preprint*, arXiv:2411.02397.

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

Yucheng Li, Huiqiang Jiang, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Amir H Abdi, Dongsheng Li, Jianfeng Gao, Yuqing Yang, and Lili Qiu. 2025. MMIference: Accelerating pre-filling for long-context vlms via modality-aware permutation sparse attention. In *Forty-second International Conference on Machine Learning*.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. 2023. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv*.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. 2024a. Minicache: Kv cache compression in depth dimension for large language models. *Preprint*, arXiv:2405.14366.

Songwei Liu, Chao Zeng, Lianqiang Li, Chenqian Yan, Lean Fu, Xing Mei, and Fangmin Chen. 2024b. Foldgpt: Simple and effective large language model compression scheme. *arXiv preprint arXiv:2407.00928*.

Songwei Liu, Chao Zeng, Chenqian Yan, Xurui Peng, Xing Wang, Fangmin Chen, and Xing Mei. 2025. Error propagation mechanisms and compensation strategies for quantized diffusion. *arXiv preprint arXiv:2508.12094*.

Xinyin Ma, Gongfan Fang, Michael Bi Mi, and Xinchao Wang. 2024. Learning-to-cache: Accelerating diffusion transformer via layer caching. *Preprint*, arXiv:2406.01733.

Maxim Milakov and Natalia Gimelshein. 2018. Online normalizer calculation for softmax. *Preprint*, arXiv:1805.02867.

Xurui Peng, Hong Liu, Chenqian Yan, Rui Ma, Fangmin Chen, Xing Wang, Zhihua Wu, Songwei Liu, and Mingbao Lin. 2025. Ertacache: Error rectification and timesteps adjustment for efficient diffusion. *Preprint*, arXiv:2508.21091.

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

Xinghao Wang, Pengyu Wang, Dong Zhang, Chenkun Tan, Shaojun Zhou, Zhaoxiang Liu, Shiguo Lian, Fangxu Liu, Kai Song, and Xipeng Qiu. 2025. Sparser block-sparse attention via token permutation. *Preprint*, arXiv:2510.21270.

Haocheng Xi, Shuo Yang, Yilong Zhao, Chenfeng Xu, Muyang Li, Xiuyu Li, Yujun Lin, Han Cai, Jintao Zhang, Dacheng Li, and 1 others. 2025. Sparse videogen: Accelerating video diffusion transformers with spatial-temporal sparsity. *arXiv preprint arXiv:2502.01776*.

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

Ruyi Xu, Guangxuan Xiao, Haofeng Huang, Junxian Guo, and Song Han. 2025. Xattention: Block sparse attention with antidiagonal scoring. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025a. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Shuo Yang, Haocheng Xi, Yilong Zhao, Muyang Li, Jintao Zhang, Han Cai, Yujun Lin, Xiuyu Li, Chenfeng Xu, Kelly Peng, and 1 others. 2025b. Sparse videogen2: Accelerate video generation with sparse attention via semantic-aware permutation. *arXiv preprint arXiv:2505.18875*.

Chao Zeng, Songwei Liu, Yusheng Xie, Hong Liu, Xiaojian Wang, Miao Wei, Shu Yang, Fangmin Chen, and Xing Mei. 2025a. Abq-llm: Arbitrary-bit quantized inference acceleration for large language models. *Preprint*, arXiv:2408.08554.

Chao Zeng, Songwei Liu, Shu Yang, Fangmin Chen, Lean Fu, and Xing Mei. 2025b. Gqsa: Group quantization and sparsity for accelerating large language model inference. *Preprint*, arXiv:2412.17560.

Jintao Zhang, Chendong Xiang, Haofeng Huang, Jia Wei, Haocheng Xi, Jun Zhu, and Jianfei Chen. 2025a. Spargeattn: Accurate sparse attention accelerating any model inference. In *International Conference on Machine Learning (ICML)*.

Yu Zhang, Dong Guo, Fang Wu, Guoliang Zhu, Dian Ding, and Yiming Zhang. 2025b. AnchorAttention: Difference-aware sparse attention with stripe granularity. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 8548–8560, Suzhou, China. Association for Computational Linguistics.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, and 3 others. 2025. A survey of large language models. *Preprint*, arXiv:2303.18223.

Chang Zou, Xuyang Liu, Ting Liu, Siteng Huang, and Linfeng Zhang. 2025. Accelerating diffusion transformers with token-wise feature caching. *Preprint*, arXiv:2410.05317.

Appendix

A LLM Usage

Large Language Models were used solely to refine the manuscript's language, including sentence rephrasing, grammar checking, and improving readability. The LLM was not involved in ideation, methodology, experimental design, or data analysis. All scientific content, concepts, and analyses were developed by the authors, who take full responsibility for the manuscript. The LLM's role was limited to linguistic polishing, with strict adherence to ethical guidelines and avoidance of plagiarism or scientific misconduct.

B Permutation Overhead Analysis

B.1 Why Online Permutation Adds Negligible Overhead

A natural concern of permutation-based sparsification is the potential overhead introduced by (i) explicit permutation, (ii) extra data movement, and (iii) irregular memory access that may reduce GPU efficiency. In this appendix, we explain why our online permutation incurs negligible overhead in practice.

No physical permutation: permutation is implemented as index-driven loading. We do **not** physically permute $Q/K/V$ in memory, nor do we materialize permuted tensors. Instead, permutation is converted into a coordinate-driven loading policy: the attention kernel reads $Q/K/V$ from their original storage, but uses permuted indices for gather-style addressing and directly computes

under the permuted schedule (Fig. 3). This eliminates the cost of explicit tensor permutation and large-scale memory copies.

Metadata is lightweight and bandwidth-negligible. Online permutation only generates lightweight metadata (e.g., row indices for Q and selected column indices for KV). Compared to the numeric $Q/K/V$ tensors, the index payload is tiny. For example, with $\texttt{head\_dim} = 128$, a typical K/V tile load can be as large as $2 \times (128, 128)$ (for K and V), while the extra index read is only $(1, 128)$ in scale, leading to negligible bandwidth overhead.

Even with non-consecutive token indices, the underlying loads can remain regular and efficient. A GPU thread typically issues vectorized loads at a 128-bit (16 B) granularity; for BF16 (2 B/elem), each load covers 8 consecutive elements. Hence, for a token feature of dimension D=64, reading 64 BF16 elements amounts to 8 such vector loads, which can be viewed as 8 threads collaboratively fetching one token (each responsible for 8 contiguous elements). A warp contains 32 threads and can therefore cover roughly $32/8$=4 tokens in parallel. Crucially, these tokens do not need to be adjacent in memory: as long as each token is stored contiguously and aligned along its feature dimension, the hardware still observes vectorized, coalesced transactions. Therefore, our offline token permutation does not require physical token contiguity; by preserving intra-token contiguity, it adheres to the vectorized/coalesced access principles and effectively utilizes memory bandwidth.

The compute pipeline is unchanged: only iteration order and mapping change. Our kernel follows the FlashAttention execution pattern: for each Q tile, we iterate over selected K/V tiles, load them into SRAM/shared memory, and accumulate softmax statistics online. Permutation only changes the tile iteration order and the mapping from logical tile ids to physical offsets. The load–compute–accumulate pipeline and tiling configuration remain unchanged, introducing no extra synchronizations or additional passes.

Empirical validation via ABCD decomposition. To precisely quantify the overhead of each component in our sparse attention pipeline, we design an ABCD decomposition benchmark. Starting from the dense FlashAttention kernel (Method A), each subsequent variant adds exactly one new mechanism, isolating its marginal cost:

Method	N=8192	N=16384	N=32768	N=65536	N=131072
A: Dense	3.38	11.95	44.67	175.40	717.49
B: Indexed seq	3.65	12.63	47.33	186.56	759.61
C: Indexed sorted	3.65	12.65	47.20	184.34	744.75
D: EarlyStop(off)	3.93	13.73	51.31	200.10	807.92

Table 4: **ABCD decomposition (Causal).** Kernel-only latency (ms) on A100, bf16, B=1, H=32, D=128, seg_len= 1024. "vs A" = A kernel time / this kernel time.

Method	N=8192	N=16384	N=32768	N=65536	N=131072
A: Dense	5.89	22.27	86.29	339.17	1344.97
B: Indexed seq	6.26	23.43	91.04	357.93	1421.07
C: Indexed sorted	6.24	23.53	91.02	358.14	1425.71
D: EarlyStop(off)	6.81	25.33	98.09	385.92	1565.86

Table 5: **ABCD decomposition (Non-causal).** Same setup as Table 4.

- **A (Dense):** Standard FlashAttention-2 tiled attention with contiguous memory access.
- **B (Indexed sequential):** Replaces contiguous tile loading with index-driven gather (`cp_async` indexed load). The indices are the identity sequence $[0, 1, 2, \ldots]$, so the physical K/V access order is preserved; this isolates the overhead of adding the index-based addressing layer itself (the extra index-tensor load and the `cp_async` indexed-mode instruction path), independent of any access-pattern irregularity.
- **C (Indexed sorted):** Identical to B except that the indices are pre-sorted by importance. Compared with B's identity order, C's indices induce a genuinely non-contiguous K/V access order, so the B→C delta directly isolates the kernel-side cost of non-contiguous K/V access.
- **D (Early-stop logic):** Adds the deferred early-stop ratio check (computed after $P \cdot V$ GEMM, checked at the start of the next iteration) on top of C, measuring the overhead of the early-stopping control flow.

Kernel provenance and controlled overhead. Kernel A is built directly from the official FlashAttention-2 repository, while B, C, and D are implemented as minimal, strictly additive modifications on top of A, sharing the same tile sizes, warp/stage configuration, and online-softmax update. Therefore, the latency difference between any two adjacent variants can be attributed to exactly one newly introduced mechanism. All variants are run with $\texttt{threshold} = -1$ (i.e., no actual skipping),

Transition	Causal	Non-causal	Description
A → B	∼5–8%	∼5–6%	Indexed gather vs. contiguous tiled copy
B → C	∼0%	∼0%	Sorted gather vs. sequential gather
C → D	∼8–9%	∼8–10%	Deferred early-stop ratio check
A → D (total)	**∼13–16%**	**∼14–16%**	**Full S2O machinery overhead**

Table 6: **Per-component marginal overhead** (kernel only, relative to the previous variant).

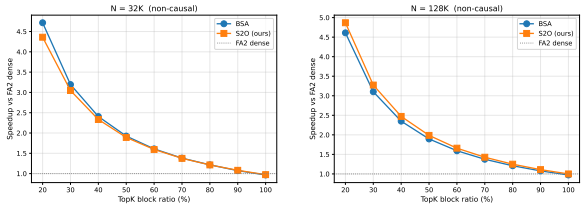


Figure 6: **Comparison with BSA under matched sparsity.** Speedup over FA2 dense under different TopK block ratios for non-causal attention. Left: sequence length $N = 32$K. Right: sequence length $N = 128$K. Under matched sparsity settings, S2O achieves runtime comparable to BSA, and the measured speedup remains largely aligned with the actual sparsity level. This suggests that sparsity beyond block granularity is practical and can be effectively translated into wall-clock acceleration.

so they perform identical FLOPs as dense attention. Hence, any latency difference comes purely from the added mechanism itself. As shown in Tables 4, 5, and 6, the extra overhead introduced by S2O is overall small and well controlled: indexed gather adds only about 5–8% overhead over dense attention, sorting is nearly negligible, and the deferred early-stop logic contributes about another 8–10%. Overall, even in the worst case without any actual skipping, the full S2O machinery increases kernel latency by only about 13–16%, indicating that the added mechanisms are lightweight and their implementation overhead is controllable in practice.

Comparison with BSA under matched sparsity. As shown in Fig. 6, we compare S2O and BSA under matched sparsity settings at two sequence lengths, with $N = 32$K on the left and $N = 128$K on the right. Specifically, we adopt a top-k selection strategy based on Kernel C and tune both methods to operate at approximately the same effective sparsity, so that the comparison is not confounded by different amounts of computation removed. Across both sequence lengths, S2O achieves runtime comparable to BSA, while the measured speedup remains largely aligned with the actual sparsity level. These results suggest that going beyond block-level sparsity is feasible in practice: finer-grained sparsity does not introduce noticeable extra overhead, and can still be effectively translated into wall-clock acceleration. In the following, we further analyze in DiT the additional gains enabled by lower sparsity granularity.

B.2 Why Online Permutation Integrates Seamlessly with FlashAttention

FlashAttention recap: tile streaming with an online softmax. FlashAttention accelerates attention by streaming over K/V in tiles and keeping only a small working set on-chip. Concretely, for each query tile Q_i, the kernel iterates over a sequence of key/value tiles $\{(K_j, V_j)\}$, loads each tile into SRAM/shared memory, computes the partial score block $S_{ij} = Q_i K_j^\top$, and updates the output using an online softmax with running statistics (e.g., row-wise max and normalizer) before moving to the next tile. This load–compute–accumulate pipeline is highly structured: performance comes from (i) fixed-shape tiling, (ii) aligned/coalesced global-memory transactions per tile, and (iii) a single-pass streaming update without materializing the full $L \times L$ attention matrix.

Where permutation traditionally hurts: physical permutation and extra passes. A straightforward way to apply permutation is to physically permute $Q/K/V$ into new contiguous layouts, or to scatter/gather them at very fine granularity. Such implementations introduce extra kernels, global-memory copies, and/or irregular accesses that break the tightly optimized FlashAttention dataflow. In practice, the overhead typically comes from *changing the data layout* rather than changing the tile traversal order.

Our key idea: replace physical permutation with coordinate scheduling. Our online permutation does not modify the FlashAttention compute pipeline. Instead, it replaces the **tile enumeration policy**:

- **Baseline FlashAttention:** visit tiles in a canonical order (e.g., increasing j) with contiguous offsets.
- **Online permutation:** visit tiles in a permuted schedule determined by lightweight index arrays (e.g., Q_{perm}, K_{perm}), and map each logical tile id to its physical offset via index-addressed gathers (Fig. 3).

Crucially, within each visited tile, we still load contiguous fragments of $Q/K/V$ with the same alignment as FlashAttention. Thus, the GPU observes

regular tile-shaped memory transactions, while the logical computation order is globally permuted.

What is changed vs. unchanged. Online permutation changes only: (i) the iteration order of (K_j, V_j) tiles for a given Q_i (and optionally the ordering of Q_i tiles), and (ii) the address mapping from a logical tile id to a physical base pointer. Everything else remains identical to FlashAttention: the tile sizes, the number of stages/warps, the single-pass online softmax update, and the synchronization structure. As a result, online permutation can be integrated as a thin addressing layer on top of a FlashAttention-style kernel, preserving its efficiency while enabling global, index-guided prioritization of high-contribution tiles.

C Additional Experimental Details

C.1 Baseline hyperparameters

Unified settings. To ensure fair comparisons, all methods are evaluated with the same model weights, maximum context length, batch size, dtype, attention backend , and decoding configuration. Whenever applicable, we use the same input samples and random seeds across methods. When reporting sparsity and approximation error across different context lengths, we use the first example in LongBench v2 and construct each length setting via truncation.

PBS-Attn. We follow the default PBS-Attn configuration: block size $B = 128$, segment size $S = 256$, and a block-selection threshold of 0.9 in all experiments. We also keep the default block grouping/selection procedure as in the original implementation.

FlexPrefill. We set the sparse-pattern threshold $\tau = 0.1$ for all models. We use the head-wise budget controller γ to adapt the compute budget online, and set $\gamma = 0.9$ by default.

X-Attn. We use the default X-Attn configuration with threshold $\tau = 0.9$ and stride 16 for all models.

RULER. Under the same settings as Table 7, we report the prefill-stage sparsity ratio and accuracy. RULER spans a wide range of context lengths, enabling a more comprehensive evaluation of how sparsification strategies behave and pay off in typical-length regimes. Our results show that even at shorter sequence lengths, our method still achieves high sparsity and consistently translates it into end-to-end speedups, indicating that the proposed coordinate scheduling and early-stopping mechanism remains effective beyond the ultra-long-context setting and provides a favorable benefit–overhead trade-off in practice.

Methods	8K	16K	32K	64K	128K	Avg.
Llama-3.1-8B						
Full	91.53	87.61	89.30	87.96	75.42	86.36
	0.000	0.000	0.000	0.000	0.000	0.000
FlexPrefill	86.89	86.51	81.86	76.35	72.28	80.78
	0.684	0.729	0.790	0.828	0.840	0.774
Xattn	91.10	88.62	87.90	84.67	**73.15**	85.09
	0.456	0.559	0.652	0.653	0.737	0.611
PBS	91.06	88.46	85.69	78.36	67.01	82.12
	0.356	0.468	0.560	0.668	0.747	0.560
Ours	**93.03**	**90.33**	**87.91**	**85.29**	72.40	**85.79**
	0.324	0.591	0.768	0.875	0.932	0.698
Qwen3-8B						
Full	85.77	81.81	81.40	73.66	69.50	78.43
	0.000	0.000	0.000	0.000	0.000	0.000
FlexPrefill	71.65	73.89	75.38	72.65	68.51	72.42
	0.675	0.704	0.752	0.777	0.826	0.747
Xattn	85.63	82.25	**81.60**	73.18	69.91	78.51
	0.458	0.534	0.609	0.668	0.737	0.601
PBS	85.56	79.34	80.95	70.70	67.90	76.89
	0.413	0.504	0.579	0.653	0.733	0.576
Ours	**85.80**	**82.73**	80.34	**73.95**	**69.97**	**78.56**
	0.191	0.440	0.644	0.796	0.893	0.593

Table 7: RULER accuracy (top) and prefill-stage sparsity ratio (bottom) across input lengths (8K–128K). Accuracy is averaged over tasks for each length; sparsity ratios are computed under identical decoding settings.

LongBench v2. On LongBench v2 (Table 9), our method continues to deliver strong accuracy while leveraging high prefill sparsity for acceleration, demonstrating robust generalization to a broader set of long-context tasks.

C.2 Additional details on end-to-end operator performance

Retrieval overhead vs. sequence length. Our operator consists of (i) a lightweight retrieval stage that computes segment-wise ranking indices (e.g., sorted key offsets and an optional query permutation), followed by (ii) the sparse attention computation. In our implementation, the dominant retrieval cost comes from scoring keys using per-segment query summaries, which scales approximately as $\mathcal{O}(L^2/S)$ where L is the sequence length and S is the segment length. The remaining parts (e.g., local query scoring and intra-segment sorting) contribute lower-order terms close to $\mathcal{O}(L)$ (and a small $\mathcal{O}(L \log S)$ sorting cost). Therefore, increasing S reduces retrieval overhead roughly inversely, while increasing L amplifies it superlin-

L	Retrieval (ms)				Flash latency (ms)
	S=512	S=1024	S=2048	S=4096	
4K	0.254	0.255	0.256	0.246	0.756
8K	0.374	0.311	0.266	0.315	1.720
16K	0.785	0.545	0.386	0.328	5.712
32K	2.415	1.410	0.920	0.636	21.09
64K	8.644	4.646	2.667	1.713	81.48
128K	33.238	16.906	9.017	5.122	320.2

Table 8: Measured retrieval latency (ms) as a function of segment length S and sequence length L, with Flash latency shown on the right.

Method	Llama-3.1-8B	Qwen3-8B
Full	30.22	34.19
FlexPrefill	26.44	30.21
Xattn	29.82	32.00
PBS	30.41	32.00
Ours	**31.41**	**32.60**

Table 9: LONGBENCH V2 average score.

early. In practice, we use a smaller segment length for short sequences to increase sparsity, and switch to a larger segment length as L grows to reduce retrieval time, since at ultra-long contexts the sparsity ratio is often already very high and the retrieval overhead can become non-negligible relative to the sparse attention computation itself. Table 8 reports the measured retrieval latency (ms) across (L, S).

C.3 Single-pass Variant

For completeness, we provide a variant that does not reorder Q and fuses Pass-1 (dense intra-segment causal initialization) and Pass-2 (coordinate-scheduled prefix traversal with early stopping) into a single streaming procedure. Unlike the two-pass implementation that materializes intermediate online-softmax states $(A, Lbuf, Mbuf)$ in (H)BM, the fused variant keeps the per-query states (m, ℓ, acc) in registers and updates them sequentially: (i) it first scans intra-segment causal tiles to initialize the states (equivalent to Pass-1), (ii) then computes a small dense local window for boundary stability, and (iii) finally traverses historical prefix tiles in the retrieved order with a monotone-gain early-stopping rule. See pseudocode in Alg. 4.

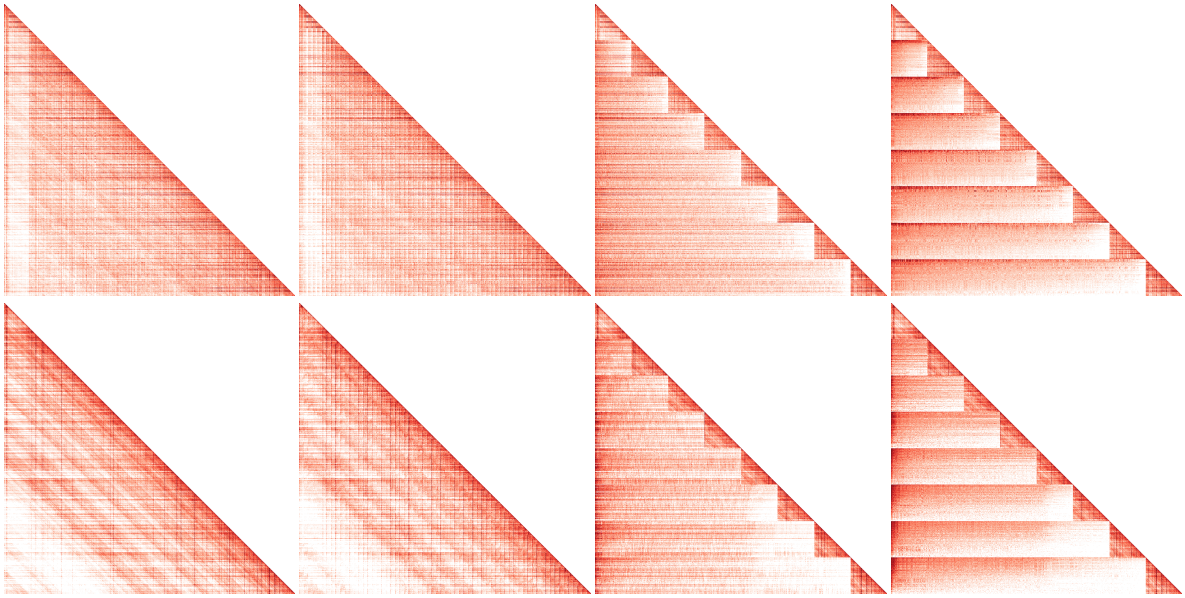
Algorithm 4 Fused Pass-1+Pass-2 (Single-pass, No Q Reordering): Coordinate-Scheduled Sparse Attention

Input: Q, K, V; KV_{perm} (Alg. 1); segment length S; local window $W \le S$; threshold τ; tile sizes (B_M, B_N)
Output: O (causal)

for $n \leftarrow 0$ **to** $N-1$ **do**
  for $h \leftarrow 0$ **to** $H-1$ **do**
    # scan queries in the original order, in B_M tiles
    for all each query tile b in segment n **do**
      $q \leftarrow Q(n,h,b)$
      $s \leftarrow (-\infty,\ 0,\ 0)$
        # (Pass-1) dense intra-segment causal init
      for all each key tile t in segment n **do**
        $k \leftarrow K(n,h,t)$; $v \leftarrow V(n,h,t)$
        $\mathcal{M} \leftarrow \textsc{CausalMask}(q \text{ tile } b,\ k \text{ tile } t)$
        $s \leftarrow \textsc{Softmax}(q,\ k,\ v;\ \mathcal{M},\ s)$
      end for
    # (Pass-2) prefix traversal in ranked order with early stopping
      for $T \in KV_{\text{perm}}[n]$ **do**
        $k_T \leftarrow K[T]$; $v_T \leftarrow V[T]$
        $s' \leftarrow \textsc{Softmax}(q,\ k_T,\ v_T;\ s)$
        $\Delta\ell \leftarrow s'.\ell - s.\ell$
        if $\Delta\ell < \tau \cdot s.\ell$ **then**
          break
        end if
        $s \leftarrow s'$
      end for
      $O(n,h,b) \leftarrow s.acc / s.\ell$
    end for
  end for
end for
return O

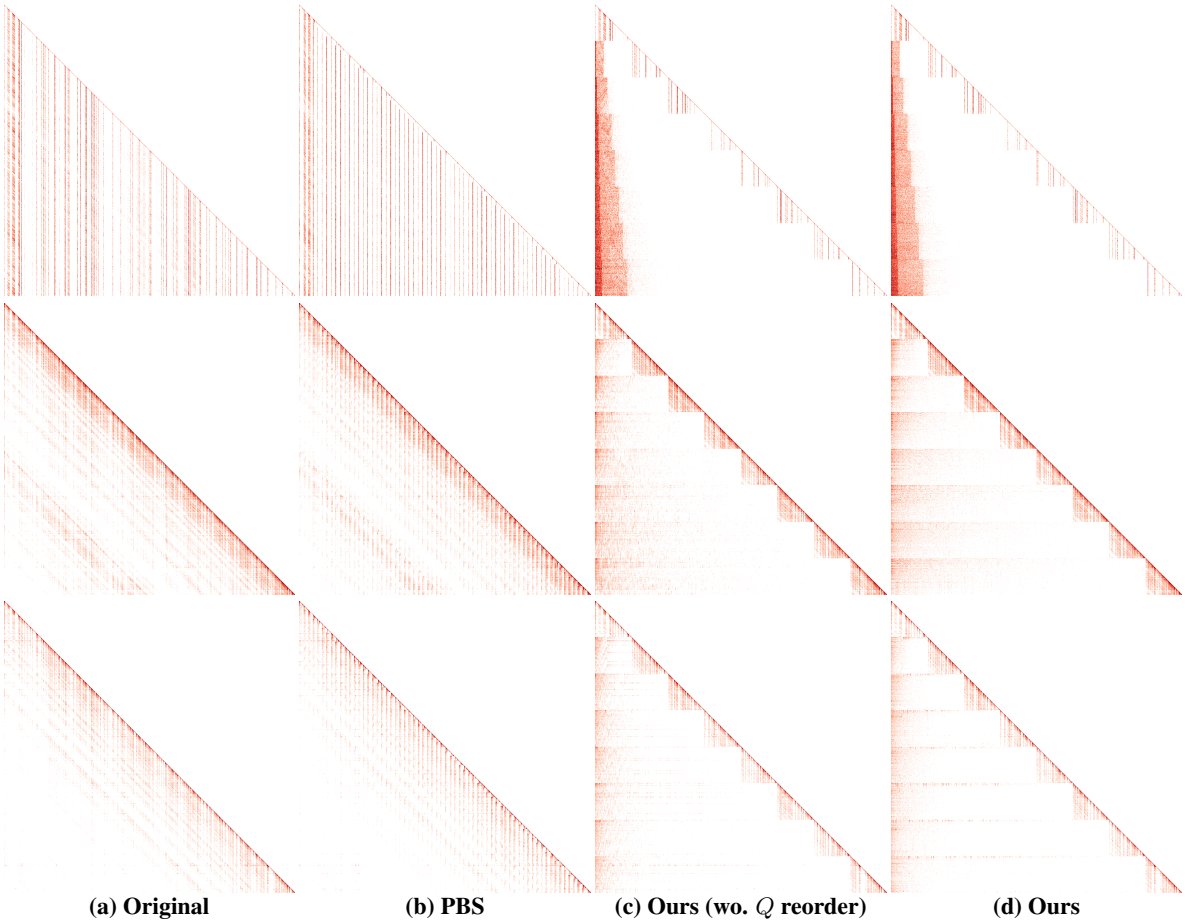
D Additional Attention Heatmaps

We provide additional qualitative attention heatmaps for two representative backbones: **LLaMA-3.1-8B** and **Qwen3-8B**. For each model, we compare four strategies: **Original**, **PBS** (local K/V permutation), **Ours (w/o Q permutation)** (global K/V permutation only), and **Ours** (global permutation of both Q and K/V), as shown in Fig. 7 and Fig. 8, respectively. These heatmaps indicate that across a broader set of layers/heads and inputs, our method consistently compacts salient stripe-like structures toward the upper-left region, demonstrating strong usability and consistency.



(a) Original **(b) PBS** **(c) Ours(wo. Q reorder)** **(d) Ours**

Figure 7: **Additional attention heatmaps on Qwen3-8B under different reordering strategies.**



(a) Original **(b) PBS** **(c) Ours (wo. Q reorder)** **(d) Ours**

Figure 8: **Additional attention heatmaps on LLaMA-3.1-8B under different reordering strategies.**