

# Improving Data Reuse across Blocks for Efficient Block-sparse Transformers on GPUs

Lihan Hu  
University of Iowa  
Iowa City, Iowa, USA  
lihan-hu@uiowa.edu

Xian-He Sun  
Illinois Institute of Technology  
Chicago, Illinois, USA  
sun@illinoistech.edu

Xiaoyang Lu  
Illinois Institute of Technology  
Chicago, Illinois, USA  
xlu40@illinoistech.edu

Peng Jiang  
University of Iowa  
Iowa City, Iowa, USA  
peng-jiang@uiowa.edu

## Abstract

Transformers are state-of-the-art models for many machine learning tasks, but they suffer from high computational costs due to the quadratic complexity of self-attention layers. Block-sparse attention has been proposed to alleviate this cost. However, despite their theoretical benefits, existing implementations of block-sparse attention remain inefficient and have yet to fully realize the potential of these sparse models. In this paper, we propose Ziggs, a reuse-aware block scheduling and execution framework for efficient block-sparse attention. Ziggs clusters sparse attention blocks to expose reuse opportunities, and formulates intra-group block scheduling as a Markov Decision Process (MDP) to select execution order under limited shared memory. By combining an offline learned pattern-to-order table with a lightweight heuristic fallback for unseen block-group patterns, Ziggs reduces redundant global memory accesses while preserving GPU parallelism. The resulting schedules are encoded into lightweight metadata that guides optimized SDDMM and SpMM kernels, enabling inter-block reuse and reducing memory loads. Experiments on A100 and H200 GPUs show that Ziggs achieves up to 2.05 $\times$  speedup and reduces global memory traffic by up to 1.44 $\times$  compared to prior sparse implementations, while delivering 1.61 $\times$ –3.15 $\times$  end-to-end speedup over PyTorch dense baselines.

## Keywords

Attention mechanism, sparse matrix, block sparse, GPU

## ACM Reference Format:

Lihan Hu, Xiaoyang Lu, Xian-He Sun, and Peng Jiang. 2026. Improving Data Reuse across Blocks for Efficient Block-sparse Transformers on GPUs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26)*, October 19–22, 2026, Chicago, IL, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3838684.3846855>



This work is licensed under a Creative Commons Attribution 4.0 International License. PACT '26, Chicago, IL, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2915-7/2026/10

<https://doi.org/10.1145/3838684.3846855>

## 1 Introduction

Transformer-based architectures have become a cornerstone of modern deep learning, achieving remarkable success in natural language processing [16] and computer vision [21]. At the core of this success is the self-attention mechanism [50], which enables Transformers to capture long-range dependencies and dynamically focus on the most relevant tokens within an input sequence. However, self-attention layers incur quadratic complexity with respect to sequence length, resulting in significant computational and energy challenges [32, 33]. The attention scores are computed by multiplying the query  $Q$  and key  $K^T$  matrices, followed by a softmax normalization. The attention output  $O$  is then obtained by multiplying the normalized attention matrix  $P$  with the value  $V$  matrix. Recent systems studies [1, 26, 42] further show that Transformer inference often presents greater engineering challenges than training, particularly in achieving ultra-high throughput and efficient GPU utilization under tight memory and latency constraints.

To address these challenges, a variety of optimization techniques have been explored, including quantization [6], kernel-based approximations [8], and structured sparsity [3, 25]. Among these, block-sparse attention [5, 27, 30, 45, 46, 52] has emerged as one of the most promising approaches. The key idea is to partition an attention score matrix into sub-blocks, computing only selected blocks while treating the rest as zeros. Recent widely used studies, such as Routing Transformer [46], Reformer [27], and Pixelated Butterfly [5], leverage different block-sparse attention patterns to avoid unnecessary computation on less informative connections, reducing computational overhead without sacrificing accuracy.

Despite these advancements, state-of-the-art implementations [5, 27, 46] of block-sparse attention still deliver limited performance gains. The main reason is that they treat each block-wise multiplication as an independent GEMM operation, thereby missing opportunities for data reuse across blocks. As illustrated in Figure 1, identical  $Q$  or  $K^T$  blocks are repeatedly loaded from global memory to shared memory during attention score computation. For instance, the computation of block  $S(2, 0)$  and block  $S(2, 2)$  both require  $Q_2$  blocks, while block  $S(0, 2)$  and block  $S(2, 2)$  demand the same  $K_2^T$  blocks. Also, when multiplying  $P$  with  $V$  for attention output  $O$ , multiple blocks of  $P$  may request the same rows of  $V$ . Although substantial data reuse exists across different block computations, prior implementations overlook these opportunities and treat block operations in isolation. As a result, this reuse is not

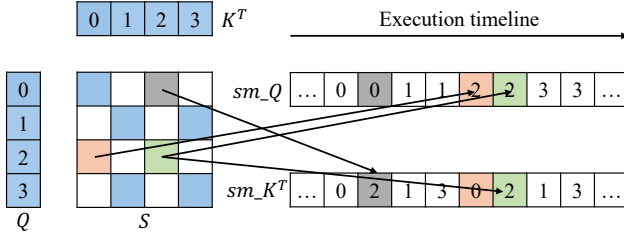


**Figure 1: Evolution of data loaded in shared memory over execution time.  $Q$  and  $K^T$  are redundantly loaded from global memory to shared memory. Left: block-sparse attention. Right: data stored in shared memory over execution time.**

effectively exploited, leading to unnecessary global memory traffic and underutilization of both computing units and on-chip memory. These inefficiencies motivate the need for inter-block data reuse across attention computations.

However, effectively exploiting inter-block reuse is non-trivial. First, finding an optimal execution order is itself a challenging scheduling problem. Data reuse depends on whether required data remains resident across consecutive computations, making reuse highly dependent on execution order. Second, inter-block reuse patterns in block-sparse Transformers are irregular and sparsity-pattern-dependent, making it difficult to design general scheduling strategies. Third, maximizing reuse does not always align with GPU-friendly parallel execution: larger block groups expose more reuse but reduce parallelism, whereas smaller groups make the opposite trade-off. Finally, even when reuse opportunities are identified, translating them into practical GPU speedups remains challenging due to limited shared memory capacity and the need for lightweight runtime control to avoid excessive divergence and overhead.

Based on these challenges, we propose Ziggs for efficient block-sparse Transformers. Ziggs adopts a two-stage design consisting of (1) reuse-aware block scheduling and (2) scheduling-guided GPU kernel execution. In the first stage, Ziggs groups sparse attention blocks to expose potential reuse opportunities and formulates block ordering within each group as a Markov Decision Process (MDP), allowing Ziggs to select execution orders that account for immediate reuse and future cache evolution. The learned scheduling results are stored as a pattern-to-order table for common sparse patterns, while unseen patterns are handled by a reuse-aware heuristic. In the second stage, Ziggs translates the scheduling decisions into compact metadata that guides optimized SDDMM and SpMM kernels. This metadata allows kernels to decide whether operands should be loaded from global memory or reused from shared memory, and enables local accumulation of SpMM partial results before final writeback. Together, reuse-aware scheduling and metadata-guided kernels reduce redundant global memory accesses and translate inter-block reuse into practical GPU speedups.

To summarize, this paper makes the following contributions:

- We identify the performance issues of previous block-sparse attention implementations and introduce an offline Q-learning-based block scheduling strategy that builds a pattern-to-order table for inter-block data reuse.

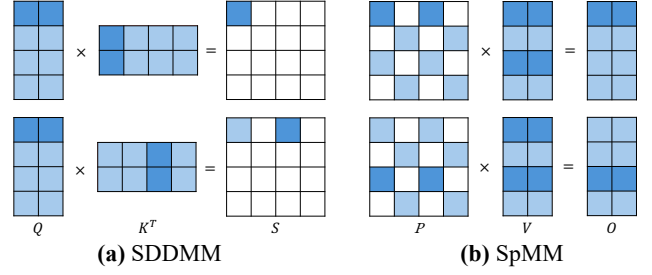


**Figure 2: Sparse attention computation.**

- We develop optimized GPU kernels that exploit the preprocessed data and achieve inter-block data reuse through GPU shared memory for both CUDA Cores and Tensor Cores.
- Our kernels achieve 1.18–2.05 $\times$  speedups and reduce global memory traffic by 1.18–1.44 $\times$  over state-of-the-art sparse implementations, while delivering 1.61–3.15 $\times$  end-to-end speedup over the PyTorch dense baseline.

## 2 Background

This section gives background on block-sparse attention and its previous implementation methods.

### 2.1 Sparse Attention

Self-attention mechanism [50] enables each token to attend to all others by computing attention scores ( $S$ ) through the dot product of the query ( $Q$ ) and key ( $K^T$ ) matrices. These scores are normalized with softmax to form  $P$ , which weights the value ( $V$ ) matrix to produce the output ( $O$ ). While this fully connected structure allows Transformers to capture rich contextual information, it incurs quadratic computational and memory complexity with respect to the input sequence length. Research has shown that many token interactions are redundant and that the attention pattern is inherently sparse [7]. Motivated by this observation, a variety of sparse attention mechanisms have been proposed [9, 10, 31, 47, 53, 54].

Sparsification in attention can be broadly categorized as static or dynamic. Static sparsification [3, 5, 10, 52] employs fixed, data-independent sparsity patterns, which lack adaptability and may degrade accuracy under aggressive pruning. Dynamic sparsification [23, 27, 46], in contrast, determines sparsity at runtime based on input data, typically achieving a better efficiency–accuracy trade-off. Our inter-block data reuse method is orthogonal to sparsity pattern generation and supports memory-efficient execution of both static and dynamic sparse attention.

For sparse attention, the computation can be decomposed into two sparse matrix operations. First, SDDMM (Sampled Dense Matrix Multiplication), shown in Figure 2(a), computes  $QK^T$  only at positions specified by the sparsity pattern of the attention score matrix  $S$ . In block-sparse attention, the sparsity pattern is defined at the block level, allowing entire blocks of  $S$  to be skipped. Second, SpMM (Sparse Matrix-Matrix Multiplication), shown in Figure 2(b), multiplies the sparse attention matrix  $P$  with the dense value matrix  $V$  to generate the output.

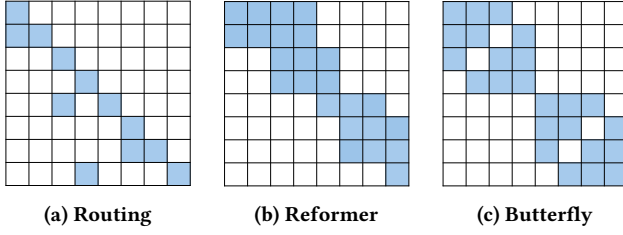


**Figure 3: Three representative block-sparse attention patterns. Each colored block consists of multiple elements in the attention matrix. Previous works typically assign each GPU thread block to process an individual block.**

## 2.2 Block-sparse Attention Mechanisms

Recent advances in efficient Transformers have introduced variants that exploit block-sparse attention patterns to reduce computational complexity while maintaining accuracy, making them well-suited for efficient execution on modern hardware accelerators. Figure 3 illustrates three representative block-sparse attention patterns, where nonzero entries appear only within specific blocks, enabling efficient computation while preserving relevant dependencies. Specifically, Routing Transformer [46] uses content-based routing to dynamically select a subset of keys for each query, generating sparse attention scores clustered by token similarity. Reformer [27] employs locality-sensitive hashing to group queries and keys into buckets, resulting in block-sparse attention where each query attends only to keys within the same hash bucket. Pixelated Butterfly [5] organizes tokens in a hierarchical butterfly structure, producing sparse attention patterns characterized by diagonal blocks corresponding to multi-scale token interactions.

## 2.3 Existing Implementations of Block-sparse Attention

Existing implementations of block-sparse attention adopt different execution strategies on GPUs. Some libraries [14, 19, 28, 39, 45, 48] adopt the Block Compressed Sparse Row (BCSR) format, which facilitates efficient parallelization of dense matrix blocks. However, this approach often suffers from load imbalance, since different rows often contain varying numbers of blocks.

Reformer [27] and Routing Transformer [46] instead rely on PyTorch-based implementations using BCOO format. In these implementations, the corresponding  $Q$ ,  $K^T$ , and  $V$  blocks are gathered and reshaped into dense block tensors and processed block by block. Although this approach is simple, it incurs many kernel launches and intermediate-memory operations, which limit practical performance. Pixelated Butterfly [5] adopts a Triton-based implementation that builds lookup tables from the sparse layout and iterates over blocks accordingly. Although this improves flexibility and programmability, block computations are still executed largely independently, limiting opportunities for data reuse.

Despite these differences, existing implementations share a common limitation: they typically process each block independently and do not explicitly exploit data reuse across blocks. In practice, many blocks share rows or columns, and reusing the corresponding  $Q$ ,  $K^T$ , or  $V$  data through shared memory could significantly

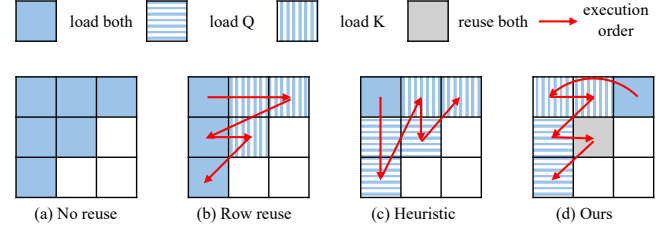


**Figure 4: Illustration of inter-block data reuse effectiveness under different block execution orders.**

reduce redundant memory accesses. In addition, block-wise execution often introduces extra intermediate storage and frequent kernel launches, further limiting effective throughput. According to our profiling results (shown in Figure 9), PyTorch implementation achieves only 5%-15% of the peak throughput on an NVIDIA A100 GPU.

## 3 Improving Inter-block Data Reuse through Block Scheduling

To achieve inter-block reuse, Ziggs introduces an offline Q-learning based block scheduling strategy that builds a pattern-to-order table for reordering sparse attention blocks before kernel launch.

### 3.1 Block Scheduling as a Markov Decision Process

To expose inter-block reuse opportunities, we first group attention blocks using a hierarchical clustering method inspired by prior work [24], based on similarity over their row and column indices. We use this clustering only for initial block grouping, while Ziggs further introduces cache-aware scheduling and reuse-aware kernel execution. Row overlap captures opportunities for reusing  $Q$  and output accumulation, while column overlap captures opportunities for reusing  $K^T$  and  $V$ . The group size is constrained to balance reuse, parallelism, and shared memory usage.

After attention blocks are clustered into block groups, we want to maximize data reuse among the blocks within each block group. This depends critically on the execution order, because different schedules lead to different shared memory states and therefore expose different opportunities for reusing  $Q$  and  $K^T/V$ . Since shared memory can hold only a small number of active blocks at a time, reuse is possible only when subsequent computations access the same row or column data before the resident blocks are evicted. Therefore, even for the same block group, different execution orders can result in substantially different memory access overhead.

Figure 4 shows that different execution orders can lead to different memory access costs for the same block group. Without inter-block reuse, each block independently loads both its  $Q$  and  $K^T/V$  blocks, resulting in 12 block data loads, as shown in Figure 4(a). A simple row-wise order reuses  $Q$  across blocks in the same row and reduces the number of memory loads to 9, but it misses column-wise reuse opportunities, as shown in Figure 4(b).

Figure 4(c) shows a frequency-based heuristic schedule under a shared memory budget that can hold at most three  $Q$  or  $K^T/V$  blocks. This heuristic first pins a persistent  $Q$  block from the most

frequent row ( $Q_0$  in this example) to enable row reuse, and then orders the remaining blocks by prioritizing column locality so that consecutive blocks can reuse the same  $K^T/V$  block whenever possible. With this pinned-row and column-oriented ordering, the number of loads is reduced to 7. The shared memory contents evolve as  $Q_0K_0 \rightarrow Q_0Q_1K_0 \rightarrow Q_0Q_2K_0 \rightarrow Q_0Q_2K_1 \rightarrow Q_0Q_1K_1 \rightarrow Q_0Q_1K_2$ . In contrast, our method shown in Figure 4(d) uses the same shared memory capacity but manages the contents as an LRU-like cache. This allows the scheduler to preserve data useful for future blocks rather than only following a fixed pinned-row and column-local order, further reducing the number of loads to 6. The corresponding shared memory contents evolve as  $Q_0K_2 \rightarrow Q_0K_0K_2 \rightarrow Q_0K_0K_1 \rightarrow Q_1K_0K_1 \rightarrow Q_1K_0K_2 \rightarrow Q_1Q_2K_0$ .

This indicates that simply pinning a frequently accessed block or following local row/column locality is not always sufficient, since the benefit of an execution order depends on how memory accesses evolve across the entire block group. Scheduling is therefore an ordering problem whose objective is to minimize the total memory access cost. To make effective decisions, the scheduler needs to track both the blocks that have already been executed and the data currently residing in shared memory.

Given the resulting block groups, we formulate block scheduling within each block group as a Markov Decision Process (MDP). Specifically, a state  $s_t$  is defined as

$$s_t = (\mathcal{V}_t, \mathcal{M}_t),$$

where  $\mathcal{V}_t$  denotes the set of visited blocks and  $\mathcal{M}_t$  denotes the shared memory cache state after step  $t - 1$ . Unlike a static reuse annotation,  $\mathcal{M}_t$  explicitly records which  $Q$  and  $K^T/V$  blocks are currently resident in shared memory. We model  $\mathcal{M}_t$  as an LRU-like cache: when a block is executed, its corresponding  $Q$  and  $K^T/V$  blocks are inserted into or refreshed in the cache; if the cache capacity is exceeded, the least recently used entry is evicted.

An action  $a_t \in \mathcal{A}_t$  corresponds to selecting the next block from the candidate set in  $\mathcal{G}$ . The state transition is deterministic: executing  $a_t$  updates the state to  $s_{t+1}$  by adding  $a_t$  to the visited set  $\mathcal{V}_t$  and updating  $\mathcal{M}_t$  according to the newly loaded and reused blocks in shared memory. Since each reused operand avoids one global-memory block load in our block-level model, we define the reward as the number of avoided operand loads:

$$\mathcal{R}(s_t, a_t) = \begin{cases} 2, & \text{if both } Q_t \text{ and } K_t^T/V_t \text{ are reused,} \\ 1, & \text{if either } Q_t \text{ or } K_t^T/V_t \text{ is reused,} \\ 0, & \text{if both } Q_t \text{ and } K_t^T/V_t \text{ must be loaded,} \\ -1000, & \text{if } a_t \in \mathcal{V}_t. \end{cases}$$

Under this formulation, the scheduler can determine whether  $Q$  can be reused by checking whether the row block required by  $a_t$  is present in  $\mathcal{M}_t$ . Similarly,  $K^T/V$  can be reused if the column block required by  $a_t$  is present in the current cache state. Otherwise, the missing data must be loaded from global memory and inserted into  $\mathcal{M}_t$  according to the cache update rule. To prevent a block in  $\mathcal{V}_t$  from being selected again, we assign a large negative reward ( $-1000$ ) to such actions. The scheduling objective is therefore

$$\arg \max_{\pi} \sum_t \mathcal{R}(s_t, a_t).$$

We use this MDP formulation to construct a reuse-aware scheduler to select efficient execution order for block groups. Specifically, we apply tabular Q-learning [2] offline on representative block-sparse patterns collected from commonly used sparse attention mechanisms [3, 5, 12, 27, 46, 52]. During offline training, Q-learning explores different execution orders under the visited block set  $\mathcal{V}_t$  and the shared memory cache state  $\mathcal{M}_t$ , and updates the state-action values according to the reuse reward. The learned Q-table captures how scheduling decisions affect both immediate operand reuse and future shared memory states.

After offline training, we use the learned policy to generate execution orders for frequently occurring block-group patterns and store the results in a pattern-to-order table. At runtime, Ziggs canonicalizes the sparse pattern of each block group and queries this table. If a matching pattern is found, Ziggs directly uses the stored execution order, avoiding online Q-learning during inference.

For block-group patterns not covered by the table, Ziggs invokes the frequency-based heuristic illustrated in Figure 4 as a lightweight fallback. The heuristic prioritizes the most frequent row to capture  $Q$  reuse and then orders the remaining blocks to improve column locality for  $K^T/V$  reuse. Once an unseen pattern is scheduled, Ziggs inserts the generated order into a runtime scheduling cache, so repeated occurrences of the same pattern can reuse the cached order directly. The runtime table contains about 50K canonical block-group patterns and occupies 20 MB. It covers over 90% of groups for static patterns and 50–70% for dynamic patterns, while the remaining groups use the heuristic fallback. The temporary Q-table is constructed separately for each pattern, and only the best learned order is retained after offline training.

### 3.2 Preprocessing Overhead

Although Ziggs introduces a preprocessing stage before kernel execution, its runtime overhead remains manageable because the block-scheduling optimization is largely removed from the inference critical path. Since the Q-learning procedure is performed offline, runtime scheduling for common block-group patterns only requires canonicalizing each sparse pattern and querying the pattern-to-order table. These operations are lightweight compared with the subsequent attention computation, especially because the table stores only compact block execution orders rather than dense data structures.

The remaining runtime cost comes from block-group patterns not found in the pattern-to-order table. For these unseen patterns, Ziggs applies the reuse-aware heuristic only within each clustered block group, rather than over the entire sparse attention matrix. Since clustering has already grouped blocks with related row and column indices, the heuristic efficiently orders blocks based on row frequency and column locality to capture dominant reuse opportunities. This design avoids iterative learning or global optimization during inference while preserving most inter-block reuse opportunities. Once an unseen pattern is scheduled, Ziggs stores the generated order in a runtime scheduling cache, so repeated patterns can directly reuse the cached order. The reported preprocessing time includes block grouping, canonicalization, table lookup, and heuristic fallback when needed, and is measured after the sparse attention pattern has been generated.

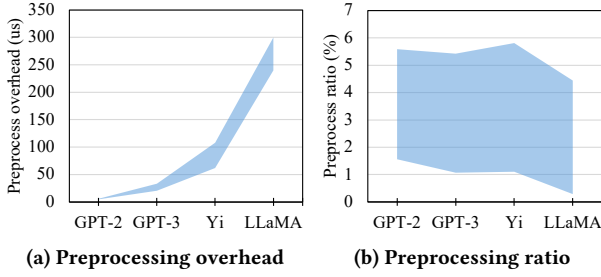


Figure 5: Preprocessing overhead and ratio for a single layer across different models and sparse patterns on an A100 GPU.

As shown in Figure 5, preprocessing accounts for less than 6% of the overall runtime in most cases. Moreover, as sequence length and model size increase, attention computation grows faster than the preprocessing cost, making the relative overhead even smaller. This trend shows that Ziggs’ scheduling pipeline scales well for long-sequence Transformer workloads, where reducing redundant global memory traffic dominates the small preprocessing cost. This also indicates that the scheduling cost is effectively amortized as the attention workload grows.

## 4 Implementation on GPU Kernels

Section 3 transforms block-sparse attention into reuse-aware block groups with explicit metadata. Based on this representation, we next describe how the resulting scheduling and metadata are mapped to efficient GPU kernels, enabling lightweight runtime control and coordinated shared memory management.

### 4.1 Reuse-aware Execution with $reuse\_flg$

To enable efficient GPU execution, Ziggs encodes the scheduling results into a compact metadata array, denoted as  $reuse\_flg$ . The blocks in each group are processed according to the execution order obtained from the pattern-to-order table or generated by the runtime heuristic. For each scheduled block, Ziggs assigns a  $reuse\_flg$  based on the shared memory cache state induced by this order. Specifically, for a block  $(r, c)$ , reuse of  $Q$  is enabled if the required  $Q_r$  block is still resident in the LRU-like cache after previous block computations. Similarly, reuse of  $K^T/V$  is enabled if the required  $K_c^T/V_c$  block is resident in the cache. If either operand is not resident, it is loaded from global memory and the cache state is updated accordingly.

Based on the cache state before each block is executed, Ziggs assigns one of four flags:  $reuse\_flg = 0$  for  $Q$  reuse only,  $reuse\_flg = 1$  for reuse of both  $Q$  and  $K^T/V$ ,  $reuse\_flg = 2$  for  $K^T/V$  reuse only, and  $reuse\_flg = 3$  for no reuse. During kernel execution,  $reuse\_flg$  reduces reuse decisions to simple integer checks. Each thread block either reuses the required operands from shared memory or loads the missing operands from global memory.

Ziggs also records  $row\_reuse$  to better support SpMM. After the execution order is determined, Ziggs selects the most frequent rows in each block group as  $row\_reuse$ . This metadata is independent of LRU-based operand reuse and identifies output rows whose partial sums are accumulated locally before the final writeback in SpMM.

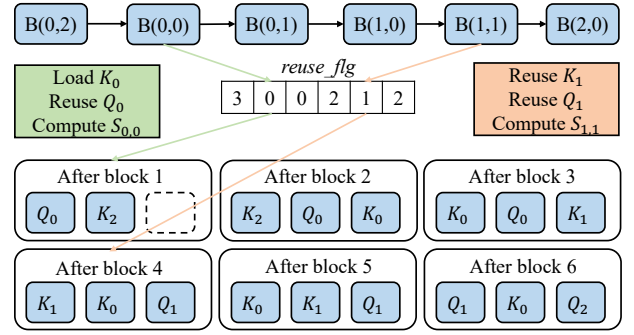


Figure 6: Running example of reuse-aware SDDMM execution. Ziggs uses the scheduled order and  $reuse\_flg$  to guide  $Q/K^T$  loading, reuse, and LRU-like cache updates.

### 4.2 Block-sparse SDDMM

Figure 6 illustrates how  $reuse\_flg$  guides the block-sparse SDDMM kernel using a running example. Each thread block processes one scheduled block group and computes one dense output block  $S_{r,c} = Q_r K_c^T$  at a time. Before each block multiplication, the kernel checks the corresponding  $reuse\_flg$  value to decide whether the required  $Q_r$  and  $K_c^T$  operands should be fetched from the shared memory cache or loaded from global memory.

The green box in Figure 6 shows a case where  $reuse\_flg = 0$ , corresponding to  $Q$  reuse only. For block  $B(0,0)$ , the kernel reuses  $Q_0$  from the shared memory cache and loads the required column operand  $K_0^T$  from global memory. It then computes the dense block multiplication  $S_{0,0} = Q_0 K_0^T$  and refreshes the LRU-like cache state according to this access. In this way, the kernel avoids recomputing reuse conditions at runtime and only performs the data movement already encoded by the scheduler. The orange box shows a full-reuse case where  $reuse\_flg = 1$ . For block  $B(1,1)$ , the kernel directly obtains both  $Q_1$  and  $K_1^T$  from the shared memory cache and computes  $S_{1,1} = Q_1 K_1^T$  without issuing additional global loads for either operand. After each block is processed, reused operands are refreshed, newly loaded operands are inserted, and the least recently used operand is evicted when the cache is full. This cache evolution allows later blocks to reuse operands that remain useful for subsequent computations.

Figure 7 further shows how this execution model is applied to representative block-sparse attention patterns. After clustering, nonzero blocks with reuse potential are assigned to the same block group, and each colored group is mapped to one GPU thread block. The dashed arrows indicate the execution order produced by Ziggs within each block group, showing how the thread block traverses its assigned sparse blocks to exploit local shared memory reuse.

After the required operands are available in shared memory, threads collaboratively compute the dense block multiplication. In the CUDA-Core implementation, warps load tiles of  $Q_r$  and  $K_c^T$  from shared memory, accumulate partial results in registers, and write the resulting  $S_{r,c}$  block back to global memory. To further improve performance, we adopt double buffering to overlap memory movement with computation. While the current tile is being processed, data for the next tile can be prefetched into shared

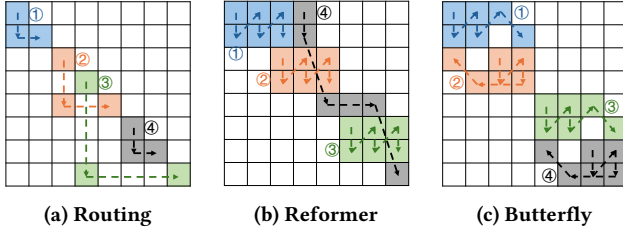


**Figure 7: Block-group execution for representative block-sparse attention patterns. Blocks with the same color are mapped to the same GPU thread block, and numbered arrows show the scheduled order within each group.**

memory using vectorized asynchronous copy instructions such as LDGSTS.128, improving global-to-shared memory efficiency. For the Tensor-Core implementation, the same reuse-aware schedule is preserved, while dense block multiplication is mapped to warp-level `mma.sync.aligned.m16n8k8` instructions. Thus, both implementations share the same cache-aware execution path but use different compute primitives to match the target GPU execution units.

### 4.3 Block-sparse SpMM

Algorithm 1 presents the pseudocode of our block-sparse SpMM kernel. Compared with SDDMM, SpMM has two distinct characteristics. First, the sparse attention block  $P_{r,c}$  is consumed only once for the current output update and therefore does not provide reuse opportunity. Second, multiple blocks with the same row index contribute partial sums to the same output row  $O_r$ . If each block result is immediately written back to global memory, these partial sums repeatedly update the same output row, causing frequent global memory writes and atomic operations.

In Ziggs, the `reuse_flg` metadata is used to control the reuse of  $V_c$ . Since the column index  $c$  corresponds to both  $K_c^T$  in SDDMM and  $V_c$  in SpMM, the same schedule-derived reuse information can be reused in both kernels. In particular, `reuse_flg = 1` and `reuse_flg = 2` indicate that the column operand is resident in the shared memory cache, so the SpMM kernel obtains  $V_c$  through `GetSmem`. Otherwise, the required  $V_c$  block is loaded from global memory through `LoadSmem`. After each access, the value cache is refreshed to reflect the current use, so that later blocks in the execution order can reuse recently accessed  $V$  blocks.

Unlike SDDMM, SpMM does not need to cache  $P$  across block computations. Each  $P_{r,c}$  block is loaded for the current multiplication and then discarded. This allows the kernel to dedicate more on-chip storage to the value cache and to local output accumulation. To reduce repeated output updates, Ziggs uses the auxiliary `row_reuse` metadata. After the execution order is determined, Ziggs selects the most frequently occurring rows within each block group as `row_reuse`. During execution, the kernel checks whether the current row  $r$  belongs to `row_reuse`. If so, the partial result  $P_{r,c}V_c$  is accumulated into the local buffer `inter_res` instead of being immediately written back to global memory. If the row is not in `row_reuse`, the result is directly committed using `atomicAdd`.

#### Algorithm 1: Pseudocode of block-sparse SpMM

**Input:**  $P, V, row\_idx, col\_idx, reuse\_flg, row\_reuse$   
**Output:**  $O$

```

1 _shared_  $p\_sm, v\_sm$  // Shared memory
2  $inter\_res[] = 0$  // Local accumulation buffer
3 for  $bid$  in  $range(n\_blocks)$ 
4    $r = row\_idx[bid], c = col\_idx[bid]$ 
5    $reuse\_id = row\_reuse.indexing(r)$ 
6    $p\_sm = LoadSmem(P, bid)$ 
7   if  $reuse\_flg[bid] == 1$  or  $reuse\_flg[bid] == 2$ 
8      $v\_sm = GetSmem(smem\_cache, V, c)$ 
9   else
10     $v\_sm = LoadSmem(smem\_cache, V, c)$ 
11     $Refresh(smem\_cache, v\_sm)$ 
12     $res = p\_sm * v\_sm$ 
13    if  $reuse\_id \neq -1$ 
14       $inter\_res[reuse\_id] += res$ 
15    else
16       $atomicAdd(O[r], res)$ 
17  $atomicAdd(O[row\_reuse], inter\_res)$ 

```

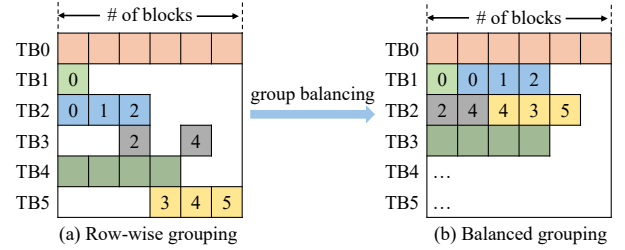


**Figure 8: Comparison between row-wise grouping and our balanced grouping strategy. The numbers in each block indicate the corresponding column blocks. Each color represents blocks sharing the same row.**

After all blocks in the group have been processed, the accumulated partial sums in `inter_res` are written back to the corresponding output rows in a single step. This design reduces redundant memory loads through  $V$  reuse and separately reduces irregular output updates through row-aware local accumulation. As a result, the SpMM kernel preserves the same cache-aware execution order as SDDMM while addressing the additional write-side bottleneck introduced by sparse output accumulation.

### 4.4 Kernel Fusion

For long-sequence inputs, materializing the attention matrix incurs substantial memory overhead. To avoid this cost, attention computation is often implemented using fused kernels that combine SDDMM, softmax, and SpMM, eliminating the need to materialize intermediate attention scores [36]. Following the design of FlashAttention-2 [11], recent block-sparse fused attention implementations, such as Block-Sparse Attention [20] and FlexAttention [13], adopt a row-wise execution strategy in which blocks sharing the same row indices are assigned to the same thread block,

**Table 1: Parameters of attention models with widely adopted Transformer architectures in our experiments.**

| Model Name       | $seq\_len$ | $n_{layers}$ | $d_{model}$ | $n_{heads}$ |
|------------------|------------|--------------|-------------|-------------|
| GPT-2 Large [43] | 1024       | 36           | 1280        | 20          |
| GPT-3 6.7B [4]   | 2048       | 32           | 4096        | 32          |
| Yi 6B [51]       | 4096       | 32           | 4096        | 32          |
| LLaMA 3 70B [17] | 8192       | 80           | 8192        | 64          |

allowing intermediate results to be retained locally throughout the fused computation.

However, this row-wise execution can lead to severe workload imbalance under skewed sparse patterns. As illustrated in Figure 8(a), different row groups may contain highly varying numbers of blocks, causing some thread blocks to process many blocks while others remain underutilized. Moreover, this imbalance cannot be addressed by arbitrarily repartitioning large row groups, because in fused kernels all blocks belonging to the same row group must be processed by the same thread block. These blocks share the same softmax normalization and output accumulation for that row, and splitting them across multiple thread blocks would require additional synchronization and intermediate writes to global memory, thereby breaking the intended fused execution flow.

To address this issue, our reuse-aware framework operates as a row-aware scheduling procedure under the fusion constraint, keeping each row group intact while merging two small row groups into the same block group whenever possible, as shown in Figure 8(b). Each block group is then assigned to a single thread block. During execution, the thread block maintains in shared memory the  $Q$  blocks corresponding to the two rows, together with the softmax normalization statistics, thereby preserving the correctness of fused execution since each row group still performs its own softmax normalization and output accumulation. At the same time, placing two row groups in the same block group also creates additional inter-block reuse opportunities, allowing the loaded  $K^T/V$  data to be reused across row groups. As a result, this design simultaneously ensures correct fused execution, better load balance, and reduced global memory accesses.

## 5 Experiment

### 5.1 Experimental Setup

**Dataset:** We evaluate Ziggs on three widely used standard benchmarks for Transformer models with long-context dependencies: WikiText-103 [35], EnWiki8 [34], and PG-19 [44].

**Models and Configurations:** Our experiments are conducted on six representative block-sparse attention patterns: Reformer [27], Butterfly [5], and Routing [46], which are introduced earlier, as well as Longformer [3], BigBird [52], and LongNet [12] to demonstrate the generality of our method. To demonstrate the scalability of Ziggs, we evaluate it using model configurations consistent with widely adopted Transformer architectures (Table 1). Kernel performance is profiled with Nsight Compute and averaged over 50 runs. We also validate Ziggs against the original block-sparse computation and observe numerical equivalence within a tolerance of  $10^{-3}$ .

**Baselines:** We compare Ziggs with the following dense and sparse implementations on GPUs:

- PyTorch [41] Dense/Sparse (PyDen/PySp): dense attention uses `torch.matmul` for end-to-end comparison, while sparse attention uses gathered block-wise einsum.
- cuSPARSE (cuSp) [39]: a highly optimized library released by NVIDIA for sparse operations, including SDDMM and SpMM.
- xFormers [28]: a Triton-based library for block-sparse attention.
- RoDe [40]: a row-wise decomposition strategy that restructures sparse matrix multiplications.
- FlashSparse [48]: an optimized sparse operations framework for SpMM and SDDMM on Tensor Cores.
- Block-Sparse Attention (BSAttn) [20]: a block-sparse implementation of FlashAttention-2 [11] released by MIT Han Lab.
- FlexAttention (FlexAttn) [13]: a compiler-driven fused block-sparse implementation integrated into PyTorch by Meta.
- Triton-based implementation: a fused block-sparse attention kernel implemented in Triton [49].

For each comparison, all applicable methods are evaluated under the same sparsity patterns and hardware settings. Figures 9–11 include only baselines compatible with the corresponding CUDA Core or Tensor Core execution paths.

**Platform and Settings:** All experiments are conducted on two GPU servers in the UIUC NCSA Delta cluster. One server is equipped with two AMD EPYC 7763 CPUs @ 2.45 GHz, 2 TB RAM, and four NVIDIA A100 GPUs. The other server is equipped with two Intel Xeon Platinum 8558 CPUs @ 2.1 GHz, 2 TB RAM, and eight NVIDIA H200 GPUs. The operating systems are both Red Hat Enterprise Linux 8.8. Our CUDA code is compiled with the CUDA 12.8 toolkit and NVCC V12.8.93.

### 5.2 Performance Comparison

**5.2.1 Throughput.** We evaluate the throughput of Ziggs against several baselines using both CUDA-core and Tensor-core implementations. As shown in Figures 9 and 10, Ziggs consistently achieves the highest throughput, with the CUDA-core implementation evaluated on A100 and the Tensor-core implementation evaluated on H200. Compared with xFormers, our method achieves  $1.18\times$ – $2.05\times$  higher throughput. It delivers  $1.45\times$ – $3.58\times$  speedup over RoDe,  $1.34\times$ – $2.16\times$  over FlashSparse,  $3.03\times$ – $14.41\times$  over cuSPARSE, and up to  $6.33\times$  over the PyTorch sparse baseline.

These performance gains stem from explicitly exploiting inter-block data reuse, which enables  $Q$ ,  $K^T$ , and  $V$  blocks to be reused across both SDDMM and SpMM phases. This significantly reduces redundant global memory accesses and lowers overall memory traffic, the dominant bottleneck in sparse attention.

In contrast, xFormers achieves strong performance on both CUDA and Tensor Cores but does not exploit inter-block reuse, leading to redundant memory loads and limited GPU utilization due to independent block execution. RoDe and cuSPARSE are primarily optimized for highly sparse regimes and CSR-based computation, and their efficiency degrades when sparsity is moderate or block density increases. FlashSparse effectively leverages Tensor Cores for block-sparse computation, but similarly lacks inter-block reuse and thus cannot eliminate redundant memory traffic.

**Table 2: Global memory traffic comparison with xFormers using CUDA Cores on an A100 GPU.**

| Models           | SDDMM  |          | SpMM   |          | Reduction (×) |      |
|------------------|--------|----------|--------|----------|---------------|------|
|                  | Ziggs  | xFormers | Ziggs  | xFormers | SDDMM         | SpMM |
| <b>Routing</b>   |        |          |        |          |               |      |
| GPT-2 Large      | 14.6M  | 17.3M    | 11.2M  | 13.3M    | 1.18          | 1.19 |
| GPT-3 6.7B       | 165.5M | 202.2M   | 168.1M | 201.7M   | 1.22          | 1.2  |
| Yi 6B            | 531.7M | 717.3M   | 522.1M | 710.7M   | 1.35          | 1.36 |
| LLaMA 3 70B      | 3.82G  | 5.17G    | 3.97G  | 5.11G    | 1.35          | 1.29 |
| <b>Reformer</b>  |        |          |        |          |               |      |
| GPT-2 Large      | 15.8M  | 20.1M    | 11.7M  | 14.5M    | 1.27          | 1.24 |
| GPT-3 6.7B       | 215.6M | 279.9M   | 191.5M | 254.3M   | 1.3           | 1.33 |
| Yi 6B            | 613.7M | 886.5M   | 602.2M | 868.4M   | 1.44          | 1.44 |
| LLaMA 3 70B      | 4.75G  | 6.67G    | 4.83G  | 6.71G    | 1.4           | 1.39 |
| <b>Butterfly</b> |        |          |        |          |               |      |
| GPT-2 Large      | 14.8M  | 18.7M    | 11.5M  | 13.9M    | 1.26          | 1.21 |
| GPT-3 6.7B       | 192.4M | 251.2M   | 178.7M | 232.2M   | 1.3           | 1.3  |
| Yi 6B            | 584.5M | 811.1M   | 577.3M | 802.2M   | 1.39          | 1.39 |
| LLaMA 3 70B      | 4.52G  | 6.11G    | 4.4G   | 6.06G    | 1.35          | 1.38 |

Figure 11 further demonstrates the generality of our method across diverse sparse attention patterns, including Longformer, Big-Bird, and LongNet, evaluated on A100 Tensor Cores. Despite their different sparsity structures, our method consistently outperforms all baselines. Additional experiments across a wider range of configurations show similar trends. Overall, by combining reuse-aware block scheduling with optimized GPU kernels, our method amortizes memory loads across multiple blocks, increases arithmetic intensity, and achieves best-in-class performance on modern GPUs.

**5.2.2 Global Memory Traffic.** To quantify the effectiveness of inter-block data reuse, we evaluate its impact on memory traffic. Table 2 compares our method with a representative xFormers implementation on the PG-19 dataset using CUDA Cores on an A100 GPU. We choose xFormers as a conservative baseline, as other sparse attention methods follow similar or less optimized memory access patterns and do not exploit inter-block reuse, resulting in comparable or higher memory traffic. Since our preprocessing is shared across CUDA Core and Tensor Core implementations, the observed memory traffic reduction is consistent across execution modes and GPU architectures, including A100 and H200.

As shown in Table 2, our method reduces memory traffic by 1.18–1.44× compared to xFormers. This reduction stems from eliminating redundant loads of identical  $Q$ ,  $K^T$ , and  $V$  blocks across sparse computations. By grouping blocks based on similarity, our method improves temporal locality and amortizes memory accesses across multiple block operations. In contrast, existing implementations process blocks largely in isolation, leading to repeated memory accesses even when blocks share rows or columns.

By reducing redundant memory loads, Ziggs alleviates bandwidth pressure and improves throughput across diverse configurations and hardware platforms. The reduction is especially important for larger models and longer sequences, where the number of nonzero blocks increases and repeated accesses to the same  $Q$ ,  $K^T$ , and  $V$  blocks become more frequent. Since these redundant accesses are removed before kernel execution through scheduling-guided metadata, the traffic reduction directly translates into higher effective bandwidth utilization during sparse attention computation.

**5.2.3 End-to-end Performance.** To assess the benefits of Ziggs, we measure the end-to-end performance of the aforementioned models under a batch size of 4, as shown in Figure 12. During the prefill stage, fully connected layers are evaluated in bf16, while the softmax layer uses float precision. Under this setting, Ziggs achieves a 1.61–3.15× speedup over the PyTorch dense baseline. These results show that accelerating block-sparse attention can reduce end-to-end latency in realistic inference pipelines, where attention is invoked repeatedly across many layers and heads.

The end-to-end gain is lower than the standalone kernel speedup because attention is only one component of the full inference pipeline. Other operations, including projections, normalization, softmax, layout transformations, and framework overhead, remain unchanged. Thus, even though Ziggs accelerates both SDDMM and SpMM, the overall speedup is bounded by the remaining non-optimized components. This effect is more visible on Tensor Cores: after matrix multiplications are accelerated, surrounding memory-bound operations such as softmax occupy a larger fraction of the total inference runtime.

Overall, these results complement the kernel-level throughput and memory-traffic measurements. The standalone results show that Ziggs reduces redundant global memory accesses inside SDDMM and SpMM, while the end-to-end results show that these improvements remain visible after integration into complete Transformer inference. This confirms that inter-block reuse is not only a kernel-level optimization, but also an effective system-level technique for practical block-sparse Transformer execution.

**5.2.4 Performance on Long Sequences.** Figure 13 presents the end-to-end performance of our fused kernel on an A100 GPU, normalized to the Block-Sparse Attention (BSAttn) baseline. We compare against FlexAttention and Triton-based implementations across three sparsity patterns: Butterfly, Routing, and Reformer. All experiments are conducted with a head dimension of 32, while varying the sequence length (131K–1M) and the number of attention heads.

Overall, Ziggs consistently outperforms all baselines across a wide range of sequence lengths and attention head configurations. The improvement becomes more pronounced as sequence length increases, reaching up to 1.78× speedup for the Reformer pattern. This trend highlights the effectiveness of reuse-aware scheduling in long-context attention: longer sequences create more nonzero blocks, which expose more repeated accesses to the same  $Q$ ,  $K^T$ , and  $V$  operands. By scheduling blocks with row or column locality close together, Ziggs amortizes these repeated loads and reduces memory traffic, which becomes the dominant bottleneck at long sequence lengths. In contrast, Triton-based implementations often underperform relative to BSAttn due to inefficient sparse indexing and limited pipeline optimization, which lead to irregular memory accesses and higher overall memory traffic. FlexAttention achieves moderate improvements but remains consistently below Ziggs, as it does not exploit inter-block data reuse across blocks.

We further observe that the magnitude of improvement varies across sparsity patterns. Ziggs achieves the largest gains on Reformer and Butterfly patterns, where blocks exhibit higher overlap in row or column indices, enabling effective reuse of  $Q$ ,  $K^T$ , and  $V$ . In comparison, Routing patterns provide fewer reuse opportunities due to their more irregular connectivity, leading to relatively

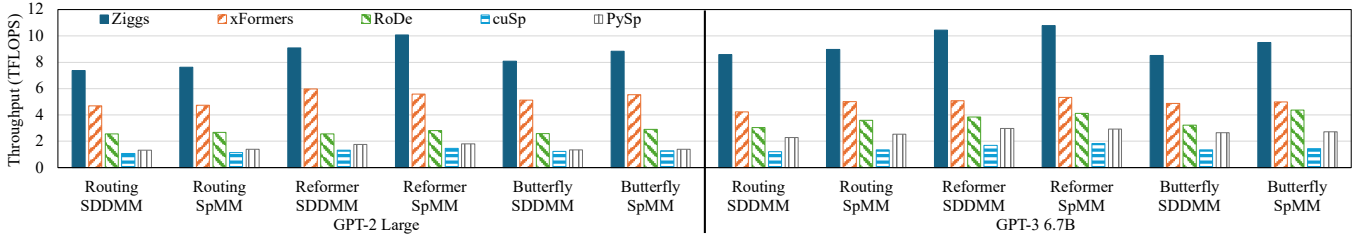


Figure 9: Throughput comparisons on EnWiki8 dataset for GPT-2 Large and GPT-3 6.7B using CUDA Cores on an A100 GPU.

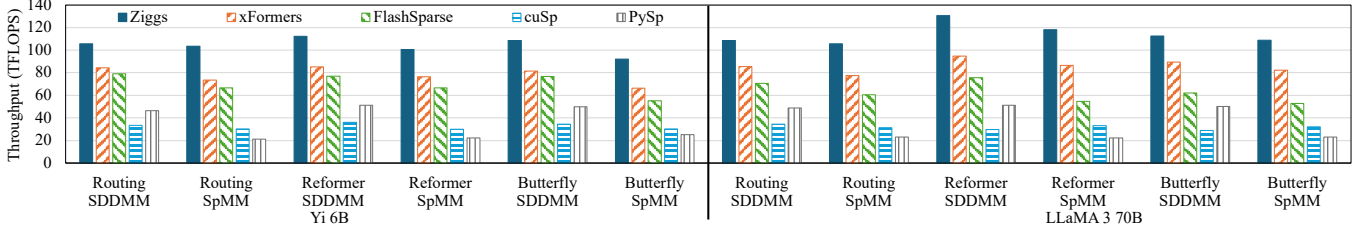


Figure 10: Throughput comparisons on WikiText-103 dataset for Yi 6B and LLaMA 3 70B using Tensor Cores on an H200 GPU.

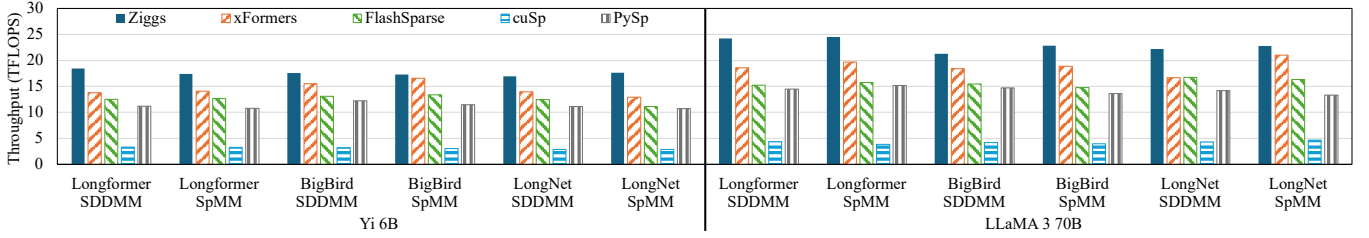


Figure 11: Throughput comparison across additional sparse attention patterns (Longformer, BigBird, and LongNet) on an A100 GPU using Tensor Cores, demonstrating the generality of our reuse-aware scheduling.

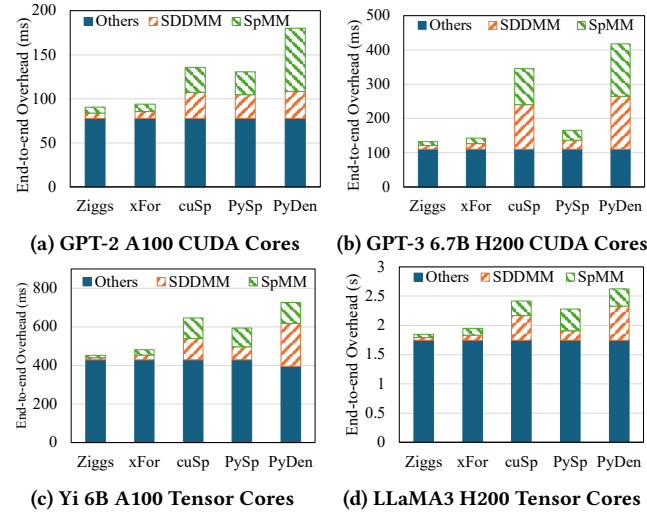


Figure 12: End-to-end performance comparisons across different models on A100 and H200 GPUs. xFor represents xFormers in the above experiments.

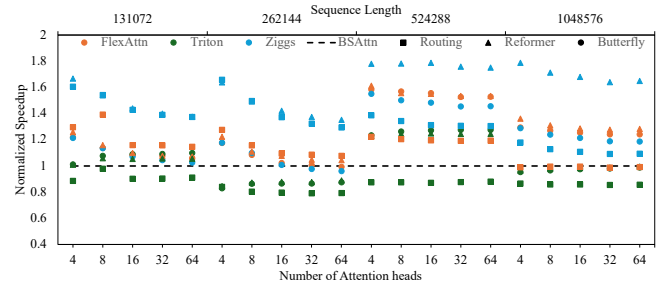


Figure 13: End-to-end performance comparison on an A100 GPU with long sequence length, normalized to the Block-Sparse Attention.

smaller speedups. Finally, we note that increasing the number of attention heads reduces the per-head workload and limits the amount of reusable data within each block group, which slightly diminishes the relative performance gains. Nevertheless, Ziggs consistently maintains superior performance across all configurations.

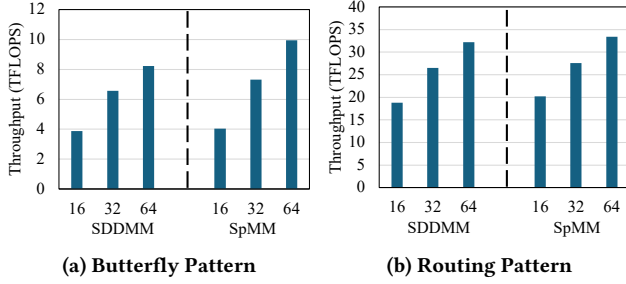


Figure 14: Effect of block size on the computational performance of SDDMM and SpMM across GPT-2 Large (Butterfly) and Yi 6B (Routing).

### 5.3 Effects of Parameters on Performance

Following prior block-sparse attention work, we evaluate block sizes of 16, 32, and 64, which provide a practical balance between sparsity granularity and model accuracy. We evaluate the impact of two key parameters in Ziggs: block size and  $n_{blocks}$ . These two parameters respectively control the computation granularity of dense block multiplication and the group size used in reuse-aware scheduling. Both directly affect the trade-off between data reuse, memory efficiency, and GPU parallelism.

**Block size.** Figure 14 shows the performance of SDDMM and SpMM under different block sizes. Increasing the block size from 16 to 32 significantly improves throughput, as larger blocks provide higher arithmetic intensity, better amortization of memory accesses, and more efficient use of GPU execution units. However, the performance gain becomes marginal when the block size is further increased to 64. Although larger blocks expose more computation per memory access, they also increase shared memory usage and register pressure, which in turn reduce SM occupancy and limit parallelism. As a result, excessively large blocks do not translate into proportional performance gains. Overall, these results suggest that a moderate block size, such as 32, provides the best balance between computational efficiency and hardware utilization.

**Effect of  $n_{blocks}$ .** Figure 15 illustrates the effect of varying  $n_{blocks}$ , which controls the maximum number of blocks in each reuse-aware group. Larger  $n_{blocks}$  exposes more inter-block reuse by allowing more blocks to be scheduled together, but it also reduces the number of independent GPU work units and may increase synchronization or write contention in SpMM. As shown in the figure, short sequences benefit less from larger groups because the limited reuse opportunity cannot compensate for the reduced parallelism. For longer sequences, performance first improves as  $n_{blocks}$  increases, but eventually degrades once the loss of parallelism and scheduling overhead outweighs the additional reuse. Therefore, Ziggs sets  $n_{blocks}$  from 8 to 12, which provides a practical balance between data reuse and GPU parallelism.

### 5.4 Effectiveness of Block Scheduling

We evaluate the effectiveness of the proposed block scheduling strategy by comparing three scheduling methods on an A100 GPU: (i) **No reuse**, (ii) **Heuristic**, which uses high-frequency row selection and column-aware ordering to capture dominant reuse patterns,

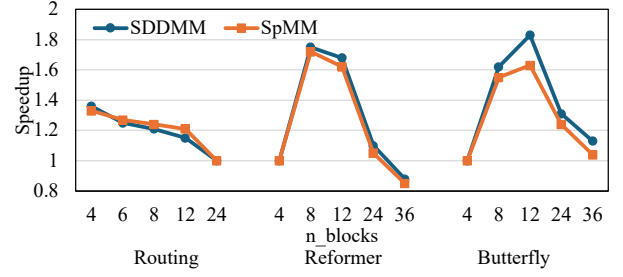


Figure 15: Effect of varying  $n_{blocks}$  on the computational performance of SDDMM and SpMM across GPT-2 Large (Routing), GPT-3 6.7B (Reformer), and Yi 6B (Butterfly).

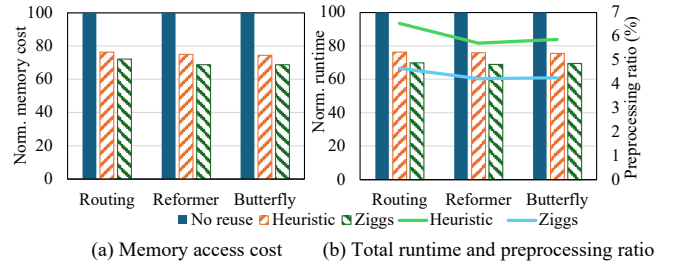


Figure 16: Effectiveness of different block scheduling strategies for SDDMM on the LLaMA 3 70B model.

and (iii) **Ziggs**, which uses an offline Q-learning scheduler to build a pattern-to-order lookup table for frequently occurring patterns and falls back to the heuristic for patterns not covered by the table. All three methods use the same block grouping, so the comparison isolates the effect of block scheduling.

Figure 16a compares the normalized memory access cost of the three scheduling strategies. Compared with **No reuse**, the heuristic strategy already achieves a substantial reduction in memory cost, lowering it to around 75% on average by exploiting dominant row and column reuse opportunities. **Ziggs** further reduces the average normalized memory cost to about 70%, showing that the offline learned pattern-to-order table can select better execution orders than the heuristic-only strategy for common block group patterns. This result indicates that effective inter-block reuse requires more than locally favorable row or column locality.

Figure 16b shows the normalized runtime together with the preprocessing overhead ratio. The heuristic strategy reduces normalized runtime to about 72% – 75%, while Ziggs further lowers it to around 68% – 70%. At the same time, the preprocessing ratio decreases from 5.72% – 6.55% for the heuristic strategy to 4.23% – 4.66% for Ziggs, showing that the learned pattern-to-order table not only improves the execution order but also reduces runtime scheduling overhead. Therefore, Ziggs consistently achieves lower total runtime than the heuristic-only strategy across all sparse patterns. For training patterns where exhaustive search is tractable, the MDP schedules achieve the same minimum memory-access count as the brute-force optimum, although the exact orders may differ when multiple optimal schedules exist. We observe similar scheduling

trends on the other evaluated models and report LLaMA 3 here due to space constraints.

## 6 Related Work

**Efficient Transformers.** There are several approaches [7, 9, 11, 18] that have been proposed to simplify the Transformer model for efficient training. The most widely studied is attention score matrix sparsification. Child et al. [7] introduced sparse transformer by choosing some local and strided attention patterns to reduce computational and memory overhead while allowing models to capture long-range dependencies. Adaptive Sparse Transformer [9] dynamically prunes or allocates the connections of attention score based on content information for higher accuracy, rather than relying on a fixed pattern. FlashAttention-2 [11] is an optimized implementation of the attention mechanism that reduces memory usage and improves GPU throughput. It is particularly effective for long sequences and high-dimensional inputs. nmSPARSE [30] proposes an N:M block-sparse format to better support GPU acceleration, but it is limited to N:M sparsity and cannot be applied to dynamic block-sparse patterns. DeepSpeed [45] introduces viable sparse patterns but it still lacks inter-block data reuse. xFormers [28] supports block-diagonal and sliding-window sparse attention, which lacks the inter-block data reuse.

**Fused Sparse Attention and Tensor Core Kernels.** Recent block-sparse attention implementations improve performance through kernel fusion and specialized GPU execution. Block-Sparse Attention [20] and FlexAttention [13] fuse attention operations to reduce intermediate data movement, while FlashSparse [48] accelerates sparse SDDMM and SpMM using Tensor Cores. These methods mainly optimize fusion and intra-kernel execution, whereas Ziggs focuses on inter-block data reuse through cache-aware scheduling.

**Sparse Matrix Operations.** Many studies [22, 29, 39, 49] have focused on optimizing the internal data layout of individual blocks to align with the memory hierarchy for achieving optimal performance. They employ common CUDA kernel optimization techniques to enhance computation within each block. For instance, tiling [22, 38] is used to improve data locality, and coalesced memory access is applied to reduce global memory access overhead. Sputnik [15] proposed the reverse offset memory alignment (ROMA) technique for regularizing sparse rows to enable vector memory instructions, along with the row swizzle technique for load balancing on GPUs. Additionally, techniques such as double buffering [29, 38] are utilized to overlap data loading and computation, and vectorized memory operations [37, 38] are implemented to reduce the number of instructions executed within the kernel. While these optimizations have proven effective in many scenarios, they still face bottlenecks when applied to block-sparse Transformers.

## 7 Conclusion

In this work, we present Ziggs, a reuse-aware framework for accelerating block-sparse Transformers on GPUs. Ziggs combines offline reuse-aware block scheduling, lightweight metadata encoding, and optimized SDDMM/SpMM kernels to systematically expose and exploit inter-block reuse in sparse attention. By storing common execution orders in a pattern-to-order table and using a lightweight heuristic fallback for unseen patterns, Ziggs reduces redundant

memory accesses while preserving GPU parallelism and maintaining low runtime overhead. Experimental results show that Ziggs achieves up to 2.05× throughput improvement and reduces global memory traffic by up to 1.44× over state-of-the-art block-sparse attention baselines, while delivering 1.61–3.15× end-to-end speedup over PyTorch.

## Acknowledgments

We thank the anonymous reviewers for their constructive and insightful feedback. This research is supported in part by the National Science Foundation under Grants CNS-2310423, CNS-2310422, and CNS-2152497.

## References

- [1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, TX, USA, November 13–18, 2022, Felix Wolf, Sameer Shende, Candace Culhane, Sadaf R. Alam, and Heike Jagode (Eds.). IEEE, 46:1–46:15. doi:10.1109/SC41404.2022.00051
- [2] Charles W. Anderson. 1992. Q-Learning with Hidden-Unit Restarting. In *Advances in Neural Information Processing Systems 5*, [NIPS Conference, Denver, Colorado, USA, November 30 - December 3, 1992], Stephen Jose Hanson, Jack D. Cowan, and C. Lee Giles (Eds.). Morgan Kaufmann, 81–88. <http://papers.nips.cc/paper/597-q-learning-with-hidden-unit-restarting>
- [3] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *arXiv:2004.05150* (2020).
- [4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. (2020). *arXiv:2005.14165* [cs.CL]
- [5] Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Ré. 2022. Pixelated Butterfly: Simple and Efficient Sparse training for Neural Network Models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25–29, 2022*. OpenReview.net. <https://openreview.net/forum?id=NL-iXa-y7R>
- [6] Wenhua Cheng, Weiwei Zhang, Hailiao Shen, Yiyang Cai, Xin He, Kaokao Lv, and Yi Liu. 2024. Optimize Weight Rounding via Signed Gradient Descent for the Quantization of LLMs. *arXiv:2309.05516* [cs.CL] <https://arxiv.org/abs/2309.05516>
- [7] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. *CoRR abs/1904.10509* (2019). *arXiv:1904.10509* <http://arxiv.org/abs/1904.10509>
- [8] Krzysztof Marcin Choromanski, Valerii Likhoshesterov, David Dohan, Xingyou Song, Andreea Gane, Tamás Szilvási, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J. Colwell, and Adrian Weller. 2021. Rethinking Attention with Performers. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3–7, 2021*. OpenReview.net. <https://openreview.net/forum?id=Ua6zUK0WRH>
- [9] Gonçalo M. Correia, Vlad Niculae, and André F. T. Martins. 2019. Adaptively Sparse Transformers. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3–7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 2174–2184. doi:10.18653/V1/D19-1223
- [10] Baiyun Cui, Yingming Li, Ming Chen, and Zhongfei Zhang. 2019. Fine-tune BERT with Sparse Self-Attention Mechanism. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3–7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 3546–3551. doi:10.18653/V1/D19-1361
- [11] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *arXiv:2307.08691* [cs.LG] <https://arxiv.org/abs/2307.08691>
- [12] Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, and Furu Wei. 2023. LongNet: Scaling Transformers to 1,000,000,000 Tokens. In *Proceedings of the 10th International Conference on Learning Representations*.

- [13] Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. 2024. Flex Attention: A Programming Model for Generating Optimized Attention Kernels. *CoRR* abs/2412.05496 (2024). arXiv:2412.05496 doi:10.48550/ARXIV.2412.05496
- [14] Ryan Eberhardt and Mark Hoemmen. 2016. Optimization of Block Sparse Matrix-Vector Multiplication on Shared-Memory Parallel Architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 663–672. doi:10.1109/IPDPSW.2016.42
- [15] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. doi:10.1109/SC41405.2020.00021
- [16] Yair Ori Gat, Nitay Calderon, Amir Feder, Alexander Chapanin, Amit Sharma, and Roi Reichart. 2024. Faithful Explanations of Black-box NLP Models Using LLM-generated Counterfactuals. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7–11, 2024*. OpenReview.net. <https://openreview.net/forum?id=UMfcdRIotC>
- [17] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, and Abhishek Kadian et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] <https://arxiv.org/abs/2407.21783>
- [18] Scott Gray, Alec Radford, and Diederik P Kingma. 2017. Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224* 3, 2 (2017), 2.
- [19] Zijing Gu. 2020. Optimizing Block-Sparse Matrix Multiplications on CUDA with TVM. arXiv:2007.13055 [cs.MS] <https://arxiv.org/abs/2007.13055>
- [20] Junxian Guo, Haotian Tang, Shang Yang, Zhekai Zhang, Zhijian Liu, and Song Han. 2024. Block Sparse Attention. <https://github.com/mit-han-lab/Block-Sparse-Attention>.
- [21] Gregory Holste, Ziyu Jiang, Ajay Jaiswal, Maria Hanna, Shlomo Minkowitz, Alan C. Legasto, Joanna G. Escalon, Sharon Steinberger, Mark Bittman, Thomas C. Shen, Ying Ding, Ronald M. Summers, George Shih, Yifan Peng, and Zhangyang Wang. 2023. How Does Pruning Impact Long-Tailed Multi-label Medical Image Classifiers?. In *Medical Image Computing and Computer Assisted Intervention - MICCAI 2023 - 26th International Conference, Vancouver, BC, Canada, October 8–12, 2023, Proceedings, Part V (Lecture Notes in Computer Science, Vol. 14224)*, Hayit Greenspan, Anant Madabhushi, Parvin Mousavi, Septimiu E. Salcudean, James Duncan, Tanveer F. Syeda-Mahmood, and Russell H. Taylor (Eds.). Springer, 663–673. doi:10.1007/978-3-031-43904-9\_64
- [22] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadaayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16–20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 300–314. doi:10.1145/3293883.3295712
- [23] Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir 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 *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10–15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). [http://papers.nips.cc/paper\\_files/paper/2024/hash/5dfbe6f5671e82c76841ba687a8a9ecb-Abstract-Conference.html](http://papers.nips.cc/paper_files/paper/2024/hash/5dfbe6f5671e82c76841ba687a8a9ecb-Abstract-Conference.html)
- [24] Peng Jiang, Changwan Hong, and Gagan Agrawal. 2020. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22–26, 2020*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 376–388. doi:10.1145/3332466.3374546
- [25] Peng Jiang, Lihan Hu, and Shihui Song. 2022. Exposing and Exploiting Fine-Grained Block Structures for Fast and Accurate Sparse Training. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28–December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). [http://papers.nips.cc/paper\\_files/paper/2022/hash/fa69e968b7319fd42524feb41475fb3-Abstract-Conference.html](http://papers.nips.cc/paper_files/paper/2022/hash/fa69e968b7319fd42524feb41475fb3-Abstract-Conference.html)
- [26] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S3: increasing GPU utilization during generative inference for higher throughput. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (New Orleans, LA, USA) (NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 791, 13 pages.
- [27] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The Efficient Transformer. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26–30, 2020*. OpenReview.net. <https://openreview.net/forum?id=rkgNKKHtVB>
- [28] Benjamin Lefaudeux, Francisco Massa, Diana Liskovitch, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. 2022. xFormers: A modular and hackable Transformer modelling library. <https://github.com/facebookresearch/xformers>.
- [29] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (Washington, District of Columbia) (PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 229–241. doi:10.1145/3293883.3295734
- [30] Bin Lin, Ningxin Zheng, Lei Wang, Shijie Cao, Lingxiao Ma, Quanlu Zhang, Yi Zhu, Ting Cao, Jilong Xue, Yuqing Yang, and Fan Yang. 2023. Efficient GPU Kernels for N:M-Sparse Weights in Deep Learning. In *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen (Eds.), Vol. 5. Curran, 513–525. [https://proceedings.mlsys.org/paper\\_files/paper/2023/file/a10deb4d5227a8ea307ea8f3cb712f4-Paper-mlsys2023.pdf](https://proceedings.mlsys.org/paper_files/paper/2023/file/a10deb4d5227a8ea307ea8f3cb712f4-Paper-mlsys2023.pdf)
- [31] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A Co-Design Framework for Enabling Sparse Attention using Reconfigurable Architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (Virtual Event, Greece) (MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 977–991. doi:10.1145/3466752.3480125
- [32] Xiaoyang Lu, Lihan Hu, Hongrui Huang, Peng Jiang, and Xian-He Sun. 2026. I/O-Aware PIM Acceleration for Long-Sequence LLM Inference with Hybrid Sparse Attention. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2026, New Orleans, LA, USA, May 25–29, 2026*. IEEE, 445–457. doi:10.1109/IPDPS65963.2026.00046
- [33] Xiaoyang Lu, Boyu Long, Xiaoming Chen, Yinhe Han, and Xian-He Sun. 2026. I/O Analysis is All You Need: An I/O Analysis for Long-Sequence Attention. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2026, Pittsburgh, PA, USA, March 22–26, 2026*, Benjamin C. Lee, Harry Xu, Mark Silberstein, and Bingyao Li (Eds.). ACM, 962–977. doi:10.1145/3779212.3790174
- [34] Matt Mahoney. 2011. Large text compression benchmark. <https://mattmahoney.net/dc/textdata.html>
- [35] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. arXiv:1609.07843 [cs.CL]
- [36] Maxim Milakov and Natalia Gimelshein. 2018. Online normalizer calculation for softmax. *CoRR* abs/1805.02867 (2018). arXiv:1805.02867 <http://arxiv.org/abs/1805.02867>
- [37] NVIDIA. 2025. CUDA C++ Programming Guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- [38] NVIDIA. 2025. Matrix Multiplication Background User's Guide. <https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html>
- [39] NVIDIA. 2025. NVIDIA cuSPARSE library. [https://docs.nvidia.com/cuda/pdf/CUSPARSE\\_Library.pdf](https://docs.nvidia.com/cuda/pdf/CUSPARSE_Library.pdf)
- [40] Meng Pang, Xiang Fei, Peng Qu, Youhui Zhang, and Zhaolin Li. 2024. A Row Decomposition-based Approach for Sparse Matrix Multiplication on GPUs. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (Edinburgh, United Kingdom) (PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 377–389. doi:10.1145/3627535.3638470
- [41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- [42] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *Proceedings of the Sixth Conference on Machine Learning and Systems, MLSys 2023, Miami, FL, USA, June 4–8, 2023*, Dawn Song, Michael Carbin, and Tianqi Chen (Eds.). mlsys.org. [https://proceedings.mlsys.org/paper\\_files/paper/2023/hash/c4be71ab8d24cdfb45e3d06dbfca2780-Abstract-mlsys2023.html](https://proceedings.mlsys.org/paper_files/paper/2023/hash/c4be71ab8d24cdfb45e3d06dbfca2780-Abstract-mlsys2023.html)
- [43] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).
- [44] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, Chloe Hillier, and Timothy P Lillicrap. 2019. Compressive Transformers for Long-Range Sequence Modelling. *arXiv preprint* (2019). <https://arxiv.org/abs/1911.05507>
- [45] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (Virtual Event, CA, USA) (KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. doi:10.1145/3394486.3406703
- [46] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. 2021. Efficient Content-Based Sparse Attention with Routing Transformers. *Trans. Assoc. Comput. Linguistics* 9 (2021), 53–68. doi:10.1162/TACL\_A\_00353

- [47] Kimia Saedi, Aditya Desai, Apoorv Walia, Jihyeong Lee, Keren Zhou, and Anshumali Shrivastava. 2024. SS1: Accelerating Inference with Fast and Expressive Sketch Structured Transform. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 48921–48954. [https://proceedings.neurips.cc/paper\\_files/paper/2024/file/57ef0373c890b30407eadfe6e06c8c84-Paper-Conference.pdf](https://proceedings.neurips.cc/paper_files/paper/2024/file/57ef0373c890b30407eadfe6e06c8c84-Paper-Conference.pdf)
- [48] Jinliang Shi, Shigang Li, Youxuan Xu, Rongtian Fu, Xueying Wang, and Tong Wu. 2025. FlashSparse: Minimizing Computation Redundancy for Fast Sparse Matrix Multiplications on Tensor Cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, USA) (PPoPP '25). Association for Computing Machinery, New York, NY, USA, 312–325. doi:10.1145/3710848.3710858
- [49] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (Eds.). ACM, 10–19. doi:10.1145/3315508.3329973
- [50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4–9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html>
- [51] Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, Kaidong Yu, Peng Liu, Qiang Liu, Shawn Yue, Senbin Yang, Shiming Yang, Tao Yu, Wen Xie, Wenhao Huang, Xiaohui Hu, Xiaoyi Ren, Xinyao Niu, Pengcheng Nie, Yuchi Xu, Yudong Liu, Yue Wang, Yuxuan Cai, Zhenyu Gu, Zhiyuan Liu, and Zonghong Dai. 2024. Yi: Open Foundation Models by 01.AI. *CoRR* abs/2403.04652 (2024). arXiv:2403.04652 doi:10.48550/ARXIV.2403.04652
- [52] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big Bird: Transformers for Longer Sequences. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6–12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). <https://proceedings.neurips.cc/paper/2020/hash/c8512d142a2d849725f31a9a7a361ab9-Abstract.html>
- [53] Zhaoyang Zhang, Wenqi Shao, Yixiao Ge, Xiaogang Wang, Jinwei Gu, and Ping Luo. 2024. Cached transformers: improving transformers with differentiable memory cache. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'24/IAAI'24/EAAI'24)*. AAAI Press, Article 1888, 9 pages. doi:10.1609/aaai.v38i15.29636
- [54] Lei Zhu, Xinjiang Wang, Zhanghan Ke, Wayne Zhang, and Rynson Lau. 2023. BiFormer: Vision Transformer with Bi-Level Routing Attention. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 10323–10333. doi:10.1109/CVPR52729.2023.00995