Fused3S: Fast Sparse Attention on Tensor Cores

Zitong Li
University of California, Irvine
Irvine, USA
zitongl5@uci.edu

Aparna Chandramowlishwaran
University of California, Irvine
Irvine, USA
amowli@uci.edu

Abstract

Sparse attention is a core building block in many leading neural network models, from graph-structured learning to sparse sequence modeling. It can be decomposed into a sequence of three sparse matrix operations (3S): sampled dense-dense matrix multiplication (SDDMM), softmax normalization, and sparse matrix multiplication (SpMM). Efficiently executing the 3S computational pattern on modern GPUs remains challenging due to (a) the mismatch between unstructured sparsity and tensor cores optimized for dense operations, and (b) the high cost of data movement. Previous works have optimized these sparse operations individually or addressed one of these challenges. This paper introduces Fused3S, the first fused 3S algorithm that jointly maximizes tensor core utilization and minimizes data movement. Across real-world graph datasets, Fused3S achieves $1.6-16.3\times$ and $1.5-14\times$ speedup over state-of-the-art on H100 and A30 GPUs. Furthermore, integrating Fused3S into Graph Transformer inference accelerates end-to-end performance by $1.05-5.36\times$, consistently outperforming all 3S baselines across diverse datasets (single and batched graphs) and GPU architectures.

CCS Concepts

• **Computing methodologies** → **Neural networks**; **Massively parallel algorithms**.

Keywords

Transformers, Sparse Attention, Graph Neural Networks, Long Sequence Modeling, Tensor Core, Kernel Fusion

ACM Reference Format:
Zitong Li and Aparna Chandramowlishwaran. 2025. Fused3S: Fast Sparse Attention on Tensor Cores. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3721145.3730430

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3730430

1 Introduction

Attention has become fundamental in machine learning models from transformers [35] to graph neural networks (GNNs) [5, 34, 36]. However, its computational cost remains a bottleneck as we scale in sequence length and graph size. While dense and block-sparse attention have benefited from hardware-aware algorithm design [3, 4], sparse attention–essential for graph-based learning and dynamic sparsity patterns–remains under-optimized on modern hardware accelerators. This inefficiency is especially pronounced on GPUs with tensor cores, which deliver peak throughput for dense matrix multiplications (or limited structured sparsity such as 2:4) with strict operand shapes. In contrast, sparse operations involve irregular memory accesses and unstructured computation, making it poorly suited for current tensor core design. As a result, tensor cores remain largely underutilized for sparse workloads.

Sparse attention can be decomposed into a sequence of three operations: Sampled Dense-Dense Matrix Multiplication (SDDMM) to compute attention scores, softmax normalization, and Sparse Matrix Multiplication (SpMM) to aggregate features. We refer to this computational pattern as **3S**, which recurs in GNNs [5, 34, 36], sparse transformers [18, 22], and models that exploit dynamic sparsity.

Prior efforts to accelerate the 3S pattern fall into two broad categories: (1) *Individual kernel optimizations*, which improves the performance of one or more sparse operations (such as SDDMM and/or SpMM) in isolation using specialized sparse tensor formats and kernel-local optimizations [7, 13, 17, 20, 27, 32, 38, 40, 46]. These approaches incur unnecessary data movement when intermediate results are materialized in global memory. (2) *Kernel fusion*, which reduces memory traffic by combining the 3S operations into a single kernel. However, existing fused kernels for sparse attention are designed either for CPUs [29] or CUDA cores [21], leaving tensor core acceleration untapped. As summarized in Table 1, no existing work fuses the 3S operations while targeting tensor cores.

To bridge this gap, we propose **Fused3S**, the first fused sparse attention algorithm and kernel designed for GPU tensor cores. Fused3S: (1) adopts a block-structured sparse format aligned with tensor core operand shapes, (2) fuses SDDMM, softmax, and SpMM into a single kernel to reuse intermediate results in registers and shared memory, and

Table 1: Summary of algorithms designed for 3S or its sub-operations (SDDMM or SpMM).

Method	Hardware	Format	Precision	Kernels		Fusion	3S
				SDDMM	SpMM		
Sputnik [9]	CUDA	CSR	fp16, fp32	●	●	●	●
RoDe [28]	CUDA	CSR	fp32, fp64	●	●	●	●
JigSaw[44]	SPTC	Reorder-aware	fp16	●	●	●	●
TCA-SpMM[13]	TC	CSR	fp16/fp32	●	●	●	●
Magicube [20]	TC	SR-BCRS	int16, int8, int4	●	●	●	●
SMaT[27]	TC	BCSR	fp16	●	●	●	●
BSA-SpMM [17]	TC	CSR, Blocked-ELL	fp16	●	●	●	●
Flash-LLM [40]	TC	Tiled-CSL	fp16/fp32	●	●	●	●
TC-GNN[38]	TC	TCF	tf32	●	●	●	●
DTC-SpMM[7]	TC	ME-TCF	tf32	●	●	●	●
Acc-SpMM[46]	TC	BitTCF	tf32	●	●	●	●
FlashSparse[32]	TC	ME-BCRS	fp16/tf32	●	●	●	●
FusedMM[29]	CPU	CSR	fp32, fp64	●	●	●	●
DF-GNN[21]	CUDA	CSR+COO, CSC	fp32	●	●	●	●
Fused3S (this paper)	TC	BSB	fp16/fp32	●	●	●	●

(3) implements a mixed precision pipeline with numerically stable online softmax to maximize performance while maintaining accuracy.

Our contribution can be summarized as follows.

- **Fused3S**[1], an open-source kernel that simultaneously exploits kernel fusion *and* tensor core utilization for the 3S sparse computational pattern.
- The fused algorithm is designed to be fully on-chip with high parallelism. This is achieved using multi-level tiling with efficient block- and warp-level work partitioning to avoid global-memory synchronization. Reordering and register-level remapping optimizations improve load balance and memory accesses for irregular graphs.
- Across real-world graph datasets, Fused3S achieves $1.6-16.3\times$ and $1.5-14\times$ speedups over DF-GNN [21], FlashSparse [32], and PyG [8] on H100 and A30 GPUs respectively.
- Integrated into the Graph Transformer [5] implemented in DGL [37], Fused3S achieves $1.05-5.36\times$ end-to-end inference speedup over state-of-the-art 3S baselines across graph datasets and GPUs.

2 Background

2.1 Computational Pattern in Sparse Attention

A common computation in machine learning models from graph-structured learning to sparse sequence modeling is the **3S** pattern: a sequence of SDDMM, softmax normalization, and SpMM. 3S can be formulated as:

$$\mathbf{O} = \text{softmax}(\mathbf{Q}\mathbf{K}^T \odot \mathbf{A})\mathbf{V} \tag{1}$$

where $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$, and $\mathbf{O} \in \mathbb{R}^{N\times d}$ are dense matrices and $\mathbf{A} \in \mathbb{R}^{N\times N}$ is a sparse matrix that defines attention patterns (e.g., adjacency or masking). Equation 1 can be decomposed into three operations:

(1) **SDDMM:** Compute attention scores $\mathbf{S} = \mathbf{Q}\mathbf{K}^T \odot \mathbf{A}$, where the dense-dense multiplication $\mathbf{Q}\mathbf{K}^T$ is computed only for non-zeros in $\mathbf{A}$.
(2) **Softmax:** Normalize the scores row-wise $\mathbf{E} = \text{softmax}(\mathbf{S})$.
(3) **SpMM:** Aggregate output $\mathbf{O} = \mathbf{E}\mathbf{V}$.

This 3S pattern appears in several popular architectures.

Graph Attention Network (GAT). In GATs [36], nodes in a graph attend selectively to their neighbors using $\mathbf{A}$ as the adjacency matrix. A typical formulation of GAT attention is:

$$\mathbf{O} = \text{softmax}\Big(\text{LeakyReLU}([\mathbf{W}\mathbf{H}||\mathbf{W}\mathbf{H}]) \odot \mathbf{A}\Big)(\mathbf{W}\mathbf{H}), \tag{2}$$

where $\mathbf{H}$ is the input node features, $\mathbf{W}$ is a learnable weight matrix, and $||$ denotes concatenation. (1) SDDMM computes the unnormalized attention coefficients between nodes. (2) Softmax normalizes the attention coefficients across all neighbors of a node. (3) SpMM aggregates the transformed features of neighboring nodes, weighted by the normalized attention coefficients.

Attention-based Graph Neural Network (AGNN). AGNN [34] introduces a dynamic, adaptive attention. We can formulate AGNN as:

$$\mathbf{O} = \text{softmax}\Big(\beta^{(t)} \cos(\mathbf{H}^{(t)}, \mathbf{H}^{(t)^T}) \odot \mathbf{A}\Big)\mathbf{H}^{(t)} \tag{3}$$

[1] https://github.com/HPCForge/Fused3S

where $\mathbf{A}$ includes self-loops, $\beta^{(t)}$ is a learnable parameter for layer t, and $\cos(\cdot,\cdot)$ denotes the cosine similarity. Here, $\mathbf{Q} = \mathbf{K} = \mathbf{V} = \mathbf{H}^{(t)}$, with the SDDMM step computing scaled cosine similarities.

Graph Transformer (GT). GTs [5, 33, 45] extend attention to entire graphs, treating nodes as tokens. A representative formulation is:

$$\mathbf{O} = \text{softmax}\Big((\mathbf{W}_Q\mathbf{H})(\mathbf{W}_K\mathbf{H})^T \odot \mathbf{A}\Big)(\mathbf{W}_V\mathbf{H}), \qquad (4)$$

where $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ are learnable projections for queries, keys, and values. Unlike standard transformers, GTs explicitly encode graph structure through $\mathbf{A}$.

Recent surveys [23, 31] note that many GT models use dense global attention, augmented with structural bias (e.g., node degrees, shortest paths) to avoid over-smoothing and improve expressivity. However, this approach can be computationally expensive for large graphs, motivating the need for sparse attention.

Sparse Transformers. Sparse transformers [2] reduce the quadratic complexity of attention by applying a sparse mask $\mathbf{M}$.

$$\mathbf{O} = \text{softmax}\Big((\mathbf{W}_Q\mathbf{X})(\mathbf{W}_K\mathbf{X})^T \odot \mathbf{M}\Big)(\mathbf{W}_V\mathbf{X}), \qquad (5)$$

where $\mathbf{X}$ is the input sequence. The mask $\mathbf{M}$ determines token interactions and may be static or dynamically generated. Static masks [1, 42] impose structured sparsity (e.g., block-diagonal and block-sparse) that is GPU friendly. Dynamic variants [18, 22] compute $\mathbf{M}$ on-the-fly enabling context-aware sparsity. While dynamic masks often improve accuracy, they introduce irregular sparsity that is difficult to optimize efficiently.

Although GATs use fixed $\mathbf{A}$ and sparse transformers generate $\mathbf{M}$ dynamically, these diverse models share the same 3S bottleneck: computing and applying sparse attention on modern hardware accelerators. Our work targets this unifying 3S abstraction to develop a hardware-optimal algorithm.

2.2 Tensor Core and Operand Shapes

Tensor Cores (TCs) are specialized hardware units on NVIDIA GPUs designed for high-throughput matrix multiply and accumulate operations. Since their introduction in 2017, FP16 throughput using TCs has increased from 125 TFLOPS on V100 to 990 TFLOPS on H100, an improvement of 8× in 5 years [24, 26]. This rapid progression has significantly improved the performance of dense matrix computations.

There are two primary programming interfaces for TCs: CUDA `wmma` (Warp Matrix Multiply Accumulate) and PTX `mma` (Matrix Multiply Accumulate) instructions. The choice of interface depends on specific optimization goals. The PTX `mma` is a lower-level interface that allows direct operand loading from global memory (HBM) into registers, bypassing shared memory. This provides finer-grained control and can be advantageous for workloads with limited data reuse. In contrast, `wmma` operates at a higher abstraction level, requiring both input matrices to be explicitly staged in shared memory before loading into registers.

Table 2: Precision formats and operand shapes on Tensor Cores. Here m, n, k denote tile dimensions for matrix multiplication.

Precision	Type	Operand Shapes
FP16	wmma	m16n16k16, m8n32k16, m32n8k16
	mma	m8n8k4, m16n8k8, **m16n8k16**
BF16	wmma	m16n16k16, m8n32k16, m32n8k16
	mma	m16n8k8, **m16n8k16**
TF32	wmma	m16n16k8
	mma	m16n8k4, m16n8k8
FP8	wmma	–
	mma	m16n8k32, **m16n8k16**

TCs support various operand shapes and precision formats summarized in Table 2. These shapes dictate how input matrices are partitioned into tiles and strongly influence performance. For sparse computations, the optimal tile size may not always align with the hardware's peak capability. The architectural trend towards "dense TCs" suggests larger tile shapes to maximize TC utilization but this can lead to more zero computations when applied naively to sparse matrices. Smaller tile shapes reduce the occurrence of zeros and improve compute density on sparse data, but may result in underutilized TCs. Among the available configurations, the `m16n8k16` tile shape emerges as a practical compromise, supported in multiple precision formats (FP16, BF16, FP8).

3 Fused3S

This section presents the design of Fused3S. We first describe how the sparse matrix $\mathbf{A}$ is stored in a block-structured format tailored to tensor core operand shapes. Then, we detail the Fused3S kernel resulting in Algorithm 1 and highlight the key optimizations:

- Fusing the 3S operations (SDDMM, Softmax, SpMM) into a single kernel using multi-level tiling/blocking to reduce memory traffic and enable on-chip data reuse.
- Incremental softmax computation to support large attention matrices.
- Warp-level parallelism for SIMT-friendly execution.
- Permuted data layouts and register-level remapping to enable coalesced memory access patterns.

3.1 Sparse Format for Tensor Cores

We introduce the **Binary Sparse Block (BSB)** format to efficiently map a sparse matrix A onto tensor cores. BSB

extends the Memory-Efficient Tensor Core Format (ME-TCF) [7], which in itself builds on the TC-GNN Compressed Format (TCF) [38]. Like block-CSR (BCSR), these formats use a block layout with local indexing but are specifically designed for tensor core operand shapes.

The construction of the BSB format proceeds as follows and is illustrated in Figure 1:

(1) Divide the sparse matrix into *row windows* (RW) of size r.
(2) Within each RW, eliminate columns containing only zeros to increase compute density.
(3) Partition the compacted RW into *tensor core blocks* (TCB) of shape $r \times c$, where r and c match supported `mma` tile sizes (e.g., 16×8 in Table 2).
(4) We maintain three data structures:
   - `tcb_row_offset (tro)`: Number of TCBs per RW.
   - `col_sparse_to_dense (sptd)`: Mapping from original to compacted column indices per RW.
   - `bitmap`: A fixed-size bitmask encoding the sparsity pattern in each TCB.

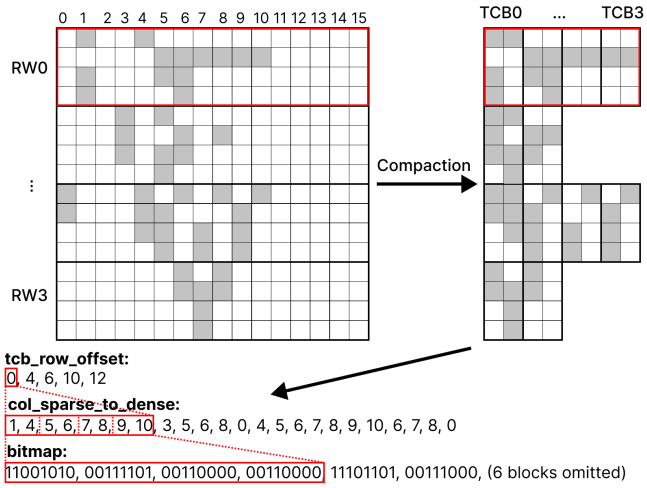


Figure 1: Binary Sparse Block (BSB) format. The TCB size in this example is 4×2 while in practice the size is larger (i.e., 16×8). Red boxes highlight how the first row window in compacted, tiled, and stored in BSB format.

ME-TCF uses two arrays to store non-zero elements: one for the number of non-zero elements in each TCB and another to store the local index of each nonzero element in all TCBs. We make the observation that adjacency matrices and attention masks exhibit binary-valued sparsity. Unlike ME-TCF and TCF, which represent the location of nonzeros using integer indices, BSB encodes the $r \times c$ TCB using a single binary bitmap. For example, a 16×8 TCB requires only 128 bits to represent its sparsity pattern, eliminating indexing overhead.

Table 3 compares the memory footprint of various sparse formats. The formats differ in how they organize sparsity (row-based vs. block-based) and whether they store explicit nonzero values. Row-based formats (such as CSR) are incompatible with tensor cores due to irregular access patterns. General-purpose block formats such as BCSR [15] and its variants [20, 27, 32] improve locality but explicitly store both nonzero values and their positions. In contrast, formats such as TCF [38], ME-TCF [7], BitTCF [46], and our BSB are designed for tensor cores. These formats align blocks with MMA tile shapes and assume binary sparsity. BSB further reduces overhead by encoding block sparsity with a fixed-size bitmap. Unlike BCSR, BSB compacts columns within row windows to increase density and reduce the total number of blocks.

Table 3: Comparison of sparse formats. `row`: row-based, `blk`: generic block-based, `mma`: MMA-tile-aligned. Matrix size is $N \times N$ with z nonzeros. Row window height is r; b: number of blocks, bc: stored columns after compaction (if any), and rc: elements per block. Sizes assume 32-bit indices and values unless format is binary.

Format	Type	Memory Footprint	NZ Value
CSR	row	$32(N + 2z)$	fp32
SR-BCSR	blk	$32(\frac{2N}{r} + bc + brc)$	fp32
ME-BCRS	blk	$32(\frac{N}{r} + bc + brc)$	fp32
BCSR	blk	$32(\frac{N}{r} + b + brc)$	fp32
TCF	mma	$32(\frac{N}{r} + N + 3z)$	binary
ME-TCF	mma	$32(\frac{N}{r} + b + z) + 8z$	binary
BitTCF	mma	$32(\frac{N}{r} + b + z) + z$	binary
BSB (ours)	mma	$32(\frac{N}{r} + bc) + brc$	binary

3.2 Fusion and Thread-block Parallelization

Algorithm 1 describes the Fused3S kernel. The input and output matrices $\mathbf{Q}$ and $\mathbf{O}$ are divided into row blocks (line 1-2), each assigned to a thread block. Each thread block loads its corresponding $\mathbf{Q}_i$ into shared memory (line 5), which is reused across TCBs in the row window. The number of TCBs in the i-th row window is determined using `tro` (line 6). Thread blocks then extract the column indices $\mathbf{c}$ that define the nonzero pattern in $\mathbf{A}_i$ using the `sptd` map (line 7). These indices are used to gather rows from $\mathbf{K}$ and $\mathbf{V}$ (line 8), which are then partitioned into warp-aligned blocks (lines 9-10). Unlike $\mathbf{Q}_i$, which is reused across all warps, $\hat{\mathbf{K}}$ and $\hat{\mathbf{V}}$ are only accessed once per row window and loaded directly from HBM into registers without staging in shared memory.

The inner loop (lines 11-23) fuses the 3S operations. SDDMM is executed using a warp-level `TBGemm` (line 13), computing attention scores $\mathbf{S}_i$, which are masked with the sparse

Table 4: Notation used in Algorithm 1.

Symbol	Definition
$\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$	Query, key, value matrices
$\mathbf{A} \in \mathbb{R}^{N \times N}$	Sparse matrix (e.g., adjacency or mask)
$\mathbf{S} \in \mathbb{R}^{N \times N}$	Attention score matrix
$\mathbf{E} \in \mathbb{R}^{N \times N}$	Row-wise normalized score matrix
$\mathbf{O} \in \mathbb{R}^{N \times d}$	Output matrix
$\mathbf{Q}_i \in \mathbb{R}^{r \times d}$	Query block
$\hat{\mathbf{K}}, \hat{\mathbf{V}} \in \mathbb{R}^{tc \times d}$	Gathered rows of key and value matrices
$\mathbf{S}_i \in \mathbb{R}^{r \times cW}$	Attention score block
$\mathbf{E}_i \in \mathbb{R}^{r \times cW}$	Normalized score block
$\mathbf{O}_i \in \mathbb{R}^{r \times d}$	Output block
$\mathbf{m}_o \in \mathbb{R}^r$	Row-wise max scores
$\mathbf{l}_o \in \mathbb{R}^r$	Row-wise softmax normalization factor
r, c	Dimensions of tensor core block
t	Number of TCBs in row window i
W	Number of warps per thread block
N	Number of rows/nodes
d	Feature dimension

Algorithm 1 FUSED3S

Require: $\mathbf{A}$ in BSB format: `tro`, `sptd`, `bitmap`; $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$
Ensure: $\mathbf{O} = \text{softmax}(\mathbf{Q}\mathbf{K}^\top \odot \mathbf{A})\mathbf{V} \in \mathbb{R}^{N \times d}$
1: Divide $\mathbf{Q}$ into $T_r = \lceil \frac{N}{r} \rceil$ blocks $\{\mathbf{Q}_1 \dots \mathbf{Q}_{T_r}\}$, each of size $r \times d$
2: Divide $\mathbf{O}$ into T_r blocks $\{\mathbf{O}_1 \dots \mathbf{O}_{T_r}\}$, each of size $r \times d$
3: **for** $i = 1$ **to** T_r **do**
4: Initialize $\mathbf{m}_o = -\infty$, $\mathbf{l}_o = 0 \in \mathbb{R}^r$, $\mathbf{O}_i = 0 \in \mathbb{R}^{r \times d}$ in `fp32`
5: Load $\mathbf{Q}_i$ from HBM to SMEM
6: $t = \texttt{tro}[i+1] - \texttt{tro}[i]$
7: $\mathbf{c}$ = getColumnVectorIndex(`sptd`, i)
8: $\hat{\mathbf{K}}, \hat{\mathbf{V}} \in \mathbb{R}^{tc \times d}$ = select rows of $\mathbf{K}, \mathbf{V}$ according to $\mathbf{c}$
9: Divide $\hat{\mathbf{K}}$ into $T_c = \lceil \frac{t}{W} \rceil$ blocks $\{\hat{\mathbf{K}}_1 \dots \hat{\mathbf{K}}_{T_c}\}$, each of size $Wc \times d$
10: Divide $\hat{\mathbf{V}}$ into T_c blocks $\{\hat{\mathbf{V}}_1 \dots \hat{\mathbf{V}}_{T_c}\}$, each of size $Wc \times d$
11: **for** $j = 1$ **to** T_c **do**
12: // SDDMM
13: $\mathbf{S}_i = \text{TBGemm}(\mathbf{Q}_i, \hat{\mathbf{K}}_j^\top, \mathbf{0})$
14: Apply `bitmap` mask to $\mathbf{S}_i$
15: // Online Softmax
16: $\mathbf{m}_i = \max(\mathbf{m}_o, \texttt{rowmax}(\mathbf{S}_i))$
17: $\mathbf{E}_i = e^{\mathbf{S}_i - \mathbf{m}_i}$
18: $\mathbf{l}_o = \text{diag}(e^{\mathbf{m}_o - \mathbf{m}_i})\mathbf{l}_o + \texttt{rowsum}(\mathbf{E}_i)$
19: Store $\mathbf{E}_i$ (cast to fp16) in SMEM
20: // SpMM
21: $\mathbf{O}_i = \text{diag}(e^{\mathbf{m}_o - \mathbf{m}_i})\mathbf{O}_i$
22: $\mathbf{O}_i = \text{TBGemm}(\mathbf{E}_i, \hat{\mathbf{V}}_j, \mathbf{O}_i)$
23: $\mathbf{m}_o = \mathbf{m}_i$
24: Write $\mathbf{O}_i = \text{diag}(\mathbf{l}_o)^{-1}\mathbf{O}_i$ to HBM

bitmap from the BSB format (line 14). Softmax is computed incrementally using a numerically stable online variant adapted from FlashAttention-2 [3] (lines 16–18). We track the running row-wise max $\mathbf{m}_o$ and normalization factor $\mathbf{l}_o$, and apply exponential rescaling across blocks to preserve numerical stability and ensure correctness, despite the blockwise computation. All scaling is done in fp32. Normalized scores $\mathbf{E}_i$ are cast to fp16 and stored in shared memory (line 19). SpMM proceeds by rescaling the accumulated output block $\mathbf{O}_i$ and invoking a second `TBGemm` (lines 21–22). After processing all blocks in the row window, the final output block $\mathbf{O}_i$ is normalized and written to HBM (line 24).

Algorithm 2 TBGEMM

Require: MMA tile: (m, n, k); $\mathbf{A} \in \mathbb{R}^{m \times K}$ in SMEM (fp16), $\mathbf{B} \in \mathbb{R}^{K \times P}$ in HBM (fp16), $\mathbf{D} \in \mathbb{R}^{m \times P}$ in SMEM (fp32)
Ensure: $\mathbf{C} = \mathbf{A}\mathbf{B} + \mathbf{D} \in \mathbb{R}^{m \times P}$ in fp32
1: Divide $\mathbf{A}$ into $T_g = \lceil \frac{K}{k} \rceil$ tiles $\{\mathbf{A}_1, \dots, \mathbf{A}_{T_g}\}$, each of size $m \times k$
2: Divide $\mathbf{B}$ into $T_h = \lceil \frac{P}{n} \rceil$ tiles $\{\mathbf{B}_1, \dots, \mathbf{B}_{T_h}\}$, each of size $K \times n$
3: Divide $\mathbf{D}$ into T_h tiles $\{\mathbf{D}_1, \dots, \mathbf{D}_{T_h}\}$, each of size $m \times n$
4: **for** $i = 1$ **to** T_h **do**
5: Load $\mathbf{D}_i$ from SMEM to registers as $\mathbf{C}_i$
6: Divide $\mathbf{B}_i$ into T_g tiles $\{\mathbf{B}_{i1}, \dots, \mathbf{B}_{iT_g}\}$, each of size $k \times n$
7: **for** $j = 1$ **to** T_g **do**
8: Load $\mathbf{A}_j$ from SMEM to registers
9: Load $\mathbf{B}_{ij}$ from HBM to registers
10: $\mathbf{C}_i = \texttt{mma}(\mathbf{A}_j, \mathbf{B}_{ij}, \mathbf{C}_i)$ // Tensor Core operation
11: **return** $\mathbf{C}$

The `TBGemm` kernel in Algorithm 2 is a core primitive used in SDDMM and SpMM (lines 13 and 22). It partitions input blocks into tensor core compatible tiles, loads operands into registers, and issues MMA instructions to perform high-throughput matrix multiply-accumulate.

There are two ways to parallelize the 3S computation: **node-parallel** and **edge-parallel**. The distinction lies in how $\mathbf{S}$, the output of SDDMM, is distributed among thread blocks, as illustrated in Figure 2. Although these terms are coined in the context of graph attention, the concepts apply more broadly to sparse attention.

Node-Parallel Fusion. In node-parallel, each thread block is assigned a fixed set of rows in $\mathbf{S}$ (i.e., a subset of nodes in the graph). As seen in Algorithm 1, softmax requires row-wise reductions (max and sum), which can be computed locally within each thread block. This enables the subsequent SpMM to also be executed within the same thread block. The primary advantage of node-parallel is independence: each thread block owns all data needed for its rows of softmax and SpMM, avoiding inter-block synchronization. This is depicted at the top of Figure 2, where thread blocks operate on disjoint rows.

Edge-Parallel Fusion. Edge-parallel distributes computation across thread blocks based on TCBs (i.e., edge-level granularity). This achieves better load balancing, especially

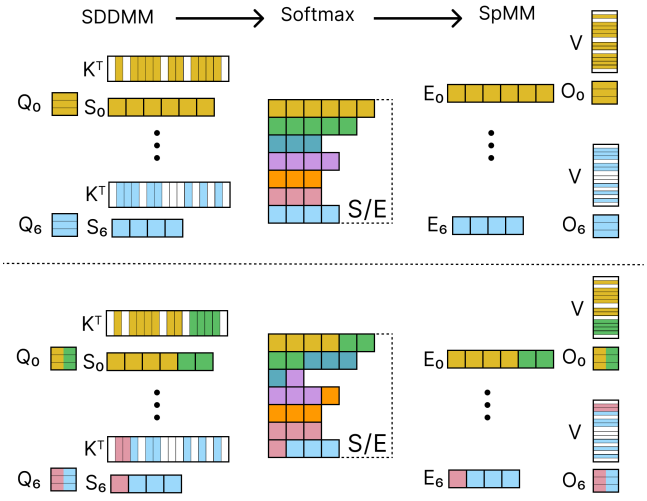


Figure 2: Comparison of node-parallel (top) and edge-parallel (bottom) strategies. Different colored blocks are owned by different thread blocks. In edge-parallel, blocks of Q or O are shaded with multiple colors if shared by multiple thread blocks. The figure is divided vertically into three stages: (1) SDDMM, (2) data distribution of S and E for softmax, and (3) SpMM.

for datasets where the number of TCBs per RW (i.e., node degree in graphs) varies widely (see Table 7). Such variance is common in real-world graphs due to their power-law degree distribution. By evenly distributing TCBs among thread blocks, edge-parallel ensures uniform workload across SMs. However, it introduces significant synchronization overhead. Since rows of **S** (and hence **O**) may be computed by multiple thread blocks, softmax and SpMM must coordinate through global synchronization or atomic updates to HBM–both of which incur performance penalties. As shown in the bottom of Figure 2, rows may be fragmented across multiple thread blocks. Prior work [7] reports that edge-parallel SpMM is 20% slower than node-parallel on average. Since Fused3S fuses softmax and SpMM, it requires an additional global synchronization for softmax, making edge-parallel even less attractive.

Load Balancing via Row Window Reordering. Fused3S fuses the 3S operations into a single kernel to reduce memory traffic, so minimizing global synchronization is important for performance. For this reason, we adopt node-parallel fusion.

By default, we assign each RW to one thread block. In Algorithm 1, this corresponds to the outer loop (line 3), with each iteration executed in parallel by thread blocks. This can lead to load imbalance across thread blocks. We visualize the performance impact in Figure 7, which shows that some SMs remain active long after the others have finished execution.

To mitigate this, we perform *row window reordering*, where RWs are sorted in decreasing order of TCB count. This prioritizes denser RWs earlier in the kernel execution when more RWs are available to be assigned to other thread blocks. Lightweight RWs that complete quickly are deferred to the end. This scheduling policy improves SM utilization and reduces kernel tail latency. Importantly, this reordering is performed during preprocessing, alongside sparse matrix compaction, and adds negligible overhead per input graph. By combining node-parallel fusion with sparse layout optimizations, Fused3S maximizes memory efficiency while maintaining scalability on irregular graphs.

3.3 Warp Partitioning Strategies

We explore two strategies to partition work among warps within a thread block: **split-column** and **split-row**. Figure 3 illustrates these two approaches for SDDMM and SpMM, highlighting each warp's data access pattern and its use of shared memory and registers.

In the split-column scheme (top), the columns of the right-hand-side matrix ($\hat{\mathbf{K}}^\mathsf{T}$ in SDDMM and $\hat{\mathbf{V}}$ in SpMM) are divided among warps. Each warp computes a distinct $r \times c$ tile of the intermediate matrix **S** and output matrix **O**. The advantage of this scheme is that warps operate on independent tiles, eliminating the need for inter-warp synchronization. However, each warp must access the entire $\mathbf{Q}_i$ row block (in SDDMM) or $\mathbf{E}_i$ (in SpMM), increasing memory pressure. The number of active warps in split-column is bounded by the number of TCBs per RW (t in line 9 of Algorithm 1). When t is small, there may be insufficient warp-level parallelism to hide the latency of memory accesses.

In the split-row scheme (bottom), the rows of $\hat{\mathbf{K}}^\mathsf{T}$ and $\hat{\mathbf{V}}$ are partitioned among warps. All warps within a thread block cooperate to compute each $r \times c$ tile of **S** and **O**. This approach reduces the memory footprint per warp—each warp only loads a fragment of $\mathbf{Q}_i$ or $\mathbf{E}_i$. However, this comes at the cost of warp synchronization or atomic operations to aggregate the partial results in shared memory. In addition, the number of warps in split-row is constrained by the feature dimension d. For small d, limited parallelism may reduce the ability to hide the latency of irregular memory accesses of **K** and **V**.

The trade-offs between these schemes involve memory access patterns, synchronization cost, register/shared memory pressure, and degree of parallelism exposed at the warp level. Algorithm 1 is based on node-parallel thread block partitioning (i.e., T_r is partitioned among thread blocks) with split-column warp partitioning (i.e., T_c is partitioned among warps). We chose split-column as the default because, for typical attention workloads, the cost of accessing the entire $r \times d$ row block is often less significant than the cost of inter-warp synchronization. Furthermore, split-column maps naturally

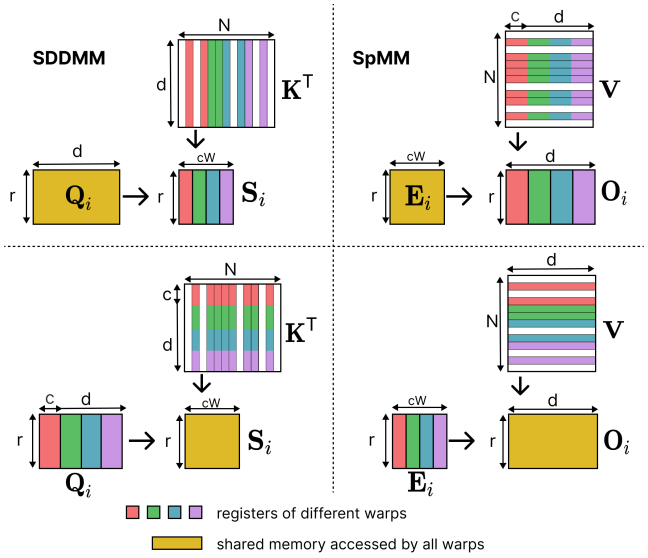


Figure 3: Work partitioning among warps within a thread block. Top: split-column (column blocks of K^T and V are divided among warps). Each warp independently computes a $r \times c$ tile of S and O. Bottom: split-row (row blocks of K^T and V are divided among warps). All warps collaborate to compute each $r \times c$ tile of S and O.

to the SIMT execution model of GPUs, enabling efficient parallel computation on independent tiles.

3.4 Data Layout and Memory Accesses

Efficient memory access is important for high performance, especially for sparse operations, which are often memory-bandwidth bound. We analyze the memory access patterns in SDDMM and SpMM at the thread block and warp levels, and describe a permuted data layout to improve memory coalescing.

SDDMM. Within each thread block, all warps access the same row block of Q (see Figure 3). Given this data reuse, $\mathbf{Q}_i$ is copied from HBM to shared memory once per thread block (line 5 of Algorithm 1). In contrast, $\hat{\mathbf{K}}_j^T$ is partitioned column-wise among warps and is not reused within the thread block. Therefore, it is loaded directly from HBM into registers. Since $\hat{\mathbf{K}}$ is formed by gathering non-contiguous rows from K based on the column indices of nonzeros in $\mathbf{A}_i$, this leads to uncoalesced memory accesses. Furthermore, the PTX `mma` interface requires tensor core operands to follow specific alignment and layout constraints. This results in each thread issuing multiple 32-bit load instructions from scattered addresses as seen at the top-left of Figure 4.

To address this, we apply a *register remapping* optimization. As illustrated in the bottom-left of Figure 4, we optimize the memory access such that each thread issues a single 128-bit wide load. This is equivalent to permuting the columns of $\hat{\mathbf{K}}_j^T$. To preserve correctness of the output and to be compatible with SpMM, we apply the same permutation to the columns of $\mathbf{Q}_i$. This maintains the mathematical operation while optimizing memory access. Softmax does not incur additional data movement, as the result $\mathbf{S}_i$ of SDDMM is already resident in registers. After softmax, each warp writes its slice of $\mathbf{E}_i$ to shared memory, where it is reused by SpMM.

SpMM. In SpMM, $\mathbf{E}_i$ is already in shared memory. The matrix $\hat{\mathbf{V}}$ is gathered from rows of V using the same indices as for $\hat{\mathbf{K}}$, and likewise consists of non-contiguous memory accesses. Naively loading $\hat{\mathbf{V}}$ results in scattered memory instructions—for example, four separate 16-bit loads from non-adjacent addresses per thread, as shown on the top-right of Figure 4.

To mitigate this, we apply a similar register remapping to permute the column layout of $\hat{\mathbf{V}}$ to increase the horizontal granularity of each thread's load (see bottom-right of Figure 4). This results in a different layout of the output block, $\mathbf{O}_i$. Since $\mathbf{O}_i$ is stored in shared memory, we reverse this permutation when writing it back to HBM, so the final output layout matches the expected format.

We use the PTX `mma` interface rather than the CUDA `wmma` API. The key difference between the two is that `wmma` requires both input operands to reside in shared memory before being loaded into registers. In Fused3S, the right-hand-side operands $\hat{\mathbf{K}}_j^T$ in SDDMM and $\hat{\mathbf{V}}_j$ in SpMM are used only once per thread block and are not reused. Staging them in shared memory would introduce unnecessary data transfers and increase memory pressure without any performance benefit. With `mma`, we can load these operands directly from HBM into registers, reducing the number of memory operations and decreasing latency.

3.5 Mixed Precision and Stability

To optimize performance and memory footprint, Fused3S employs a mixed-precision strategy. Table 5 summarizes the precision of the inputs, intermediate results, and output. The input matrices Q, K, and V are stored in `fp16` to reduce memory bandwidth requirements and leverage high-throughput `fp16` tensor core operations. Intermediate attention scores S, computed during SDDMM, are accumulated in `fp32` to minimize precision loss. Softmax is computed entirely in `fp32` for numerical stability. After softmax, the normalized scores E are cast back to `fp16` before being stored in shared memory, as the subsequent SpMM accepts `fp16` inputs and produces the final output O in `fp32`. This mixed-precision design balances performance and accuracy.

Softmax. Softmax is a key operation in attention, but it is susceptible to numerical overflow in low-precision formats.

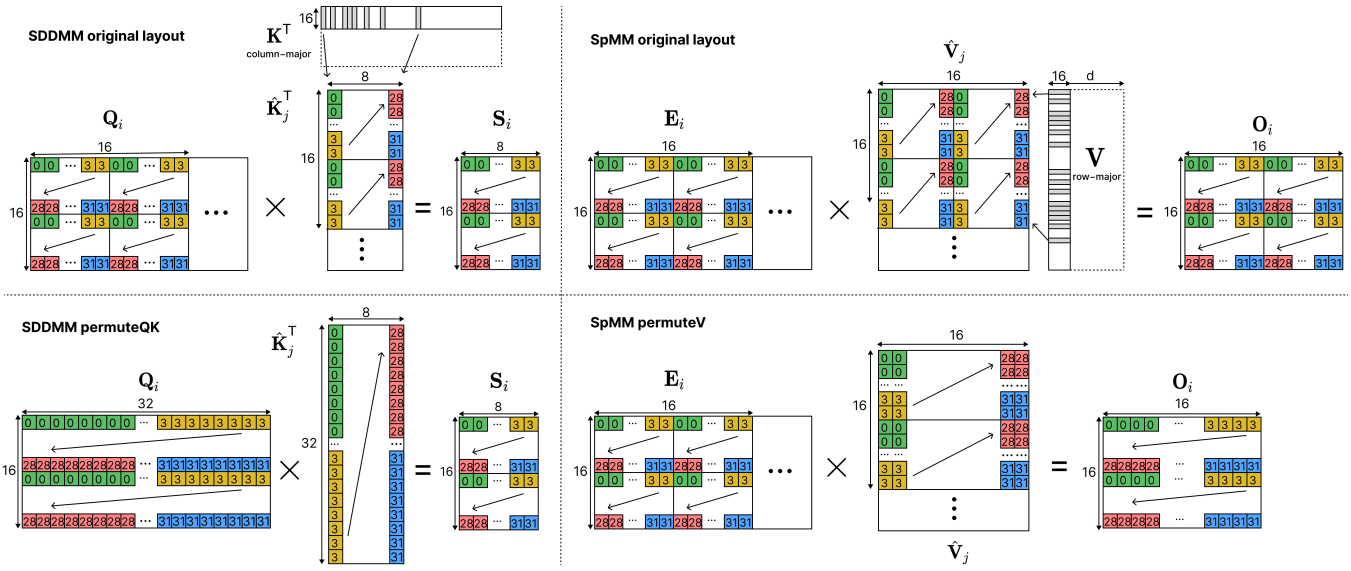


Figure 4: Register remapping in SDDMM (left) and SpMM (right). Top: original layouts. Bottom: permuted layouts.

Table 5: Precision of inputs, intermediate results and output.

Matrix	Q	K	V	S	E	O
Precision	fp16	fp16	fp16	fp32	fp32 → fp16	fp32

In its naive form,

$$\text{softmax}(\mathbf{x}) = \frac{\exp(\mathbf{x}_i)}{\sum_j \exp(\mathbf{x}_j)} \tag{6}$$

where the exponential may exceed the dynamic range of the floating point format. For example, the maximum value representable in `fp32` is approximately e^{89}. If any element in S exceeds 89, its exponential becomes infinity, resulting in `NaN` values in the output. In `fp16`, the threshold is even lower—around e^{11}—making the problem more severe. These overflows not only corrupt inference results but also break differentiability during backpropagation.

Most attention implementations use the *max-stabilized softmax* [11], defined as:

$$\text{softmax}(\mathbf{x}) = \frac{\exp(\mathbf{x}_i - \max(\mathbf{x}))}{\sum_j \exp(\mathbf{x}_j - \max(\mathbf{x}))} \tag{7}$$

which subtracts the row-wise maximum prior to exponentiation. Although the additional reduction (to compute max) introduces synchronization overhead in GPU kernels, the gain in numerical stability is typically well worth the cost.

We implement the *online softmax* algorithm [3], a blocked variant of the max-stabilized softmax. While online softmax can be less stable than the global variant, particularly with smaller block sizes [10], we find it to be a favorable trade-off for Fused3S. It significantly reduces memory consumption by avoiding the need to materialize the full attention score matrix and enables Fused3S to scale to large graphs, as demonstrated in Section 4.

4 Results

We evaluate the performance of Fused3S both as a standalone kernel and as the attention layers of a graph transformer model during inference on various real-world graphs of different sizes.

4.1 Setup

GPU Architecture. We perform experiments on NVIDIA A30 (Ampere) and H100 (Hopper) GPUs. The A30 has 56 SMs, each with 4 tensor cores and achieves up to 165 TFLOPs of FP16 tensor core throughput with 933 GB/s of DRAM bandwidth. The H100 has 132 SMs and 4 tensor cores per SM, delivering 990 TFLOPs of FP16 tensor core throughput and 4 TB/s of DRAM bandwidth. See NVIDIA datasheet for full architectural details [25, 26].

Datasets. We benchmark Fused3S on a diverse set of graph datasets drawn from popular GNN benchmarks [12, 14, 16, 19, 30, 39, 41, 43]. Table 6 summarizes their key properties. To characterize sparsity after the sparse matrix compaction described in Section 3.1, we report two metrics: `TCB/RW` and `nnz/TCB`, assuming a TCB size of 16×8. For both metrics, we include the coefficient of variation (CV = σ/μ), which

quantifies irregularity. A high CV (e.g., ≈ 1) in TCB/RW indicates a significant variation in workload per RW, which poses challenges for load balancing and warp-level parallelism.

Table 6: Single Graph Datasets. Metrics shown are after sparse compaction with TC block size 16×8.

Name	Nodes	Edges	TCB/RW		nnz/TCB	
			avg	CV	avg	CV
IGB-small	1M	12.1M	24.4	0.25	7.9	0.11
IGB-medium	10M	120M	24.4	0.58	7.9	0.11
Amazon0505	410K	3.36M	12.3	0.20	10.6	0.46
Com-Amazon	335K	926K	6	0.61	7.5	0.22
Musae-github	38K	578K	29.4	1.34	8.3	0.15
Artist	51K	819K	32	0.73	8	0.11
Pubmed	20K	89K	9.3	0.45	7.7	0.18
Cora	2.7K	10.6K	7.5	0.38	8.3	0.29
Citeseer	3.3K	9.2K	5.8	0.31	7.7	0.24
AmazonProducts	1.57M	264.3M	330.5	1.22	8.2	0.07
Yelp	717K	14M	39	1.28	8	0.09
Reddit	233K	114.9M	477.2	1.35	16.5	0.95
Blog	89K	4.19M	69	2.47	11	0.44
Elliptic	204K	234K	2.5	0.57	7.5	0.45
Ogbn-products	2.45M	123.7M	101.4	0.84	8	0.05

Table 7 presents a more detailed breakdown of work imbalance for four representative graphs. We sort all RWs by their TCB count and group them into ten deciles. Each cell reports the minimum and maximum TCB count within that decile. Graphs like Reddit exhibit a long tail: many sparse row windows, but some are extremely dense. Yelp and Github show similar irregularity, as reflected by their high CV values, making them useful for stress-testing load balance. In contrast, graphs like Pubmed have a more uniform distribution.

In addition to large single-graph datasets for node- and edge-level prediction tasks, many real-world applications–especially graph property prediction–process collections of small graphs (often fewer than 500 nodes). To improve GPU utilization, these graphs are batched together into a single larger graph. This batching introduces a unique sparsity pattern with many disconnected components. We evaluate Fused3S on batched graphs from two widely-used benchmarks: Long Range Graph Benchmark (LRGB) [6] and Open Graph Benchmark (OGB) [14], with a batch size of 1024.

Baselines. We compare Fused3S with the following competitive baselines for sparse attention and 3S:

- **DF-GNN** [21] is the state-of-the-art for the fused 3S kernel on CUDA cores in fp32, and includes a numerically stable softmax. We evaluate two variants: *tiling*, designed for larger graphs, and *hyper*, optimized for small graphs.
- **FlashSparse** [32] represents the state-of-the-art for SDDMM and SpMM as separate kernels on tensor cores with mixed precision (fp16 + fp32). The original code implements a naive softmax, so we also benchmark a modified version with a numerically stable softmax for a fair comparison.
- **PyTorch Geometric (PyG)** [8] is a widely used GNN framework with a PyTorch backend.
- **Deep Graph Library (DGL)** [37] is another popular GNN framework. We include it in the end-to-end transformer evaluation, as the original Graph Transformer [5] implementation is built on DGL.

4.2 3S Kernel Performance

Figure 5 shows the 3S kernel performance on the single graph datasets listed in Table 6. To summarize performance across the datasets, we report the geometric mean speedup computed as $\left(\prod_{d=1}^{D} s_d\right)^{\frac{1}{D}}$, where s_d is the speedup on dataset d and D is the total number of datasets.

On the A30 and H100 GPUs, Fused3S consistently outperforms all baselines achieving geometric mean speedups of $1.5 - 12.3\times$ and $1.6 - 14.7\times$ respectively. DF-GNN_hyper adopts a hybrid edge- and node-parallel strategy, partitioning non-zeros in SDDMM using edge-parallelism. This improves load balance and yields better performance than DF-GNN_tiling on small graphs. However, it consumes significantly more memory since it stores entire rows of S in shared memory. As a result, DF-GNN_hyper fails on high-degree graphs such as Reddit, AmazonProducts, Ogbn-products, and IGB-medium. In contrast, DF-GNN_tiling which is based on node-parallel fusion uses less shared memory and is preferred for large graphs but suffers from load imbalance. FlashSparse outperforms its stable-softmax variant due to the additional synchronization required to compute row-wise max. However, as discussed in Section 3.5, the naive softmax is prone to overflow errors and is not recommended in practice.

Fused kernels avoid storing intermediate results between SDDMM, softmax, and SpMM, reducing memory pressure. This is especially important on memory-constrained GPUs like the A30. For example, on the AmazonProducts dataset with the most number of edges, both FlashSparse and PyG fail due to out-of-memory (OOM) errors caused by materializing the large S matrix. In contrast, Fused3S and DF-GNN_tiling complete successfully. Fused3S further benefits from mixed precision (fp16/fp32) execution, using less memory than DF-GNN, which runs entirely in fp32. On H100, Fused3S remains the only kernel to run on the largest graphs tested (IGB-large and Ogbn-papers100M, results not shown).

Table 7: Distribution of TCB counts per RW across deciles. Each cell shows the min–max TCB range in that decile.

Dataset	decile size	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Reddit	1456	4–46	46–88	88–135	135–190	190–265	265–367	367–503	503–718	718–1113	1114–9857
Yelp	4480	4–9	9–12	12–15	15–19	19–23	23–29	29–38	38–52	52–82	82–1000
Pubmed	123	1–5	5–6	6–7	7–8	8–9	9–10	10–11	11–12	12–14	14–43
Github	236	2–13	13–16	16–18	18–20	20–23	23–25	25–29	29–34	34–46	46–1191

On highly irregular graphs such as Blog, Yelp and Github, Fused3S shows limited speedup. These datasets exhibit extreme variance in TCB counts per RW (see Table 7). For instance, in Github, 90% of row windows have fewer than 46 TC blocks, while a few exceed 1000. Even with row window reordering, such imbalance decreases compute and memory throughput. Assigning multiple thread blocks per row window could improve load balance. New GPU features such as *thread block clusters* allow thread blocks within a cluster to synchronize in shared memory, which we plan to explore in future work.

Figure 6 shows the performance on batched graphs. On the A30 and H100 GPUs, Fused3S consistently outperforms all baselines achieving geometric mean speedups of $1.5-14\times$ and $1.9-16.3\times$ respectively. Batched graphs consist of disconnected components that exhibit more regular sparsity and clustering, which can be exploited to improve memory locality. DF-GNN benefits from this naturally, whereas Fused3S currently does not exploit component boundaries or subgraph-level structure. Incorporating such structure-aware optimizations is a promising direction for future work.

4.3 Fused3S Performance Breakdown

We analyze the contribution of each kernel design decision in Fused3S by incrementally enabling optimizations. Each variant builds upon the previous one, and their performance is shown in Figures 5 and 6.

Warp partitioning. To evaluate the impact of warp partitioning strategies in Section 3.3, we compare two variants: `F3S_splitR`, the combination of split-row SDDMM and split-column SpMM, and `F3S_splitC`, the combination of split-column SDDMM and split-column SpMM. On single graph datasets, `F3S_splitC` achieves a geometric mean speedup of $1.5\times$ on both A30 and H100 GPUs. The benefit is less pronounced on batched graphs, which tend to have lower degrees and fewer TCBs per RW. As a result, the choice of warp partitioning has limited impact on overall performance.

Row window reordering. We analyze the impact of sorting RWs by their TCB count. This optimization improves load balance by scheduling expensive row windows earlier in the kernel execution. Figure 7 shows the active time of each of the 56 SMs on the A30 GPU, with and without reordering on two representative graphs (Reddit and Pubmed). Without reordering, some SMs remain active for longer than others. On average, `F3S_reorderRW` improves load balance and performance by $1.18\times$ over `F3S_splitC` on about half of the single graph datasets. However, the benefit depends on the graph structure. When only a few RWs contain many TCBs while the rest are sparse (e.g., Github and Blog), reordering offers limited gains (if any at all) as seen in Figures 5 and 6.

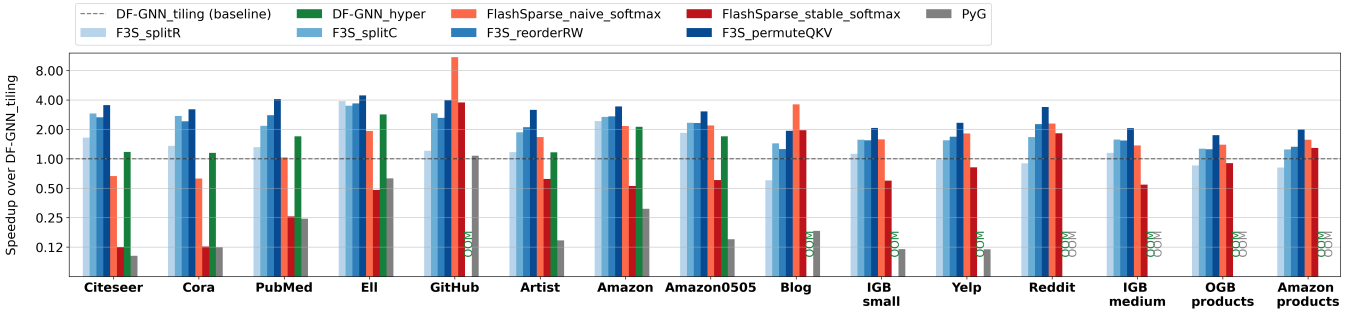
Permuting Q, K, and V. We examine the effect of permuting the layout of **Q**, **K**, and **V** as described in Section 3.4. The `F3S_permuteQKV` kernel applies this permutation on top of reordering and split-column partitioning. This optimization improves memory coalescing and instruction efficiency, achieving geometric mean speedups of $1.19-1.39\times$ on single graphs and $1.16-1.25\times$ on batched graphs.

4.4 End-to-end Model Performance

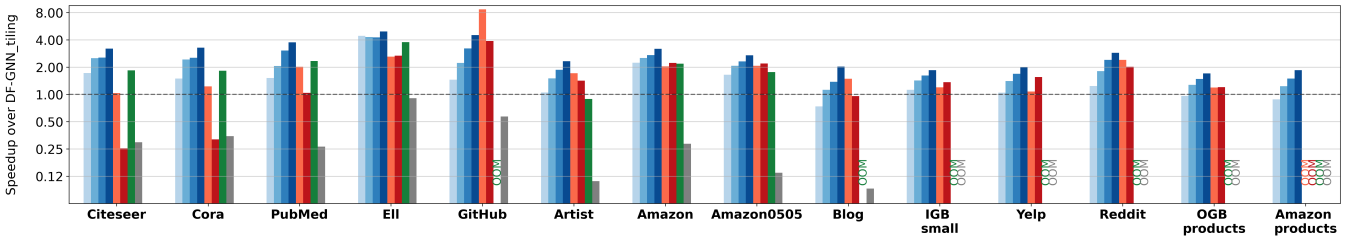
We evaluate the inference performance of the Graph Transformer (GT) model [5] which comprises 10 transformer blocks, each with an attention layer, three feedforward layers, and two normalization layers. We replace the original attention kernel implemented in DGL [37] with four 3S variants: Fused3S, DF-GNN's tiling and hyper kernels, and FlashSparse with naive softmax.

Figure 8 reports performance on five single graph and five batched graph datasets. For each dataset, we vary the embedding dimension $d \in \{64, 128, 256\}$ to assess sensitivity to model size. Fused3S improves end-to-end inference time, achieving geometric mean speedups of $1.1-3.08\times$ and $1.05-5.36\times$ over the baselines on A30 and H100 respectively. As shown in Figure 8(b) and (d), the DGL baseline spends the majority of its inference time in the attention kernel. Replacing DGL with any optimized 3S kernel (including DF-GNN and FlashSparse) significantly reduces this bottleneck. As a result, attention accounts for a smaller fraction of the total inference time, partially amortizing kernel-level speedups, especially on smaller graphs. The exceptions are larger graphs (Reddit, Ogbn-products, and AmazonProducts), where attention remains a bottleneck.

Interestingly, the effect of increasing d differs between A30 and H100. On A30, increasing d shifts the bottleneck toward the MLP layers, reducing the relative time spent in attention. In contrast, on H100, both MLP and attention layers scale efficiently, so attention remains a consistent

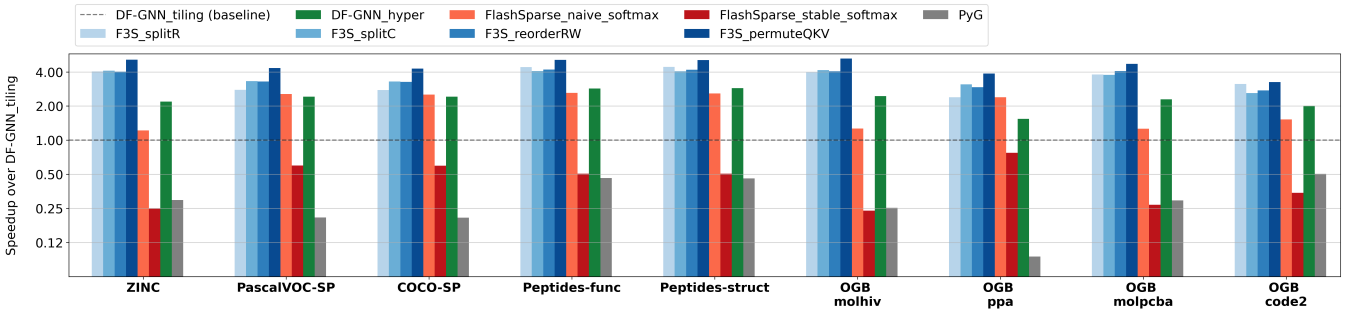


(a) H100 GPU. Fused3S achieves 2.8×, 2.2×, 1.6×, 4.4× and 14.7× geometric mean speedup over DF-GNN_tiling, DF-GNN_hyper, FlashSparse_naive_softmax, FlashSparse_stable_softmax, and PyG respectively.

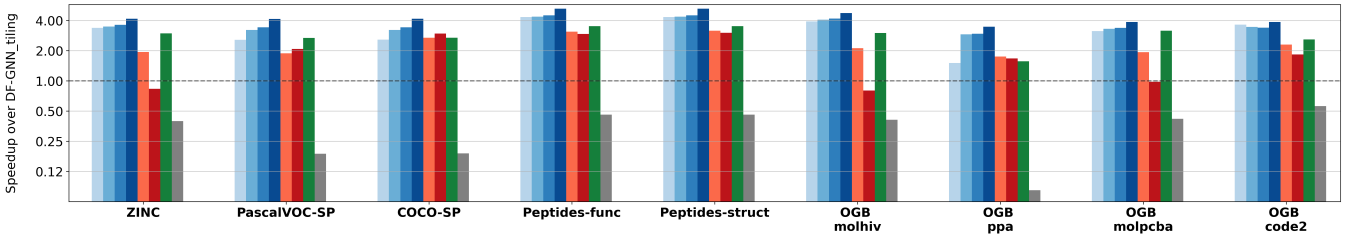


(b) A30 GPU. Fused3S achieves 2.7×, 1.7×, 1.5×, 2.2×, and 12.3× geometric mean speedup over DF-GNN_tiling , DF-GNN_hyper, FlashSparse_naive_softmax, FlashSparse_stable_softmax, and PyG respectively.

Figure 5: 3S kernel performance on single graph datasets in Table 6. Graphs are ordered by increasing number of edges (left to right). Y-axis is in log-scale.

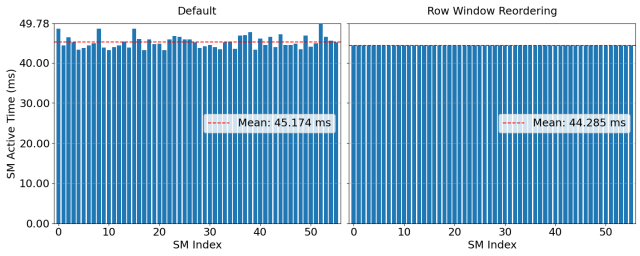


(a) H100 GPU. Fused3S achieves 4.5×, 1.9×, 2.4×, 10.8×, and 16.3× geometric mean speedup over DF-GNN_tiling , DF-GNN_hyper, FlashSparse_naive_softmax, FlashSparse_stable_softmax, and PyG respectively.

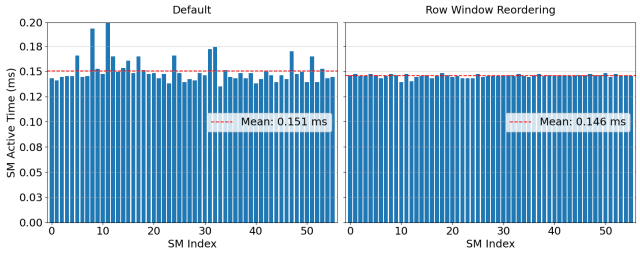


(b) A30 GPU. Fused3S achieves 4.3×, 1.5×, 1.9×, 2.5×, and 14× geometric mean speedup over DF-GNN_tiling , DF-GNN_hyper, FlashSparse_naive_softmax, FlashSparse_stable_softmax, and PyG respectively.

Figure 6: 3S kernel performance on batched graphs from LRGB [6] and OGB [14]. Y-axis is in log-scale.



(a) Reddit dataset



(b) Pubmed dataset

Figure 7: Comparison of SM active time on A30 with (right) and without (left) row window reordering.

or growing fraction of total time. This effect is particularly visible in DF-GNN_hyper and FlashSparse, where shared memory pressure or lack of fusion limits the scalability of attention at higher d.

5 Related Work

Sparse Matrix Computation on GPUs. Sparse matrix operations such as SpMM and SDDMM have received extensive attention due to their importance in GNNs, LLMs, and scientific computing. CUDA-core kernels such as Sputnik [9] and RoDe [28] use 1D/2D tiling, offset alignment, memory coalescing, and load-balancing heuristics to target unstructured sparsity. These approaches avoid preprocessing and operate directly on formats like CSR and COO.

With the growing adoption of tensor cores, recent efforts focus on enabling tensor core acceleration for sparse operations. TC-GNN [38] proposes a TC-friendly format (TCF) that aligns sparsity patterns with MMA operand constraints; DTC-SpMM [7] extends this with ME-TCF and sparse double buffering to further reduce memory latency. Both DTC-SpMM and SMaT[27] use row reordering to increase the density of MMA tiles. FlashSparse [32] introduces separate optimized kernels for SDDMM and SpMM using the memory-efficient BCRS format and forming denser MMA tiles using 8×1 vectors. Acc-SpMM [46] proposes BitTCF, a compressed bitmask format for efficient tile decoding.

Other designs focus on structured or semi-structured sparsity. JigSaw [44], Flash-LLM [40] and BSA-SpMM [17] focus on SpMM in transformer inference, where inputs are tall-skinny and sparsity is generated by weight pruning. TCA-SpMM [13] reshapes vector dot products into blocked matrix multiplications to improve TC utilization without preprocessing the sparse matrix into a different format. These techniques perform well under certain assumptions, but might not generalize to the irregular sparsity found in real-world graph data.

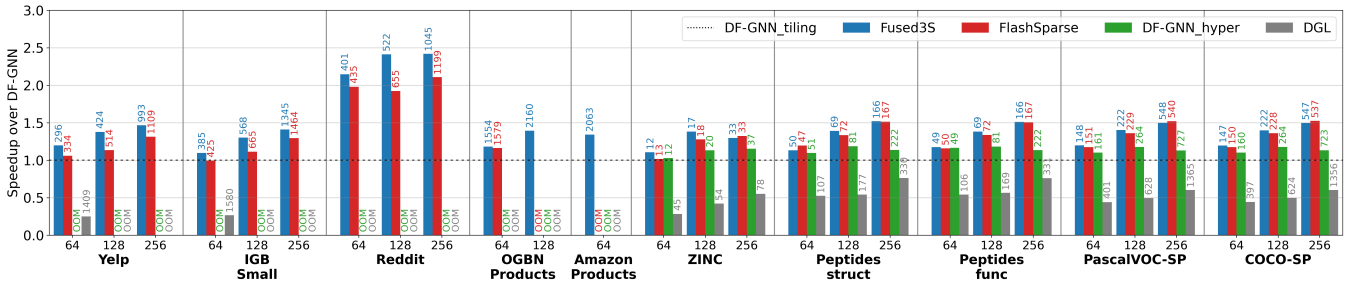
Sparse Attention and Fused Kernels. Sparse attention typically involves three operations: SDDMM, softmax, and SpMM. Popular frameworks like DGL [37] and PyG [8] implement these as separate kernels and materializing intermediate outputs in memory. This results in significant memory traffic and kernel launch overhead. DF-GNN [21] is the first work to fuse all three operations into a single CUDA-core kernel with a numerically stable softmax. It proposes two variants: tiling for large graphs and hyper for small graphs with high variance in node degree. However, DF-GNN executes entirely in `fp32`, uses CSR/COO/CSC formats, and does not target tensor cores.

Sputnik, FlashSparse, and Magicube [20] also target sparse attention but do not fuse the softmax stage—intermediate results are materialized between SDDMM and SpMM. As a result, memory pressure remains high, and performance is bounded by inter-kernel synchronization.

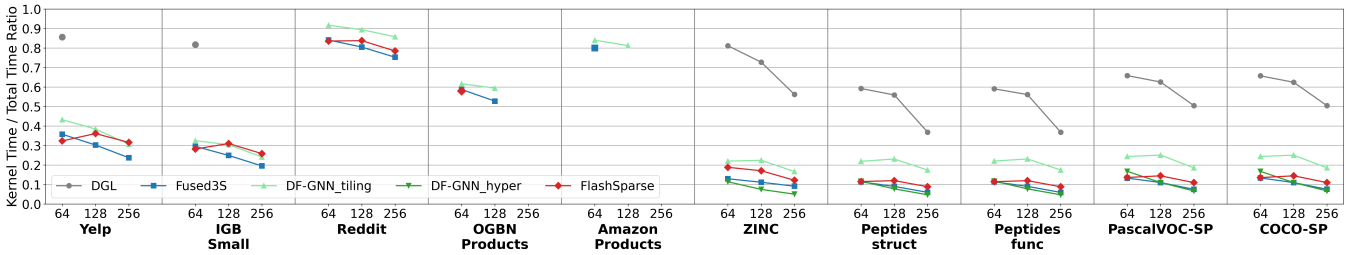
6 Conclusion

Fused3S is the first fully on-chip, fused sparse attention kernel designed for tensor cores. It introduces the BSB format, a block-aligned layout optimized for MMA operand shapes, and fuses SDDMM, softmax, and SpMM into a single TC-accelerated mixed-precision kernel. Fused3S integrates GPU optimizations including warp-level split-column partitioning, register remapping, and row window reordering to improve memory coalescing and address load imbalance under high sparsity. Experimental results show that Fused3S achieves high performance on real-world graphs with unstructured sparsity–a use case not well supported by prior tensor core or fused sparse kernels.

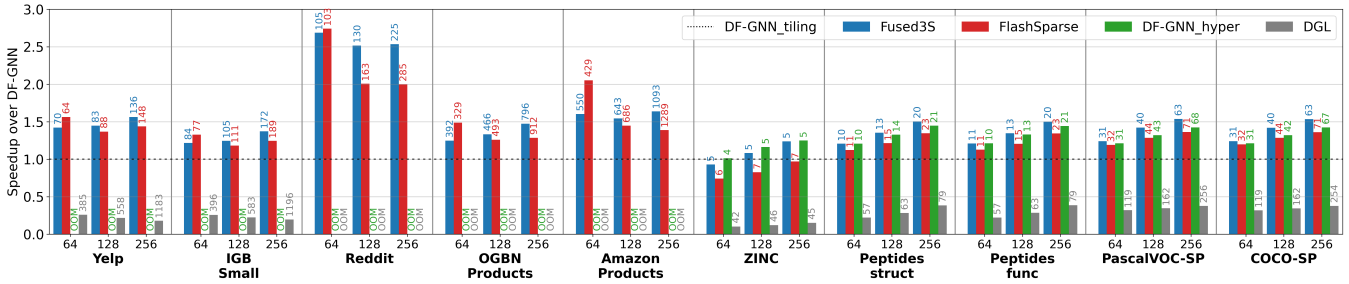
Looking ahead, several directions offer potential for improving Fused3S. Hopper's hardware features such as `fp8` compute and Tensor Memory Accelerator (TMA) could further reduce memory overhead and improve throughput. Alternative tile shapes enabled by lower precision, and operand reordering such as FlashSparse's swap-and-transpose technique, may increase computational density. While this work focuses on the forward pass, extending the optimizations to the backward pass—which also involves SpMM and SDDMM operations in reverse order—is expected to yield similar performance improvements for training. Additionally, support for thread block clusters could enable synchronization across



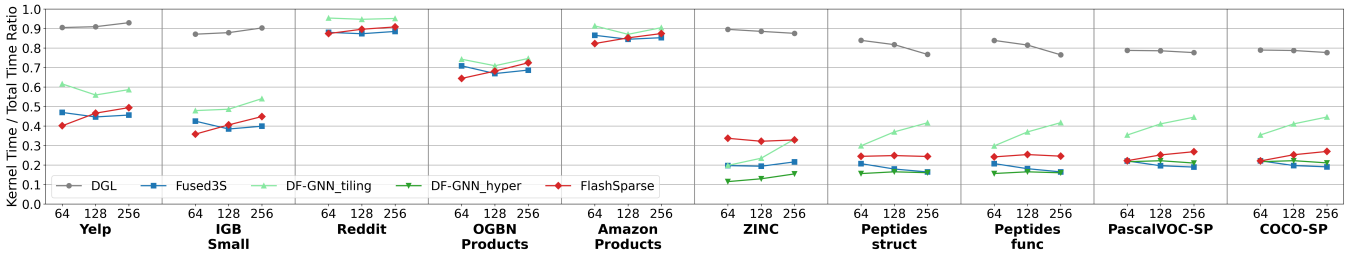
(a) Performance on A30. Fused3S achieves 1.55×, 1.29×, 1.10×, and 3.08× speedup over DF-GNN_tiling, DF-GNN_hyper, FlashSparse, and DGL respectively.



(b) Fraction of inference time spent in attention kernels on A30.



(c) Performance on H100. Fused3S achieves 1.56×, 1.05×, 1.15× and 5.36× speedup over DF-GNN-tiling, DF-GNN-hyper, FlashSparse, and DGL respectively.



(d) Fraction of inference time spent in attention kernels on H100.

Figure 8: Graph Transformer inference performance with different 3S kernels. Missing bars indicate OOM. Labels on top of bars record the runtime in milliseconds.

multiple thread blocks, unlocking finer-grained load balancing. Finally, adapting Fused3S to better support the per-graph sparsity and block-disconnected structure of batched GNN datasets (e.g., molecular graphs, abstract syntax trees, crystal graphs) may help bridge the gap between general sparse attention and multi-graph applications.

Acknowledgments

We thank the Research Cyberinfrastructure Center at UC Irvine for access to the GPUs on the HPC3 cluster. We also thank Alex Danielian and Daniel Hsu for their assistance with kernel development and data preparation.

References

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

[2] 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

[3] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=mZn2Xyh9Ec

[4] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG] https://arxiv.org/abs/2205.14135

[5] Vijay Prakash Dwivedi and Xavier Bresson. 2021. A Generalization of Transformer Networks to Graphs. *AAAI Workshop on Deep Learning on Graphs: Methods and Applications* (2021).

[6] Vijay Prakash Dwivedi, Ladislav Rampášek, Mikhail Galkin, Ali Parviz, Guy Wolf, Anh Tuan Luu, and Dominique Beaini. 2022. Long Range Graph Benchmark. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=in7XC5RcjEn

[7] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2024. DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 253–267. doi:10.1145/3620666.3651378

[8] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[9] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 17, 14 pages.

[10] Alicia Golden, Samuel Hsia, Fei Sun, Bilge Acun, Basil Hosmer, Yejin Lee, Zachary DeVito, Jeff Johnson, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. 2024. Is Flash Attention Stable? arXiv:2405.02803 [cs.LG] https://arxiv.org/abs/2405.02803

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[12] William L. Hamilton, Rex Ying, and Jure Leskovec. 2018. Inductive Representation Learning on Large Graphs. arXiv:1706.02216 [cs.SI] https://arxiv.org/abs/1706.02216

[13] Yoonsang Han, Inseo Kim, Jinsung Kim, and Gordon Euhyun Moon. 2024. Tensor Core-Adapted Sparse Matrix Multiplication for Accelerating Sparse Deep Neural Networks. *Electronics* 13, 20 (Jan. 2024), 3981. doi:10.3390/electronics13203981 Number: 20 Publisher: Multidisciplinary Digital Publishing Institute.

[14] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[15] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization framework for sparse matrix kernels. *The International Journal of High Performance Computing Applications* 18, 1 (2004), 135–158.

[16] Arpandeep Khatua, Vikram Sharma Mailthody, Bhagyashree Taleka, Tengfei Ma, Xiang Song, and Wen-mei Hwu. 2023. IGB: Addressing The Gaps In Labeling, Features, Heterogeneity, and Size of Public Graph Datasets for Deep Learning Research. In *In Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23) (KDD '23)*. doi:10.48550/ARXIV.2302.13522

[17] Eunji Lee, Yoonsang Han, and Gordon Euhyun Moon. 2024. Accelerated Block-Sparsity-Aware Matrix Reordering for Leveraging Tensor Cores in Sparse Matrix-Multivector Multiplication. In *Euro-Par 2024: Parallel Processing*, Jesus Carretero, Sameer Shende, Javier Garcia-Blas, Ivona Brandic, Katzalin Olcoz, and Martin Schreiber (Eds.). Springer Nature Switzerland, Cham, 3–16.

[18] Heejun Lee, Jina Kim, Jeffrey Willette, and Sung Ju Hwang. 2024. SEA: Sparse Linear Attention with Estimated Attention Mask. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=JbcwfmYrob

[19] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[20] Shigang Li, Kazuki Osawa, and Torsten Hoefler. 2022. Efficient quantized sparse matrix operations on tensor cores. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 37, 15 pages.

[21] Jiahui Liu, Zhenkun Cai, Zhiyong Chen, and Minjie Wang. 2024. DF-GNN: Dynamic Fusion Framework for Attention Graph Neural Networks on GPUs. In *The Third Learning on Graphs Conference*. https://openreview.net/forum?id=8GNDnBbUfF

[22] Liu Liu, Zheng Qu, Zhaodong Chen, Fengbin Tu, Yufei Ding, and Yuan Xie. 2022. Dynamic Sparse Attention for Scalable Transformer Acceleration. *IEEE Trans. Comput.* 71, 12 (2022), 3165–3178. doi:10.1109/TC.2022.3208206

[23] Luis Müller, Mikhail Galkin, Christopher Morris, and Ladislav Rampášek. 2024. Attending to Graph Transformers. *Transactions on Machine Learning Research* (2024). https://openreview.net/forum?id=HhbqHBBrfZ

[24] NVIDIA Corporation 2018. *NVIDIA TESLA V100*. NVIDIA Corporation. https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf

[25] NVIDIA Corporation 2022. *NVIDIA A30 TENSOR CORE GPU*. NVIDIA Corporation. https://www.nvidia.com/content/dam/en-zz/Solutions/data-center/products/a30-gpu/pdf/a30-datasheet.pdf

[26] NVIDIA Corporation 2025. *NVIDIA GH200 Grace Hopper Superchip*. NVIDIA Corporation. https://resources.nvidia.com/en-us-grace-cpu/grace-hopper-superchip?ncid=no-ncid

[27] Patrik Okanovic, Grzegorz Kwasniewski, Paolo Sylos Labini, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2024. High Performance Unstructured SpMM Computation Using Tensor Cores. arXiv:2408.11551 [cs.DC]

[28] Meng Pang, Xiang Fei, Peng Qu, Youhui Zhang, and Zhaolin Li. 2024. A Row Decomposition-based Approach for Sparse Matrix Multiplication on GPUs *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 377–389. doi:10.1145/3627535.3638470

[29] Md. Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. 2021. FusedMM: A Unified SDDMM-SpMM Kernel for Graph Embedding and Graph Neural Networks . In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 256–266. doi:10.1109/IPDPS49936.2021.00034

[30] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[31] Ahsan Shehzad, Feng Xia, Shagufta Abid, Ciyuan Peng, Shuo Yu, Dongyu Zhang, and Karin Verspoor. 2024. Graph Transformers: A Survey. arXiv:2407.09777 [cs.LG] https://arxiv.org/abs/2407.09777

[32] Jinliang Shi, Shigang Li, Youxuan Xu, Rongtian Fu, Xueying Wang, and Tong Wu. 2024. FlashSparse: Minimizing Computation Redundancy for Fast Sparse Matrix Multiplications on Tensor Cores. arXiv:2412.11007 [cs.DC] https://arxiv.org/abs/2412.11007

[33] Hamed Shirzad, Ameya Velingker, Balaji Venkatachalam, Danica J Sutherland, and Ali Kemal Sinop. 2023. Exphormer: Sparse transformers for graphs. In *International Conference on Machine Learning*. arXiv:2303.06147

[34] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. 2018. Attention-based graph neural network for semi-supervised learning. *arXiv preprint arXiv:1803.03735* (2018).

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[36] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[37] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[38] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. 2023. TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 149–164. https://www.usenix.org/conference/atc23/presentation/wang-yuke

[39] Mark Weber, Giacomo Domeniconi, Jie Chen, Daniel Karl I. Weidele, Claudio Bellei, Tom Robinson, and Charles E. Leiserson. 2019. Anti-Money Laundering in Bitcoin: Experimenting with Graph Convolutional Networks for Financial Forensics. arXiv:1908.02591 [cs.SI] https://arxiv.org/abs/1908.02591

[40] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. 2023. Flash-LLM: Enabling Cost-Effective and Highly-Efficient Large Generative Model Inference with Unstructured Sparsity. doi:10.48550/arXiv.2309.10285 arXiv:2309.10285 [cs].

[41] Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. 2016. Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (New York, NY, USA) *(ICML'16)*. JMLR.org, 40–48.

[42] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big bird: transformers for longer sequences. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1450, 15 pages.

[43] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. arXiv:1907.04931 [cs.LG] https://arxiv.org/abs/1907.04931

[44] Kaige Zhang, Xiaoyan Liu, Hailong Yang, Tianyu Feng, Xinyu Yang, Yi Liu, Zhongzhi Luan, and Depei Qian. 2024. Jigsaw: Accelerating SpMM with Vector Sparsity on Sparse Tensor Core. In *Proceedings of the 53rd International Conference on Parallel Processing (ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 1124–1134. doi:10.1145/3673038.3673108

[45] Meng Zhang, Jie Sun, Qinghao Hu, Peng Sun, Zeke Wang, Yonggang Wen, and Tianwei Zhang. 2024. TorchGT: A Holistic System for Large-Scale Graph Transformer Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 77, 17 pages. doi:10.1109/SC41406.2024.00083

[46] Haisha Zhao, San Li, Jiaheng Wang, Chunbao Zhou, Jue Wang, Zhikuang Xin, Shunde Li, Zhiqiang Liang, Zhijie Pan, Fang Liu, Yan Zeng, Yangang Wang, and Xuebin Chi. 2024. Acc-SpMM: Accelerating General-purpose Sparse Matrix-Matrix Multiplication with GPU Tensor Cores. arXiv:2501.09251 [cs.DC] https://arxiv.org/abs/2501.09251