

TileSpMM: A Variable-Size Tiled Algorithm for Sparse Matrix-Matrix Multiplication on Tensor Cores

Hongwei Zeng
China University of
Petroleum-Beijing
Beijing, China

hongwei@student.cup.edu.cn

Shuqin Feng
China University of
Petroleum-Beijing
Beijing, China

fengshuqin@student.cup.edu.cn

Haocheng Lian
China University of
Petroleum-Beijing
Beijing, China

haocheng.lian@student.cup.edu.cn

Yuechen Lu
China University of
Petroleum-Beijing
Beijing, China
yuechen.lu@cup.edu.cn

Weifeng Liu
China University of
Petroleum-Beijing
Beijing, China
weifeng.liu@cup.edu.cn

Abstract

Sparse Matrix-Matrix Multiplication (SpMM) is a fundamental primitive in scientific computing and artificial intelligence applications. Modern hardware, notably Tensor Core Units (TCUs), offers immense computational power, creating promising opportunities for SpMM acceleration. However, it is difficult to map sparse matrices with irregular structures onto TCUs due to their requirement for regular operands. Existing fixed-granularity tiling methods frequently face a trade-off between massive zero-padding in sparse regions and poor spatial data locality in dense regions. To bridge this gap, we propose TileSpMM, which breaks the static-granularity bottleneck through a variable-size tiling algorithm that dynamically adapts to local sparsity patterns. Furthermore, TileSpMM is equipped with an adaptive load-balancing strategy and customized granularity-specific kernels to improve hardware utilization and mitigate computation redundancy. Experiments on NVIDIA H100 and RTX 5090 GPUs with a diverse range of benchmark matrices show that TileSpMM delivers overall better performance than existing SpMM methods across the evaluated platforms and datasets. Compared with cuSPARSE, SSpMM, Acc-SpMM and FlashSparse, TileSpMM achieves geometric mean speedups of 4.76 $\times$, 2.74 $\times$, 2.38 $\times$ and 1.58 $\times$, respectively.

CCS Concepts

• **Computing methodologies** $\rightarrow$ **Parallel algorithms**; • **Computer systems organization** $\rightarrow$ **Parallel architectures**; • **Mathematics of computing** $\rightarrow$ *Mathematical software performance*.

Keywords

Sparse Matrix-Matrix Multiplication, GPU, Tensor Core, Matrix partitioning

ACM Reference Format:

Hongwei Zeng, Shuqin Feng, Haocheng Lian, Yuechen Lu, and Weifeng Liu. 2026. TileSpMM: A Variable-Size Tiled Algorithm for Sparse Matrix-Matrix Multiplication on Tensor Cores. In *Proceedings of the 55th International Conference on Parallel Processing (ICPP '26)*, September 28–October 01, 2026, Singapore, Singapore. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3832810.3832918>

1 Introduction

Sparse Matrix-Matrix Multiplication (SpMM) is a fundamental computational kernel in numerous modern workloads, including Graph Neural Networks (GNNs) [1, 4, 11, 16, 21, 22, 41, 42, 46], scientific computing [2, 18, 23, 28, 31–33, 49] and large-scale deep learning models (e.g., pruned LLMs) [3, 14, 48]. As these applications grow in scale and complexity, SpMM execution efficiency significantly impacts the overall end-to-end performance. To accelerate SpMM, previous research has focused on general-purpose GPU CUDA cores (e.g., Sputnik [10] and RoDe [39]). However, with the emergence of specialized hardware accelerators, Tensor Core Units (TCUs) [5, 8, 25, 26, 29, 30, 44, 47] now provide substantially higher throughput than CUDA cores. Consequently, migrating SpMM workloads to TCUs for optimization has become a mainstream trend.

Existing TCU-based SpMM approaches typically rely on static, single-granularity tiling. However, real-world matrices often exhibit highly polarized nonzero distributions, leading to extreme variations in spatial data locality across dense and sparse regions (Figure 2). This inherent polarization forces single-granularity methods (e.g., TC-GNN [47], DTC-SpMM [8], FlashSparse [44]) into a fundamental dilemma: fine-grained tiling accommodates sparse regions but destroys spatial locality in dense regions, causing memory overhead and pipeline stalls; conversely, coarse-grained tiling preserves dense locality but introduces severe zero-padding in sparse regions, wasting both High Bandwidth Memory (HBM) and TCU throughput.

While multi-granularity partitioning is a natural direction to resolve this dilemma, realizing its potential requires addressing two challenges. First, window sizes cannot be chosen arbitrarily. Different granularities yield different TCU execution efficiencies,



This work is licensed under a Creative Commons Attribution 4.0 International License. *ICPP '26, Singapore, Singapore*

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2657-6/26/09
<https://doi.org/10.1145/3832810.3832918>

and overly large or inefficient granularities may introduce additional resource pressure and zero-padding overhead. Therefore, the partitioning strategy must restrict candidate window sizes to hardware-efficient granularities suitable for TCU execution. Second, multi-granularity partitioning divides the uniform computation flow into heterogeneous sub-tasks with different nonzero counts, compute densities, and resource demands. Conventional static load balancing falls short in this scenario, as fixed task assignment inevitably triggers severe tail latency and restricts cross-platform adaptability.

In this paper, we propose TileSpMM, a structure-aware variable-size tiled SpMM algorithm optimized for TCUs. At the algorithm level, TileSpMM evaluates local column overlap rates to dynamically partition the sparse matrix into three distinct sub-matrices: 64-row tiles are dedicated to extracting dense regions to maximize data reuse, 8-row tiles process ultra-sparse regions to minimize invalid computation, and 16-row tiles serve as a smooth transition. While previous works typically rely on conventional static partitioning that suffers from severe load imbalance and limited platform adaptability across severely irregular matrices, TileSpMM effectively resolves this limitation. At the execution level, we deploy customized TCU kernels for different granularities, including overlapping pipelines for dense regions and an operand-swapping strategy for ultra-sparse regions. At the scheduling level, we introduce a dynamic load-balancing strategy, combining static modeling with dynamic micro-benchmarking, to identify effective thread-block assignment thresholds for these diverse sub-matrices.

We conduct evaluations on the RTX 5090 and H100 GPUs using the Deep Learning Matrix Collection (DLMC) [13] and the SuiteSparse Collection [7] datasets, alongside representative graph datasets for end-to-end workloads. Experimental results demonstrate that TileSpMM sets a new performance benchmark for unstructured sparse computing. Specifically, compared to existing works such as SMaT [38], cuSPARSE [36], SSpMM [50], Acc-SpMM [52], FlashSparse [44] and RoDe [39], TileSpMM delivers geometric mean speedups of 12.84×, 4.76×, 2.74×, 2.38×, 1.58×, and 1.29×, respectively.

In summary, the main contributions of this paper are as follows:

- We propose a variable-size tiled algorithm to balance zero-padding reduction and spatial data reuse;
- We design a load-balancing scheduling strategy and customized granularity-specific kernels, enabling efficient execution of heterogeneous tiles on TCUs;
- We demonstrate significant speedups over existing SpMM methods in both single-kernel and end-to-end evaluations.

2 Background and Motivation

2.1 Tensor Cores

To address the massive throughput demands of deep learning and high-performance computing, NVIDIA introduced TCUs. These specialized units accelerate Matrix-Multiply-Accumulate (MMA) operations ($D = AB + C$) at the hardware level. They maximize arithmetic throughput and enhance streaming multiprocessor (SM) data reuse. To balance this speed with memory bandwidth limits, TCUs rely on mixed-precision arithmetic (e.g., multiplying in FP16/TF32 and accumulating in FP32).

We leverage the fine-grained `mma.m16n8k16` and `mma.m16n8k8` intrinsic instructions for FP16 and TF32 inputs instead of the WMMA API to improve the locality of sparse computation. Binding these directly to 32-bit registers eliminates memory overhead and grants absolute control over thread-level data routing.

2.2 SpMM on Tensor Cores

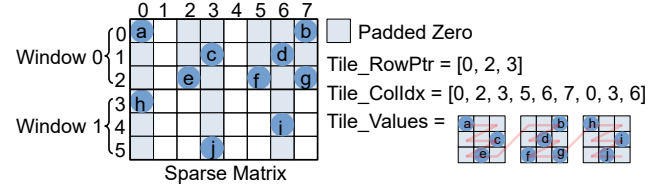


Figure 1: Window-based sparse matrix format, where the matrix is divided into row windows and organized into fixed-size dense tiles.

SpMM computes $C = AB$, where $A \in \mathbb{R}^{M \times K}$ is a sparse matrix, $B \in \mathbb{R}^{K \times N}$ and $C \in \mathbb{R}^{M \times N}$ are dense matrices. In order to use TCUs, SpMM methods usually reorganize the sparse matrix into small dense blocks. A common strategy is to partition the sparse matrix into fixed-height row windows and pack the nonzeros within each window into dense tiles that can be used by TCU instructions.

As illustrated in Figure 1, the sparse matrix is horizontally partitioned into fixed-height row windows (e.g., 3 rows) to align with hardware instruction dimensions. Within each window, compaction is performed: columns containing nonzeros are aggregated into dense blocks. For example, in Window 0, the non-empty columns 0, 2, 3 form the first 3×3 block, and columns 5, 6, 7 form the second.

The resulting window-based format is represented by arrays, including `Tile_RowPtr`, `Tile_ColIdx`, and `Tile_Values`, which record the tile offsets, original column indices, and packed dense values, respectively. During computation, each packed tile of A is multiplied with the corresponding rows of B using MMA instructions, and the results are accumulated into the rows of C .

2.3 Motivation

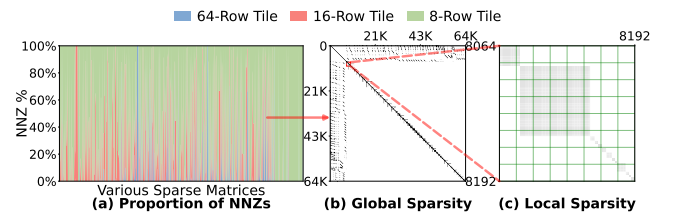


Figure 2: Motivation for Multi-granularity SpMM. (a) The number of nonzeros (NNZs) processed by different tile granularities across various datasets. (b) Global nonzero distribution of a representative sparse matrix (mip1). (c) A 128×128 zoomed-in view of the region highlighted in (b).

As shown in Figures 2(b) and 2(c), real sparse matrices can contain both dense clusters and much sparser regions. Fixed-size blocking methods struggle to handle matrices where nonzeros are non-uniformly distributed: (1) Using a smaller row window reduces zero-padding in sparse regions, but it severely disrupts data locality and limits data reuse in dense regions, while additionally incurring higher addressing and memory access overheads. (2) Using a large row window can improve reuse in dense regions, but it may introduce many padded zeros in sparse regions. **This motivates a variable-size row-window design that can match different sparsity patterns within the matrix.**

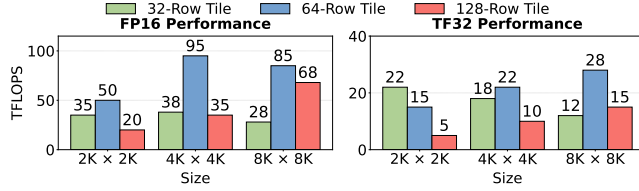


Figure 3: GEMM Performance with different tile sizes.

Given the limitation of fixed-size row windows, a key design question is how to determine appropriate window sizes for TCU execution. To examine the impact of window size on TCU efficiency, Figure 3 reports the performance of different tile granularities using GEMM kernels from the CUDA samples [37]. The results show that larger tiles do not always provide higher throughput. For both FP16 and TF32, the best-performing granularity varies with the computation dimensions. For sparse matrices, large tiles are inherently harder to aggregate. Forcing scattered nonzeros into large tiles triggers massive zero-padding and severe resource pressure, degrading overall performance. **This motivates restricting variable-size row windows to a few hardware-efficient granularities rather than allowing arbitrary window sizes.**

However, using multiple row-window sizes introduces a new load-balancing challenge. Figure 2(a) shows that the number of nonzeros assigned to different row-window granularities exhibits substantial variance across matrices. Since different granularities consume different amounts of computation and hardware resources, directly assigning these tasks to thread blocks can lead to uneven execution time: blocks processing dense or large windows may become the critical path, while blocks processing sparse or small windows finish early. **This motivates a scheduling strategy that balances heterogeneous workloads produced by variable-size row-window partitioning.**

3 TileSpMM

3.1 Overview

To resolve the inherent dilemma of traditional static windowing, we propose TileSpMM, a systematic algorithm that coordinates SpMM execution on TCUs through three sequential phases: Variable-Size Tile Partitioning, Load Balancing Strategy, and Granularity-Specific Kernel Optimization. Conventionally, sparse matrices are formatted using fixed-height row windows. In our algorithm, we define a tile as a variable-size row window. First, during the tile partitioning

phase, TileSpMM scans the input matrix and partitions its rows into three distinct sub-matrices based on localized sparsity patterns, utilizing 64-, 16-, and 8-row tiles. Second, within each sub-matrix, the extracted nonzeros from these tiles are compacted into dense fragments sized specifically for Tensor Core execution. To mitigate severe resource underutilization caused by varying tile sizes, a load-balancing strategy is deployed to systematically determine the best-performing fragment-to-thread-block assignment ratio. Finally, these dense fragments are processed by customized kernels, where we implement overlapped memory pipelines for 64/16-row tiles and an operand swapping strategy for 8-row tiles to maximize hardware utilization.

3.2 Multi-granularity Matrix Partitioning

In real-world sparse matrices, the distribution of nonzeros exhibits severe irregularity. Employing a fixed-height tiling strategy typically imposes a trade-off: in dense regions, fine-grained partitioning not only disrupts inter-row data reuse but also amplifies memory access overhead for sparse indices; while in sparse regions, coarse-grained partitioning incurs substantial zero-padding computations.

To resolve this dilemma, TileSpMM introduces a variable-size tile partitioning algorithm. Instead of physically reordering the matrix data in memory, it utilizes a sliding row pointer to logically group rows into three specific partition sizes: 64-row, 16-row, and 8-row tiles, guided by local sparsity patterns.

3.2.1 Evaluation Criteria for Partition Sizes. To balance data reuse in dense regions against zero-padding overheads in extremely sparse areas, we quantify this architectural trade-off using a non-empty column overlap ratio. Figure 4 illustrates this with a down-sized model, where 8-, 4-, and 2-row tiles represent the practical 64-, 16-, and 8-row granularities. For a given evaluation tile, let N be the total distinct non-empty columns, and S be the sum of the number of non-empty columns across its uniformly divided sub-tiles. The generic overlap ratio is S/N . For example, the initial 8-row tile in Figure 4(a) has $N_8 = 9$ and $S_4 = 12$, yielding a ratio of S_4/N_8 . A higher ratio signifies dense column sharing, justifying a larger partition to maximize data reuse. Conversely, a low ratio implies negligible overlap, where forced consolidation triggers massive invalid zero-padding.

We apply this metric to our multi-level partitioning design, where a 64-row evaluation tile uses S_{16}/N_{64} and a 16-row tile uses S_8/N_{16} . As illustrated in Figure 4(a), the initial 8-row tile starting at $i = 0$ contains $N_8 = 9$ distinct columns, while its two 4-row sub-tiles contain 7 and 5 columns, respectively, resulting in $S_4 = 12$ and an overlap ratio of S_4/N_8 . To capture the relationship between sparsity patterns and hardware efficiency, the decision thresholds γ and δ are determined using an XGBoost regression model trained with matrix structural features and empirical performance data. During host preprocessing, the model predicts the performance of different threshold combinations based on seven matrix features, including dimensions, nonzero statistics, and sparsity characteristics. The pre-trained model is deployed as lightweight C code to perform a grid search over candidate thresholds and select the pair with the highest predicted performance.

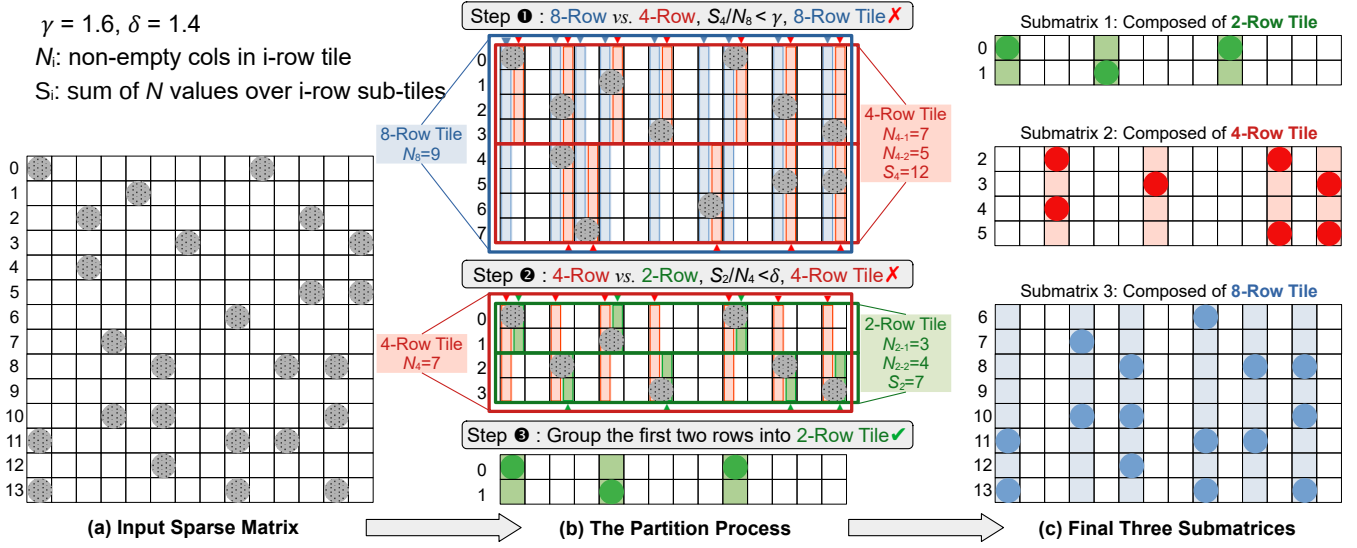


Figure 4: An example of the variable-size tile partitioning process. The practical 64/16/8-row granularities are represented as 8/4/2-row tiles. For the input matrix (a), subfigure (b) details the evaluation and fallback process of the first 8 rows. The algorithm evaluates progressively smaller tile sizes using overlap thresholds γ and δ to balance data reuse and zero-padding. Since both 8- and 4-row tiles fail to satisfy the conditions, rows 0–1 form a 2-row tile. Applying this process to the remaining rows yields the final partitioned state in (c).

3.2.2 Partitioning Flow. Driven by this metric, TileSpMM executes a single-pass row traversal to determine an appropriate partition granularity. Indexed by a row pointer i , the algorithm applies a sequential decision process: it first probes a 64-row tile. If the overlap ratio $S_{16}/N_{64} > \gamma$, it extracts this region as a reuse-intensive partition and advances i by 64. Should this fail, the algorithm downgrades to evaluate a 16-row tile, extracting it and advancing i by 16 if $S_8/N_{16} \geq \delta$. If both threshold checks fail, indicating a low-density region, the algorithm falls back to the minimum 8-row granularity to minimize zero-padding, advancing i by 8.

Figure 4(b) details this evaluation process visually. Starting at $i = 0$, the algorithm initially evaluates the 8-row tile but rejects it because $S_4/N_8 < \gamma$. It then downgrades to the 4-row tile, obtaining $N_4 = 7$ and $S_2 = 7$, which is also rejected since $S_2/N_4 < \delta$. Consequently, the fallback strategy is triggered, extracting just 2 rows to form the first partition and advancing the pointer to $i = 2$. Sequentially applying this process yields the final heterogeneous partitioned layout shown in Figure 4(c).

3.2.3 Sub-matrix Reassembly without Reordering. Variable-size tile partitioning disrupts the original matrix row sequence. Since post-computation global memory reordering incurs significant bandwidth overhead, TileSpMM circumvents this bottleneck using an implicit thread-block-level address mapping strategy. During host preprocessing, we construct a lightweight mapping array indexed by the dispatched thread blocks. On the device, each block fetches its base row index once via `Fragment_RowMap`, and threads use this base address to directly write results to the global output matrix through coalesced memory transactions. This approach reduces costly global reordering to a simple addressing operation, achieving seamless sub-matrix integration with negligible overhead.

3.3 Load Balancing Strategy

In TileSpMM, the original matrix is partitioned into three distinct sub-matrices. Enforcing a fixed fragment count Thr per thread block often leads to severe load imbalance and degraded resource utilization. To mitigate this issue, we propose a two-stage tuning strategy: combining static cost analysis with runtime micro-benchmarking to systematically determine the best-performing Thr for each sub-matrix.

Since scheduling, computation, memory hierarchy access, and atomic operations impact performance distinctly, we formulate the overall execution cost $Cost(Thr)$ as the aggregation of four time-equivalent components, which will be detailed in Section 3.3.1:

$$Cost(Thr) = C_{base}(Thr) + C_{tail}(Thr) + C_{L2}(Thr) + C_{atom}(Thr) \quad (1)$$

3.3.1 Static Cost Modeling. First, we establish the hardware boundaries. Utilizing compiler-reported register and shared memory (SMEM) allocations for each kernel, we calculate the maximum concurrent thread blocks per SM, denoted as B_{sm} . The global execution capacity is formulated as $Cap = B_{sm} \times N_{sm}$, where N_{sm} is the total number of Streaming Multiprocessors (SMs) on the target GPU.

Next, we quantify the execution cost across various Thr candidates. Suppose a sub-matrix comprises M tiles, where the i -th tile contains L_i fragments. A given Thr partitions the i -th tile into $b_i = \lfloor L_i / Thr \rfloor$ independent fragment groups, each containing up to $\lfloor L_i / b_i \rfloor$ fragments and exclusively assigned to a single thread block. The total number of dispatched thread blocks is:

$$B_{total} = \sum_{i=1}^M b_i \quad (2)$$

Since kernels execute concurrently, tasks of varying sizes overlap, rendering traditional discrete wave models inaccurate. Instead, we evaluate the base computation latency using a continuous wave-based execution model: $C_{base}(Thr) = (B_{total}/Cap) \times Thr$. Additionally, to account for the tail latency where kernel completion is bottlenecked by the most heavily loaded thread block on the critical path, we introduce a tail penalty: $C_{tail}(Thr) = \max_{1 \leq i \leq M}(\lceil L_i/b_i \rceil)$.

Assigning multiple thread blocks to a single tile ($b_i > 1$) forces these blocks to execute across different SMs, introducing severe memory and synchronization overheads.

First, it degrades temporal locality due to L2 cache contention. When a tile is partitioned into b_i concurrent thread blocks, each block independently fetches distinct portions of the dense matrix. These simultaneous accesses can easily exceed the L2 cache capacity, leading to mutual data eviction. Since any given thread block faces eviction pressure from the remaining $b_i - 1$ blocks, we formulate the inter-block interference ratio as $(b_i - 1)/b_i = 1 - 1/b_i$. The L2 cache penalty is modeled as:

$$C_{L2}(Thr) = \sum_{i=1}^M \left(1 - \frac{1}{b_i}\right) \times \lambda_{L2} \quad (3)$$

where $\lambda_{L2} = (S_{iter} \times Cap)/S_{L2}$. Here, S_{iter} denotes the total data volume fetched per thread block during a single iteration, and S_{L2} is the physical L2 capacity. It directly quantifies the instantaneous memory footprint pressure on the hardware cache.

Second, it induces write conflicts. The b_i thread blocks accumulate their partial results into the same output row via atomic additions, generating $b_i - 1$ conflicts. This penalty is proportional to the dense matrix column width ($colB$). To reflect GPU memory coalescing, the divisor 32 maps the element-wise column width to the number of warp-level memory transactions. Using λ_{atom} , an offline-benchmarked parameter representing the relative latency overhead of atomic additions compared to conflict-free memory transactions, the atomic conflict cost is modeled as:

$$C_{atom}(Thr) = \sum_{i=1}^M (b_i - 1) \times \lambda_{atom} \times \frac{colB}{32} \quad (4)$$

3.3.2 Runtime Micro-benchmarking. The static cost model evaluates the candidate space and identifies the top-4 Thr configurations with the lowest predicted costs. Because static analysis cannot perfectly capture complex runtime behaviors, the algorithm conducts a lightweight runtime micro-benchmark using these top candidates on the target GPU to determine the empirically best-performing Thr . Crucially, the overhead of this runtime profiling is a one-time cost, which is amortized across the numerous SpMM iterations typical in downstream applications such as iterative solvers.

3.4 Algorithm Description

To efficiently map the three sub-matrices partitioned on the CPU host to the TCUs on the GPU, TileSpMM designs customized low-level kernels. By employing memory locality optimizations and hardware-level instruction tuning, these kernels schedule and load the pre-packed dense fragments into registers and SMEM to maximize the computational throughput of the TCUs.

Algorithm 1 Pipelined Kernel for 64/16-Row Tile SpMM.

Input: Fragment_Values(SpMatrix A), DnMatrix B, Fragment_ColIdx, Fragment_RowPtr, Fragment_RowMap
Output: DnMatrix C

```

1: ▶ Double buffers in Shared Memory.
2: __shared__ SMEM_B[2][BLOCK_SIZE]
3: float reg_C[4][4] ← 0
4: unsigned reg_B[4][2]
5: int4 reg_A
6: ▶ Fetch tile boundaries using thread block ID.
7: tile_start ← Fragment_RowPtr[blockIdx.x]
8: num_tiles ← Fragment_RowPtr[blockIdx.x + 1] - tile_start
9: ▶ Vectorized async pre-fetch for the first LHS Tile.
10: cp.async(SMEM_B[0], B[Fragment_ColIdx[tile_start]])
11: cp.async.commit()
12: for i = 0 to num_tiles - 1 do
13:   rdSel ← i (mod 2)
14:   wrSel ← (i + 1) (mod 2)
15:   if i < num_tiles - 1 then
16:     idx_next ← Fragment_ColIdx[tile_start + i + 1]
17:     cp.async(SMEM_B[wrSel], B[idx_next])
18:     cp.async.commit()
19:   end if
20:   cp.async.wait_group()
21:   __syncthreads()
22:   ▶ Fetch A directly from compacted Values array.
23:   reg_A ← Fragment_Values[tile_start + i]
24:   ▶ Fetch B via 4x ldmatrix.x2 (transposed).
25:   reg_B[0..3] ← ldmatrix.x2.trans(SMEM_B[rdSel])
26:   ▶ 4x MMA computations per tile.
27:   MMA(reg_C[0], reg_A, reg_B[0], reg_C[0])
28:   MMA(reg_C[1], reg_A, reg_B[1], reg_C[1])
29:   MMA(reg_C[2], reg_A, reg_B[2], reg_C[2])
30:   MMA(reg_C[3], reg_A, reg_B[3], reg_C[3])
31: end for
32: row_meta ← Fragment_RowMap[blockIdx.x]
33: C ← AtomicOrDirectWrite(reg_C, row_meta)

```

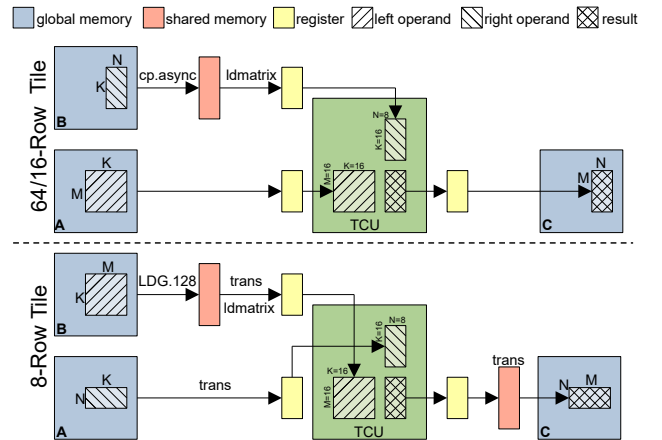


Figure 5: Memory access and execution flow of the 64/16-Row Tile mode (top) and 8-Row Tile mode (bottom).

3.4.1 64/16-Row Tile Kernels: Asynchronous Pipeline Design. The sub-matrices associated with the 64- and 16-row tiles correspond to the dense computation regions within the original matrix, exhibiting extremely high data reuse. Consequently, the primary design objective for these kernels is to maximize the overlap between computation and memory access. To achieve this, we allocate double

Algorithm 2 Swap-and-Transpose Kernel for 8-Row Tile SpMM.

Input: Fragment_Values(SpMatrix A), DnMatrix B ,
Fragment_ColIdx, Fragment_RowPtr, Fragment_RowMap

Output: DnMatrix C

```

1: ▶ Single buffer in Shared Memory.
2: __shared__ SMEM_B[BLOCK_SIZE]
3: ▶ Explicit register allocation for swapped M=16 mapping.
4: float reg_C[2][4] ← 0
5: unsigned reg_B[4]
6: int2 reg_A
7: ▶ Fetch tile boundaries using thread block ID.
8: tile_start ← Fragment_RowPtr[blockIdx.x]
9: num_tiles ← Fragment_RowPtr[blockIdx.x + 1] - tile_start
10: for i = 0 to num_tiles - 1 do
11:   idx ← Fragment_ColIdx[tile_start + i]
12:   ▶ Synchronous 128-bit vectorized load to SMEM.
13:   (int4*)SMEM_B ← (int4*)B[idx]
14:   __syncthreads()
15:   ▶ Fetch compacted A array (Acts as Right Operand).
16:   reg_A ← (int2*)Fragment_Values[tile_start + i]
17:   ▶ Fetch B via ldmatrix.x4 (Acts as Left Operand).
18:   reg_B ← ldmatrix.x4.trans(SMEM_B)
19:   ▶ Swapped TC computation:  $C^T = B^T \times A^T$ .
20:   MMA(reg_C[0], reg_B, reg_A, reg_C[0])
21:   MMA(reg_C[1], reg_B, reg_A, reg_C[1])
22:   __syncthreads()
23: end for
24: ▶ Remap 8x8 transposed results to float memory layout.
25: reg_C ← StoreTransposedMMA(reg_C)
26: row_meta ← Fragment_RowMap[blockIdx.x]
27: C ← AtomicOrDirectWrite(reg_C, row_meta)

```

buffers in the shared memory (SMEM), as illustrated in the upper half of Figure 5 and Algorithm 1.

Because matrix A is restructured into contiguous fragments during host preprocessing, we utilize 128-bit vectorized instructions to fetch it directly from global memory into registers as the left operand. For the dense matrix B , we design an efficient multi-stage data path routing data from global memory, through SMEM, and finally into registers. By employing vectorized loads for matrix B , a single thread can concurrently fetch multiple dense elements sharing the same column index (specifically, 8 elements for half-precision or 4 elements for TF32/float formats). This optimized data path substantially amortizes the memory access cost associated with sparse metadata, effectively alleviating the bandwidth pressure caused by index reading.

Building upon these optimized memory access patterns, we construct a robust multi-stage software pipeline. While the TCU executes continuous MMA operations on the active fragments—using data fetched from the current SMEM buffer via the *ldmatrix* instruction—the subsequent data fragment is asynchronously pre-fetched into the alternate SMEM buffer using the *cp.async* instruction. This asynchronous pipelining strategy effectively hides long-latency global memory accesses, thereby significantly enhancing the execution efficiency of the TCU pipeline.

3.4.2 8-Row Tile Kernel: Operand Swapping Strategy. The 8-row execution mode is specifically tailored for extremely sparse regions, aiming to reduce the computation granularity of row tiles to 8 to substantially mitigate the zero-padding overhead. However, conventional TCU MMA instructions (e.g., m16n8k16) typically dictate a minimum calculation granularity of 16 for the row dimension (M) of the left operand. If the sparse matrix A is directly assigned as

the left operand, the fragment height fed into the TCU is rigidly constrained to 16. This dimensional restriction inevitably leads to substantial hardware underutilization within the 8-row tiles.

To overcome this bottleneck, the 8-row kernel adopts an operand swapping strategy, utilizing the equivalent matrix multiplication transformation $C^T = B^T \times A^T$ to reverse the data flow. As shown in the lower half of Figure 5 and Algorithm 2, this kernel retains the original strategy of loading matrix B into SMEM via vectorized instructions, but it fundamentally restructures the register mapping logic prior to entering the TCU. Specifically, matrix A no longer enters the left operand; instead, it is directly loaded into registers via vectorized instructions to explicitly serve as the right operand for MMA. Since the TCU’s dimensional requirement for the right operand (the N dimension) is merely 8, this architectural feature allows downsizing the input fragment’s row height to 8.

Concurrently, the dense matrix B assumes the role of the left operand. When extracting data from SMEM, the kernel invokes the *ldmatrix.x4.trans* instruction, which possesses hardware-level transpose semantics. Leveraging this native underlying transpose support, the instruction interleaves the data from SMEM into registers with minimal micro-architectural overhead, efficiently converting matrix B into B^T to comply with the left operand format requirements. Ultimately, the resulting C^T residing in the registers is remapped back to the standard memory layout via a dedicated register-shuffling interface (*StoreTransposedMMA*) and safely written back to global memory. Without requiring any modifications to the underlying hardware, this design achieves fine-grained, 8-row TCU acceleration in an end-to-end manner.

4 Evaluation

4.1 Experimental Setup

Platform and Implementation: We implement TileSpMM using CUDA 12.8 and integrate our customized kernels into the PyTorch framework via ATen C++ extensions. All evaluations are conducted on NVIDIA H100 and RTX 5090 GPUs, representing successive generations of Tensor Core architectures; their theoretical peak performances are summarized in Table 1. This unified environment supports both standalone kernel-level benchmarks and end-to-end Graph Convolutional Network (GCN) training evaluations.

Table 1: Peak performance (TFLOPS) of the NVIDIA GPUs used in our experiments.

NVIDIA GPU	Precision	Perf. (TFLOPS)
RTX 5090 [35] 32 GB GDDR7, 1.79 TB/s	FP16/FP32 (mixed)	209.5
	TF32	104.8
H100 SXM [34] 80 GB HBM3, 3.35 TB/s	FP16/FP32 (mixed)	1000
	TF32	500

Baselines: We compare **TileSpMM** against six competitive SpMM frameworks, categorized by hardware execution units and numerical precisions in Table 2 to ensure fair comparisons. For TCUs, we benchmark against **cuSPARSE**, **FlashSparse**, **SMaT**, **SSpMM**, and **Acc-SpMM** under their supported FP16 or TF32 precisions. Additionally, we include **RoDe**, which executes in FP32 on

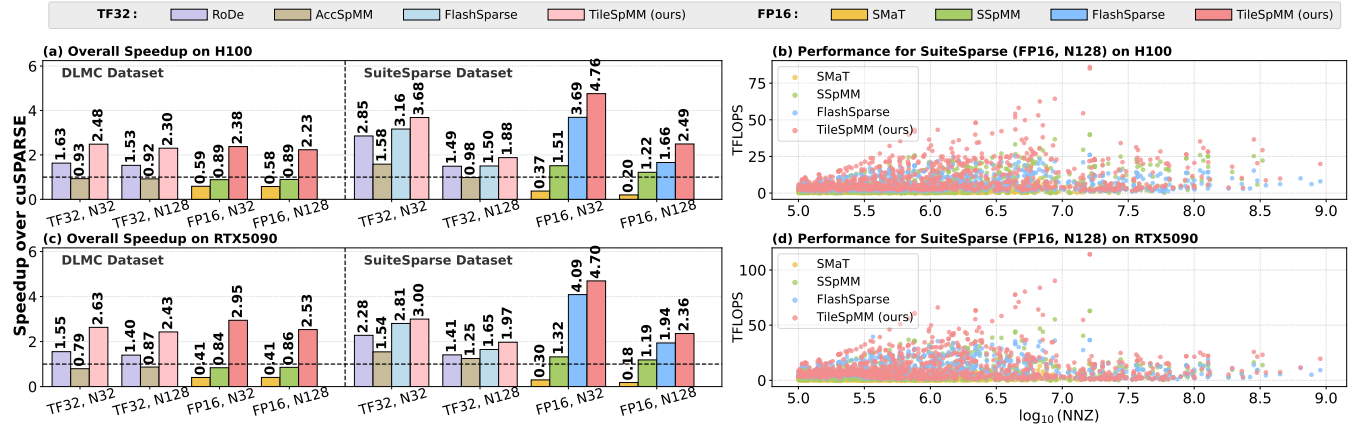


Figure 6: Kernel performance evaluation on H100 and RTX 5090 GPUs. (a, c) Overall speedup of different methods over cuSPARSE on the DLMC and SuiteSparse datasets. The configuration labels (e.g., FP16-32) denote the numerical precision (FP16) and the number of columns of the dense matrix B (32). (b, d) Absolute performance of the evaluated methods for SuiteSparse matrices under the FP16-128 configuration, plotted against the number of nonzeros.

Table 2: Comparison of evaluated SpMM frameworks by hardware execution unit and supported numerical precision.

Framework	Execution Unit	Evaluated Precision		
		FP16	TF32	FP32
TileSpMM (Ours)	Tensor Cores	✓	✓	
cuSPARSE [36]	Tensor Cores	✓	✓	
FlashSparse [44]	Tensor Cores	✓	✓	
SMaT [38]	Tensor Cores	✓		
SSpMM [50]	Tensor Cores	✓		
Acc-SpMM [52]	Tensor Cores		✓	
RoDe [39]	CUDA Cores			✓

CUDA cores. As the current SOTA for traditional CUDA core-based SpMM, RoDe provides a critical reference point to demonstrate the architectural acceleration achieved by our TCU-centric design.

Datasets: To evaluate the cross-domain efficiency of TileSpMM, we construct a comprehensive benchmark suite from three distinct data sources. First, for AI workloads, we utilize the DLMC suite [13], which provides real neural network weights strictly pruned to specific sparsity levels. Second, for scientific computing, we select diverse irregular matrices from the SuiteSparse Matrix Collection [7]. Finally, to cover graph analytics and GCN scenarios, we select 15 representative real-world graph datasets (summarized in Table 3), categorized into the Stanford Network Analysis Project (SNAP) [24], PyTorch Geometric (PyG) built-in [9], classic GCN benchmarks [20], and large-scale irregular graphs.

4.2 Performance Comparison

Building upon the diverse benchmark suite introduced previously, this section focuses on the kernel-level performance evaluation against state-of-the-art (SOTA) baselines on H100 and RTX 5090 GPUs. Specifically, we utilize the DLMC dataset to represent AI

Table 3: Datasets for evaluation. AvgL denotes the average NNZs per row. 1: SNAP [24]; 2: PyG built-in [9]; 3: classic GCN benchmarks [20]; 4: large irregular graphs.

Dataset	Abbr.	#Vertex	#Edge	AvgL
roadNet-CA ¹	rCA	1,971,281	5,533,214	2.81
roadNet-PA ¹	rPA	1,090,920	3,083,796	2.83
web-BerkStan ¹	WB	685,230	7,600,595	11.09
PROTEINS_full ²	PR	43,466	162,088	3.73
DD ²	DD	334,925	1,686,092	5.03
Yeast ²	Yt	1,710,902	3,636,546	2.13
YeastH ²	YH	3,138,114	6,487,230	2.07
citeseer ³	cite	3,327	9,228	2.77
pubmed ³	pub	19,717	88,651	4.50
ppi ³	ppi	56,944	818,716	14.38
amazon0505 ⁴	ama	410,236	4,878,874	11.89
reddit ⁴	red	232,965	114,848,857	492.99
protein ⁴	prot	132,534	79,255,038	598.00
com-amazon ⁴	com	334,863	1,851,744	5.53
soc-BlogCatalog ⁴	soc	88,784	2,093,195	23.58

Table 4: Performance Speedup Distribution of TileSpMM on RTX 5090 (SuiteSparse, $N = 128$, $NNZ \geq 100,000$)

Speedup	FP16				TF32			
	Flash.	SMaT	SSpMM	cuSP.	Acc.	Flash.	RoDe	cuSP.
< 1	9.50%	0.10%	2.89%	0.26%	32.45%	13.33%	28.47%	2.26%
1 – 1.5	73.75%	0.00%	56.23%	13.80%	30.77%	66.30%	33.21%	38.02%
1.5 – 2	12.59%	0.21%	16.40%	31.60%	13.21%	16.85%	14.86%	23.52%
≥ 2	4.16%	99.69%	24.47%	54.34%	23.57%	3.52%	23.46%	36.20%
Mean	1.27x	13.72x	2.00x	2.36x	1.64x	1.25x	1.41x	1.97x
Max	4.95x	253.62x	110.69x	24.45x	194.99x	3.20x	7.05x	15.54x

workloads and the SuiteSparse matrix collection for scientific computing, from which we exclude trivial matrices with $NNZ \leq 100,000$ to ensure rigorous evaluation.

Figure 6 summarizes the overall geometric mean speedup and absolute throughput (TFLOPS) across the evaluated datasets. TileSpMM demonstrates a significant advantage: on the RTX 5090 GPU for the SuiteSparse dataset ($N = 128$), it achieves a $2.36\times$ geometric mean speedup over cuSPARSE in FP16 precision, while also outperforming the optimized FlashSparse baseline ($1.27\times$). Similarly, under TF32 precision, TileSpMM delivers a $1.97\times$ speedup over cuSPARSE and surpasses FlashSparse by $1.25\times$. Furthermore, the scatter plots in Figures 6(b) and 6(d) reveal that TileSpMM not only raises the performance floor but also reaches high peak throughputs (exceeding 100 TFLOPS on the RTX 5090), establishing a clear upper performance envelope. This absolute advantage is primarily attributed to the algorithm’s efficient memory hierarchy utilization and dynamic workload-balancing.

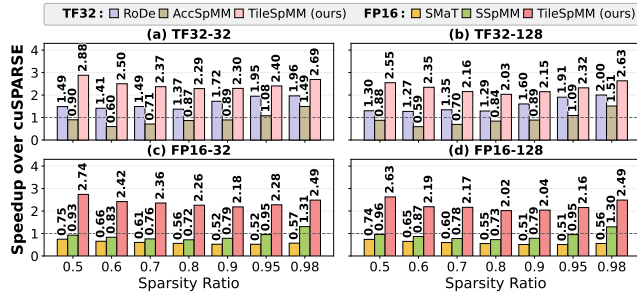


Figure 7: Performance comparison of TileSpMM and other baselines on the DLMC dataset using the H100 GPU.

Table 4 presents the speedup distribution on the RTX 5090 GPU for the SuiteSparse dataset. Under FP16 precision, TileSpMM achieves a $> 2\times$ speedup over cuSPARSE on 54.34% of the matrices (up to $24.45\times$), and a mean speedup of $13.72\times$ over SMat. In the TF32 configuration, it outperforms cuSPARSE on 97.74% of the test matrices. Such robust performance across severely irregular distributions stems from our variable-size tiling, which effectively mitigates poor data reuse and zero-padding overheads on TCUs.

Beyond scientific computing, TileSpMM delivers consistent performance improvements in AI workloads. As illustrated in Figures 6(a) and 6(c), TileSpMM achieves the highest overall geometric mean speedup on the DLMC dataset across all evaluated configurations. Figure 7 further details the performance breakdown across varying sparsity ratios. Across all four configurations (TF32/FP16, $N = 32/128$), TileSpMM consistently outperforms all SOTA baselines, achieving peak speedups of up to $2.74\times$. Furthermore, across all tested sparsity patterns, TileSpMM establishes a strict performance lower bound, guaranteeing at least a $2.0\times$ speedup over cuSPARSE. This underscores its robust adaptability to diverse intramatrix sparsity distributions. Finally, as demonstrated in Section 4.4, these kernel-level performance gains translate directly into significant end-to-end latency reductions when evaluated on real-world graph datasets in GCN scenarios.

However, a performance bottleneck emerges when true sparsity exceeds 99.99%. In such hyper-sparse scenarios, our multi-granularity partitioning degrades to the 8-row fallback mode, turning the preliminary evaluation into pure overhead. Conversely, specific baselines excel here through specialized designs: Acc-SpMM

enhances block density via reordering, FlashSparse globally adopts an 8×1 granularity without multi-level scheduling costs, and RoDe bypasses Tensor Core constraints via row decomposition on CUDA cores. This trade-off clearly delineates the architectural boundary of TileSpMM, providing clear insights for runtime engine selection in hyper-sparse applications.

4.3 Ablation Study

To evaluate the specific contributions of our optimization strategies, we conducted an ablation study across nine representative datasets. As illustrated in Figure 9, we compare three versions of our algorithm: (V1) the baseline employing only 8-row fixed partitioning; (V2) the configuration augmented with multi-granularity partitioning; and (V3) the full model incorporating both multi-granularity partitioning and the load-balancing module. The results demonstrate that each optimization component consistently contributes to the overall efficiency.

Effectiveness of Multi-granularity Partitioning. The multi-granularity partitioning strategy (V2) yields substantial gains over the baseline (V1), achieving speedups ranging from $1.12\times$ (raefsky4) to $2.13\times$ (TSOPF_RS_b2383). This improvement stems from overcoming the limitations of fixed granularity. The granularity distribution, visualized as inset pie charts in Figure 9, reveals that datasets achieving substantial speedups, such as TSOPF_RS_b2383 and pattern1, exhibit a heavily mixed distribution of the finest (8-Row) and coarsest (64-Row) thread block sizes. Consequently, V2 flexibly assigns 64-Row to dense clusters to maximize throughput, while preserving 8-Row for sparse regions to strictly control zero-padding overhead.

Evaluation of Load Balancing. Although dynamic partitioning improves micro-level execution efficiency, imbalanced workloads across thread blocks can still cause pipeline bubbles that stall global execution. By further integrating the load-balancing strategy (V3), our algorithm achieves an additional performance leap of $1.14\times$ to $1.47\times$ over V2. For instance, on the pdb1HYS and pattern1 matrices, load-balancing delivers a secondary boost of $1.47\times$ and $1.40\times$, respectively. This confirms that our strategy successfully redistributes long-tail tasks, ensuring balanced execution times across GPU SMs.

Synergy of Proposed Designs. The performance evolution from V1 to V3 demonstrates the synergy of our components. While multi-granularity partitioning addresses fine-grained instruction-data alignment at the micro-level, load-balancing mitigates global scheduling imbalance at the macro-level. Collectively, these techniques enable the system to achieve peak performance across diverse sparsity patterns.

4.4 End-to-end GCNs Performance

To evaluate the practical performance of TileSpMM within full GCN [12, 21, 41] models, we evaluate the end-to-end training execution time. In modern GNN frameworks, a typical GCN layer forward pass consists of sparse neighbor aggregation (the SpMM phase) followed by a dense feature update (the GEMM phase). We implement the end-to-end pipeline by integrating our optimized TileSpMM kernel into PyTorch [40] via ATen C++ extensions, utilizing custom autograd functions to offload both forward and backward passes. Following typical settings, we construct 5-layer GCN models with

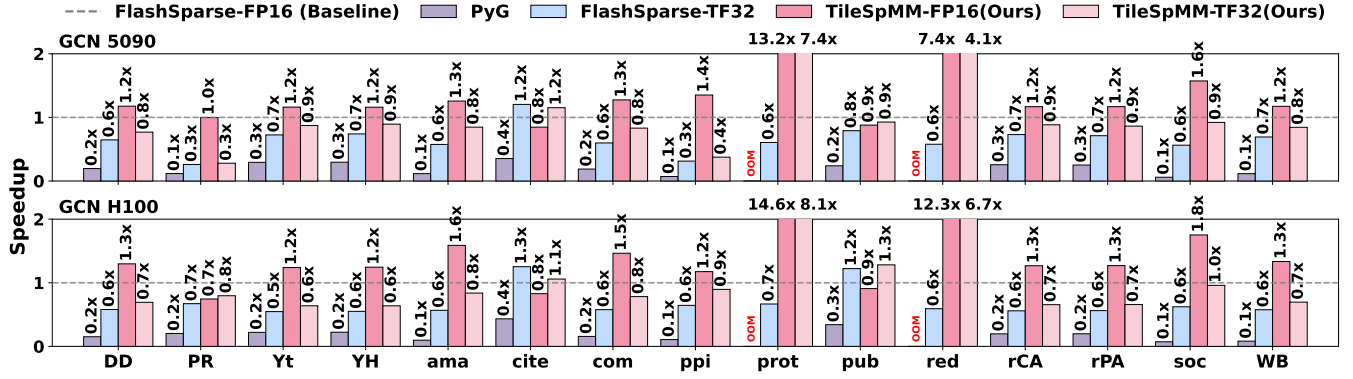


Figure 8: End-to-end performance speedup of GCN models on RTX 5090 (top) and H100 (bottom) GPUs. All speedups are globally normalized against the FlashSparse-FP16 baseline (dashed line at 1.0×). OOM indicates PyG out-of-memory failures.

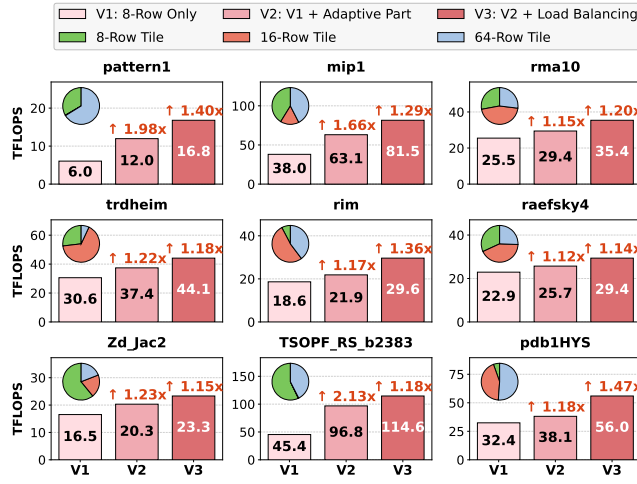


Figure 9: Performance breakdown for the ablation study. Adding dynamic partitioning (V2) and load-balancing (V3) consistently improves throughput across all benchmarks. The inset pie charts reveal the proportion of selected row granularities (64/16/8-row) for each matrix.

a hidden dimension of 128 and train them for 300 epochs. Since the graph topology remains static, the one-time CSR-to-tiled format conversion is cached, making the preprocessing overhead negligible when amortized over the full training process.

Figure 8 evaluates TileSpMM’s GCN execution efficiency on RTX 5090 and H100 GPUs against the SOTA FlashSparse [44] (with its FP16 implementation set as the 1.0× baseline). By effectively minimizing load imbalance via multi-granularity tiling, TileSpMM achieves exceptional performance across both architectures. In FP16 precision, TileSpMM delivers geometric mean speedups of 1.54× (up to 13.16×) on the RTX 5090 and 1.67× (up to 14.65×) on the H100. This superiority extends to TF32 precision, where TileSpMM consistently outperforms the FlashSparse-TF32 implementations, achieving geometric mean speedups of 1.61× (up to 12.17×) and 1.64× (up to 12.15×) on the RTX 5090 and H100, respectively.

This consistent acceleration is particularly evident on medium-to-large and extremely irregular graphs (e.g., protein, reddit). While TileSpMM matches the baseline on extremely small graphs (e.g., citeseer)—because the scheduling overhead of dynamic block division slightly offsets the load-balancing benefits on light workloads—its superiority emerges on complex graphs. Furthermore, the native PyG [9] framework remains uncompetitive, suffering from severe performance degradation and Out-Of-Memory (OOM) failures on large-scale datasets. Overall, by optimizing both memory access and workload distribution, TileSpMM establishes a new SOTA in end-to-end efficiency for real-world graph training.

4.5 Preprocessing Overhead Analysis

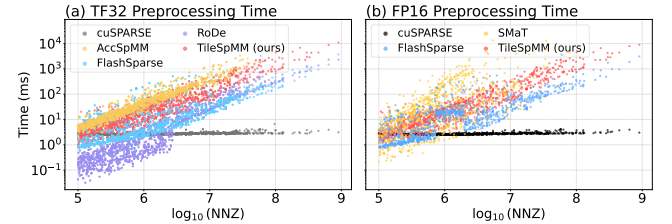


Figure 10: Preprocessing time of different SpMM methods under (a) TF32 and (b) FP16.

Figure 10 presents the preprocessing time of various SpMM methods. Due to the locality-aware partitioning and load-balancing strategies executed on the host CPU, TileSpMM inherently incurs a higher preprocessing overhead compared to lightweight libraries like cuSPARSE. Nevertheless, its preprocessing time remains comparable to those of methods such as Acc-SpMM and SMaT.

Specifically, the preprocessing pipeline comprises four stages: 1) feature extraction scans the row pointers once in $O(M)$ time; 2) matrix partitioning traverses the rows and nonzeros in $O(M + NNZ)$ time; 3) conversion to the Window-based format takes $O(M + NNZ \log C_{max} + N_{dense})$, where C_{max} is the maximum number of nonempty columns per tile and N_{dense} is the number of elements in the zero-padded tensor; and 4) the static cost model evaluates

load-balancing thresholds over a bounded hardware-dependent search space in $O(M+N)$ time. The total preprocessing complexity is therefore $O(M+N+NNZ \log C_{max} + N_{dense})$, consistent with the approximately linear trends observed in Figure 10.

This one-time preprocessing cost is amortized when the sparse matrix structure is reused across iterations, as in GCN training and iterative graph analytics. Our end-to-end GCN evaluation (Section 4.4) shows that the speedups achieved by the TCU-optimized kernels outweigh the preprocessing overhead, resulting in better end-to-end performance than the evaluated baselines.

4.6 Floating-point Accuracy of TileSpMM

To evaluate the numerical accuracy of TileSpMM, we compare its outputs against those of a naive serial CPU implementation. The CPU baseline uses double-precision floating-point computation and serves as the numerical reference. We initialize both sparse and dense input matrices with deterministic pseudo-random floating-point values generated by a linear congruential generator.

Table 5: Numerical Error Metrics against FP64 Baseline

Matrix	FP16		TF32	
	Avg.	Max.	Avg.	Max.
patt...	5.05E-03	1.03E-01	1.32E-02	2.59E-01
mip1	2.60E-03	3.17E-01	6.78E-03	8.28E-01
rma1...	1.88E-03	2.11E-02	4.81E-03	5.12E-02
trdh...	2.50E-03	2.32E-02	6.45E-03	5.97E-02
rim	1.75E-03	1.83E-02	4.48E-03	5.59E-02
raef...	2.22E-03	1.92E-02	5.70E-03	5.70E-02
Zd_J...	1.40E-03	5.08E-02	3.62E-03	1.11E-01
TSOP....	3.88E-03	6.02E-02	1.03E-02	1.47E-01
pdb1...	2.95E-03	2.68E-02	7.59E-03	5.83E-02

Table 5 reports the average and maximum numerical errors of TileSpMM under FP16 and TF32. Across all evaluated sparse matrices, the average errors remain on the order of 10^{-3} for FP16 and between 10^{-3} and 10^{-2} for TF32. These deviations primarily arise from the limited mantissa precision of TCU formats and differences in parallel accumulation order. Overall, they remain within an acceptable range for practical SpMM applications, indicating that TileSpMM achieves substantial acceleration while maintaining sufficient accuracy for downstream workloads.

5 Related Work

SpMM is a fundamental kernel in scientific computing [2, 23], graph analytics [4, 16, 46], and deep learning [14, 48]. Existing optimizations can be characterized by how they map irregular sparsity to hardware architectures, including GPUs and specialized memory-efficient accelerators [27].

From Thread-Level Scheduling to TCU Constraints. Prior SpMM optimizations on CUDA cores explored fine-grained, thread-level memory management to handle irregular nonzeros (e.g., Sputnik [10], GE-SpMM [17], ASpT [15], and others [19, 26, 30]). Advanced techniques like MergePath-SpMM [43] and RoDe [39] employ fine-grained 1D/2D element-wise workload distribution to

mitigate severe load imbalance in power-law graphs. While offering flexibility, these traditional approaches often fall short of the massive throughput demanded by modern workloads. On the other hand, accelerating SpMM on TCU offers immense performance potential, yet it is strictly constrained by rigid MMA data layouts. Consequently, early solutions relying on strict structured sparsity assumptions (e.g., Magicube [25]) lack universality, forcing a paradigm shift toward tiling at the vector and thread block levels for unstructured data.

The Trap of Static Single-Granularity on TCUs. To map unstructured sparsity onto TCUs, current SOTA solutions [45] have universally adopted static single-granularity tiling. For instance, VectorSparse [5] pioneered 1D vector-wise sparse groups, while TC-GNN [47] and DTC-SpMM [8] mapped computations using a uniform 16×1 vector granularity. Recently, FlashSparse [44] optimized this by pushing the boundary to an extreme 8×1 to further minimize zero padding. However, our profiling reveals that all these methods are trapped in a fundamental dilemma driven by their density-unaware, single-granularity design. Real-world matrices inherently exhibit both localized dense sub-matrices and extremely sparse regions. Consequently, a static uniform granularity forces a fatal compromise: coarse grains suffer massive computation waste in sparse tails, whereas extreme fine grains (e.g., 8×1) shatter the 2D $M \times N$ register and SMEM reuse potential in dense zones, incurring prohibitive indexing overhead.

Breaking the Bottleneck via Distinct Tiling. While tiling dataflows are proven in other sparse operations [31, 32, 51], applying them to unstructured SpMM on NVIDIA TCUs faces severe layout conflicts. To escape the static-granularity trap, we propose TileSpMM, the first intra-matrix distinct tiling algorithm for modern TCUs. By dynamically partitioning matrices based on local row-wise density, TileSpMM reclaims GEMM-equivalent register reuse in sub-matrices via coarse tiles, while eliminating zero padding in sparse tails via fine tiles. To resolve the load imbalance introduced by distinct granularity and computational heterogeneity [6, 39, 47], TileSpMM introduces a novel concurrent multi-stream execution engine and a hardware-aware optimizer. By dynamically hiding scheduling latencies, TileSpMM successfully bridges the SpMM-GEMM performance gap across diverse density distributions.

6 Conclusion

In this paper, we propose TileSpMM, a new distinct tiling algorithm using modern TCUs to accelerate unstructured SpMM. We identify that single-granularity tiling inherently compromises between zero-padding and data locality, and propose a locality-aware dynamic partitioning algorithm coupled with tailored load-balancing to fully utilize TC units. Experimental results on the latest NVIDIA H100 and RTX 5090 GPUs show that TileSpMM achieves consistent performance superiority over SOTA SpMM methods.

Acknowledgments

We are very grateful to all reviewers for their invaluable comments. Yuechen Lu is the corresponding author of this paper. This work was supported by the National Key R&D Program of China (Grant No. 2023YFB3001604) and the National Natural Science Foundation of China (Grant No. U23A20301 and No. 62372467).

References

- [1] Paul Almasan, José Suárez-Varela, Krzysztof Rusek, Pere Barlet-Ros, and Albert Cabellos-Aparicio. 2022. Deep reinforcement learning meets graph neural networks: Exploring a routing optimization use case. *Computer Communications* 196 (2022).
- [2] Hartwig Anzt, Stanimire Tomov, and Jack J Dongarra. 2015. Accelerating the LOBPCG method on GPUs using a blocked sparse matrix vector product.. In *SpringSim '15*.
- [3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020).
- [4] Khac-Hoai Nam Bui, Jiho Cho, and Hongsuk Yi. 2022. Spatial-temporal graph neural network for traffic forecasting: An overview and open research issues. *Applied Intelligence* 52, 3 (2022).
- [5] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. 2021. Efficient tensor core-based gpu kernels for structured sparsity under reduced precision. In *SC '21*.
- [6] Helin Cheng, Wenxuan Li, Yuechen Lu, and Weifeng Liu. 2023. HASpGEMM: Heterogeneity-Aware Sparse General Matrix-Matrix Multiplication on Modern Asymmetric Multicore Processors. In *ICPP '23*.
- [7] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *Acm transactions on mathematical software (toms)* 38, 1 (2011).
- [8] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2024. Dtc-spm: Bridging the gap in accelerating general sparse matrix multiplication with tensor cores. In *ASPLOS '24*.
- [9] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).
- [10] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse gpu kernels for deep learning. In *SC '20*.
- [11] Yingxue Gao, Lei Gong, Chao Wang, Teng Wang, Xi Li, and Xuehai Zhou. 2023. Algorithm/hardware co-optimization for sparsity-aware SpMM acceleration of GNNs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (2023).
- [12] Jaume Gibert, Ernest Valveny, and Horst Bunke. 2012. Graph embedding in vector spaces by node attribute statistics. *Pattern Recognition* 45, 9 (2012).
- [13] Google. 2020. Deep learning matrix collection (DLMC). <https://storage.googleapis.com/svk-sc2020/dlmc.tar.gz>. [Online; accessed July 29, 2026].
- [14] Torsten Hoefer, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research* 22, 241 (2021).
- [15] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and Ponuswamy Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *PPoPP '19*.
- [16] Md Saidul Hoque Anik, Pranav Badhe, Rohit Gampa, and Ariful Azad. 2024. isplib: A library for accelerating graph neural networks using auto-tuned sparse operations. In *WWW '24 Companion*.
- [17] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Ge-spm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC '20*.
- [18] Haonan Ji, Huimin Song, Shibo Lu, Zhou Jin, Guangming Tan, and Weifeng Liu. 2022. TileSpMM: A tiled algorithm for sparse matrix-sparse vector multiplication on gpus. In *ICPP '22*.
- [19] 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*.
- [20] Kristian Kersting, Nils M Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. 2016. Benchmark data sets for graph kernels. (2016).
- [21] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).
- [22] Sanjay Kumar, Abhishek Mallik, Anavi Khetarpal, and Bhawani Sankar Panda. 2022. Influence maximization in social networks using graph embedding and graph neural network. *Information Sciences* 607 (2022).
- [23] Andrew S Lan, Andrew E Waters, Christoph Studer, and Richard G Baraniuk. 2014. Sparse factor analysis for learning and content analytics. *Journal of Machine Learning Research* 15, 57 (2014).
- [24] Jure Leskovec and Rok Sosič. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016).
- [25] Shigang Li, Kazuki Osawa, and Torsten Hoefer. 2022. Efficient quantized sparse matrix operations on tensor cores. In *SC '22*.
- [26] Haocheng Lian, Qiyue Zhang, Xinran Zhao, Meichen Dong, Yijie Nie, Zhengyi Zhao, Junzhong Shen, Wei Guo, Chun Huang, Bingcai Sui, et al. 2026. Uni-STC: Unified Sparse Tensor Core. In *HPCA '26*.
- [27] Xiaobo Liu, Jianbin Fang, Lin Peng, Chun Huang, Zidong Du, Yongwei Zhao, and Zheng Wang. 2024. Mentor: A Memory-Efficient Sparse-Dense Matrix Multiplication Accelerator Based on Column-Wise Product. *ACM TACO* (2024).
- [28] Yuechen Lu and Weifeng Liu. 2023. Dasp: Specific dense matrix multiply-accumulate units accelerated general sparse matrix-vector multiplication. In *SC '23*.
- [29] Yuechen Lu, Hongwei Zeng, Marc Casas, and Weifeng Liu. 2026. Characterizing Matrix Multiplication Units across General Parallel Patterns in Scientific Computing. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
- [30] Yuechen Lu, Lijie Zeng, Tengcheng Wang, Xu Fu, Wenxuan Li, Helin Cheng, Dechuang Yang, Zhou Jin, Marc Casas, and Weifeng Liu. 2024. Amgt: Algebraic multigrid solver on tensor cores. In *SC '24*.
- [31] Zhengyang Lu and Weifeng Liu. 2023. TileSpTRSV: A tiled algorithm for parallel sparse triangular solve on GPUs. *CCF Transactions on High Performance Computing* (2023).
- [32] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. TileSpMV: A tiled algorithm for sparse matrix-vector multiplication on gpus. In *IPDPS '21*.
- [33] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: A tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *PPoPP '22*.
- [34] NVIDIA. 2022. NVIDIA H100 Tensor Core GPU Architecture. <https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c>.
- [35] NVIDIA. 2025. NVIDIA RTX Blackwell GPU Architecture. <https://images.nvidia.cn/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf>.
- [36] NVIDIA Corporation. 2024. NVIDIA cuSPARSE Library. <https://docs.nvidia.com/cuda/cusparse/>. Version 12.8.
- [37] NVIDIA Corporation. 2025. *CUDA Samples*. NVIDIA. <https://docs.nvidia.com/cuda/cuda-samples/>. Version 12.8.
- [38] Patrik Okanovic, Grzegorz Kwasniewski, Paolo Sylos Labini, Maciej Besta, Flavio Vella, and Torsten Hoefer. 2024. High performance unstructured spmm computation using tensor cores. In *SC '24*.
- [39] Meng Pang, Xiang Fei, Peng Qu, Youhui Zhang, and Zhaolin Li. 2024. A row decomposition-based approach for sparse matrix multiplication on GPUs. In *PPoPP '24*.
- [40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
- [41] Hongwu Peng, Xi Xie, Kaustubh Shivdikar, Md Amit Hasan, Jiahui Zhao, Shaoyi Huang, Omer Khan, David Kaeli, and Caiwen Ding. 2024. Maxk-gnn: Extremely fast gpu kernel design for accelerating graph neural networks training. In *ASPLOS '24*.
- [42] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008).
- [43] Mohsin Shan, Deniz Gurevin, Jared Nye, Caiwen Ding, and Omer Khan. 2023. Mergepath-spm: Parallel sparse matrix-matrix algorithm for graph neural network acceleration. In *ISPASS '23*.
- [44] 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 *PPoPP '25*.
- [45] Hongyu Wang, Mingzhen Li, Weile Jia, Hailong Yang, and Guangming Tan. 2025. FastSpMM: Leveraging Tensor Cores for Sparse Matrix Multiplication. In *CF '25*.
- [46] Hemeng Wang, Wenqing Lin, Qingxiao Sun, and Weifeng Liu. 2025. vGNN: Non-Uniformly partitioned full-graph GNN training on mixed GPUs. *CCF Transactions on High Performance Computing* (2025).
- [47] 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 *USENIX ATC '23*.
- [48] Yaoyu Wang, Xiao Guo, Junmin Xiao, De Chen, and Guangming Tan. 2025. GeneralSparse: Bridging the Gap in SpMM for Pruned Large Language Model Inference on GPUs. In *USENIX ATC '25*.
- [49] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2021. A pattern-based spgemm library for multi-core and many-core architectures. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2021).
- [50] Zeyu Xue, Mei Wen, Jianchao Yang, Minjin Tang, Zhongdi Luo, Jing Feng, Yang Shi, Zhaoyun Chen, Junzhong Shen, and Johannes Langguth. 2025. SSpmM: Efficiently Scalable SpMM Kernels across Multiple Generations of Tensor Cores. *IEEE Transactions on Parallel and Distributed Systems* (2025).
- [51] Enxin Yi, Jiarui Bai, Yijie Nie, Dan Niu, Zhou Jin, and Weifeng Liu. 2024. Leda: Leveraging Tiling Dataflow to Accelerate SpMM on HBM-Equipped FPGAs for GNNs. In *ICCAD '24*.
- [52] Haisha Zhao, San Li, Jiaheng Wang, Chunbao Zhou, Jue Wang, Zhikuan Xin, Shunde Li, Zhiqiang Liang, Zhijie Pan, Fang Liu, et al. 2025. Acc-spm: Accelerating general-purpose sparse matrix-matrix multiplication with gpu tensor cores. In *PPoPP '25*.