DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores

Ruibo Fan
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
rfan404@connect.hkust-gz.edu.cn

Wei Wang
The Hong Kong University of Science and Technology
Hong Kong SAR, China
weiwa@cse.ust.hk

Xiaowen Chu*
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
xwchu@ust.hk

Abstract

Sparse Matrix-Matrix Multiplication (SpMM) is a building-block operation in scientific computing and machine learning applications. Recent advancements in hardware, notably Tensor Cores (TCs), have created promising opportunities for accelerating SpMM. However, harnessing these hardware accelerators to speed up general SpMM necessitates considerable effort. In this paper, we undertake a comprehensive analysis of the state-of-the-art techniques for accelerating TC-based SpMM and identify crucial performance gaps. Drawing upon these insights, we propose DTC-SpMM, a novel approach with systematic optimizations tailored for accelerating general SpMM on TCs. DTC-SpMM encapsulates diverse aspects, including efficient compression formats, reordering methods, and runtime pipeline optimizations. Our extensive experiments on modern GPUs with a diverse range of benchmark matrices demonstrate remarkable performance improvements in SpMM acceleration by TCs in conjunction with our proposed optimizations. The case study also shows that DTC-SpMM speeds up end-to-end GNN training by up to 1.91× against popular GNN frameworks.

CCS Concepts: • **Computing methodologies** → **Shared memory algorithms**; • **Computer systems organization** → **Single instruction, multiple data**.

Keywords: Sparse Matrix-Matrix Multiplication, SpMM, unstructured sparsity, GPU, Tensor Core

ACM Reference Format:
Ruibo Fan, Wei Wang, and Xiaowen Chu. 2024. DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651378

*Corresponding Author

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04
https://doi.org/10.1145/3620666.3651378

1 Introduction

Sparse Matrix-Matrix Multiplication (SpMM) is an operation that multiplies two matrices A and B, where A is sparse and B is dense, and gives a dense matrix C. As a key kernel in the level-3 Sparse BLAS, SpMM is implemented with various computing libraries [2, 33, 37] and plays a pivotal role in scientific computing (SC) [14, 52] and machine learning [1, 50, 58]. There is a large body of works that improve SpMM performance on GPUs, with various optimization techniques [6, 11, 17, 18, 22, 62], such as storage formats, parallel strategies, reordering, and memory access optimizations. Most of these works are based on the general-purpose computing units of GPUs, namely CUDA cores, whereas the emergence of specialized GPU hardware, particularly **Tensor Cores (TCs)**, has enabled more promising opportunities for accelerating SpMM computations. Initially designed for mixed-precision dense matrix multiplication in deep-learning (DL), TCs offer significant computational power [48] and can also be applied to SC workloads [39].

However, it remains challenging to fully unleash the power of TCs to achieve substantial SpMM acceleration. TCs are designed to operate on dense data structures: its dense-oriented architecture may not be a natural fit to the sparse matrix operations in SpMM. Consequently, the integration of TCs into SpMM introduces additional intricacies that require careful attention and resolution to fully exploit their computational capabilities. Bridging this gap and effectively utilizing TCs for SpMM acceleration continue to be pivotal areas of research and optimization endeavors. Some prior research has focused on introducing TC-based SpMM to accelerate sparse DL workloads [3, 4, 12, 36, 59, 66]. They mainly target regular and small sparse weight matrices generated through pruning techniques, where non-zero elements follow strict distribution constraints (referred to as **structured SpMM**). However, these approaches face challenges in efficiently accelerating SpMM for larger, sparser, and highly irregular matrices (e.g., adjacency matrices in Graph Neural Networks (GNNs) [1]

and real-world matrices in SuiteSparse [32]), which we call **general SpMM**. While the recent work, TC-GNN [57], utilizes TCs for implementing general SpMM using various novel techniques, experiments reveal that significant performance improvements remain yet to be achieved. It has been observed that SpMM in TC-GNN, when executed on the recent RTX4090 GPU, achieves less than 8% utilization of the Tensor Core Pipeline and demonstrates less competitive performance compared to cuSPARSE's CUDA-core-based SpMM [37], especially on large matrices with long rows.

In this work, we focus on large and sparse matrices for GNNs and scientific computing. Starting with a thorough study of the state-of-the-art techniques for accelerating general SpMM by TCs, we identify four key barriers that impede the effective utilization of TCs. Drawing upon these observations, we propose **DTC-SpMM**[1], a novel approach with systematic techniques and optimizations designed for TCs. Our techniques encompass various aspects, including the development of novel memory-efficient formats that alleviate the issue of high memory consumption. Furthermore, we introduce a two-level hierarchical TCU-Cache-aware reordering technique to improve the density of TC blocks. At runtime, we propose shared-memory bypassing with PTX-level instructions [43] and sparse double buffering to overlap computations and memory access, which largely improve the TC pipeline utilization. Furthermore, we devise a strict-load-balance strategy and a simulation-based Selector to tackle input-adaptive workload imbalances.

We compare the efficacy of the proposed DTC-SpMM with TCGNN-SpMM [57], SparseTIR [63], Sputnik [11], Block-SpMM [44], VectorSparse [4], Flash-LLM [59], SparTA [66] and the widely-used SpMM in cuSPARSE library [37]. Experimental results on recent RTX4090 (Ada Lovelace) [41] and RTX3090 (Ampere) [40] GPUs demonstrate that DTC-SpMM achieves significant average (geometric mean or geomean) speedups over alternative approaches on diverse real-world matrices. Furthermore, breakdown analyses validate the effectiveness of our proposed techniques in reducing memory complexity, enhancing TC pipeline utilization, and alleviating load imbalance. A case study illustrates that DTC-SpMM accelerates end-to-end GNN training by 1.22-1.91× compared to DGL [53] and PyG [10], two widely used GNN training frameworks.

We summarize our contributions as follows:

- We conduct an in-depth study on recent state-of-the-art works that leverage Tensor Cores for general SpMM. Through this investigation, we identify the major gaps that hinder their ability to outperform highly-optimized CUDA-core-based counterparts.
- Building upon these insights, we propose DTC-SpMM, which incorporates a memory-efficient compressed format and a set of novel techniques explicitly tailored for accelerating general SpMM via Tensor Cores.
- DTC-SpMM achieves significant speedups over state-of-the-art TC-based and CUDA-core-based SpMM approaches, particularly on large matrices with diverse structures.

[1]DTC-SpMM's source code and datasets are publicly available at https://github.com/HPMLL/DTC-SpMM_ASPLOS24

2 Background and Related Work

2.1 NVIDIA GPU and Tensor Core

NVIDIA GPUs are highly parallel devices with multiple streaming multiprocessors (SMs) that contain CUDA cores, Tensor Cores, and other units. Kernels are launched with specific configurations of thread blocks and threads per block. Each thread block is scheduled onto an SM and runs concurrently. Threads within a warp (32 threads) execute the same instruction simultaneously in SIMT mode [42].

Tensor Cores (TCs) have been introduced since the Volta architecture [38]. TCs deliver notable speedups compared to CUDA cores and offer a wider range of precision options, such as TF32, BF16, and FP8 [48]. We target **TF32**, a more favorable alternative to FP32, as it offers superior numerical behaviors in both GNN and SC workloads [39, 48]. As dedicated units for accelerating matrix multiplication, TCs perform the computation $D_{frag} = A_{frag} \times B_{frag} + C_{frag}$, where A_{frag} and B_{frag} are input matrices of the shape $m \times k$ and $k \times n$, respectively. C_{frag} denotes the accumulator, and D_{frag} is the resulting matrix. Various matrix shapes are supported for each precision. In this paper, we use the notation $\boldsymbol{m \times k \times n}$ to represent the matrix shape for TCs. To program on TCs, CUDA provides warp-level APIs with the Matrix Multiply-Accumulate (MMA) semantic. These APIs enable a warp of 32 threads to collaboratively execute one or multiple dense matrix multiplications and accumulate the outputs. CUDA offers two sets of APIs for MMA, namely ***WMMA*** in high-level C code [42] and ***mma*** in low-level PTX code [43]. List 1 gives some simple examples. While the assemblers introduced in [60, 61] allow TC programming at the SASS (assembly) level, they are unofficial and may not be stable. Our work maintains its focus on the PTX level.

The *WMMA* APIs manage registers using *fragments*. Users only need to load tiles of data from global or shared memory into *fragments* without concerning the specific distribution of register variables across a warp (32 threads). In contrast, *mma* APIs offer greater flexibility, but users need to manage the registers themselves. With TF32 precision, *mma* APIs require the matrix shape to be $16 \times 8 \times 8$ or $16 \times 4 \times 8$.

Listing 1. C-level WMMA APIs and PTX-level instructions

```
wmma::fragment<16, 16, 8, tf32, row_major> A_frag;
wmma::load_matrix_sync(A, A_frag, ldm);
wmma::mma_sync(D_frag, A_frag, B_frag, C_frag);
-----------------------------------------------------
.reg %Ra<2>, %Rb<1>, %Rc<4>, %Rd<4>;
mma.sync.aligned.m16n8k4.row.col.f32.tf32.tf32.f32
  {%Rd0, %Rd1, %Rd2, %Rd3},
  {%Ra0, %Ra1}, {%Rb0},
  {%Rc0, %Rc1, %Rc2, %Rc3};
```

2.2 Related work

SpMM implements the following computation: $C_{M\times N} = A_{M\times K} \times B_{K\times N}$, where A, B, and C denote the sparse matrix, input dense matrix, and output dense matrix, respectively. Let NNZ be the number of non-zero elements in A. With the increasing adoption of GPUs, researchers have explored various techniques to optimize SpMM on GPUs, targeting both CUDA cores and TCs.

The widely-used NVIDIA cuSPARSE library [37] provides high-performance CUDA-core SpMM kernels and supports Compressed Sparse Row (CSR) and Coordinate (COO) formats. Yang *et al.* [62] introduced row-split and merge-path algorithms to hide global memory latency. Hong *et al.* proposed RS-SpMM [17] and ASpT [18] which use adaptive tiling to partition sparse matrices into dense and sparse parts to utilize the shared memory. Some works focus on optimizing SpMM for DL workloads. Hidayetouglu *et al.* [15, 16], champions of the 2020 SpDNN challenge [29], employed register and shared memory tiling to enhance global memory efficiency. Their kernel is fused with the ReLU function to enhance the inference performance. Gale *et al.* proposed Sputnik [11], which introduces 1-Dimensional Tiling Scheme and reverse offset memory alignment to efficiently handle sparse data structures. Ye *et al.* proposed SparseTIR [63], which introduces a sparse compilation method with a composable format and performance-tuning system. Huang *et al.* introduced GE-SpMM [22], aiming at optimizing the memory efficiency of SpMM for GNNs by reusing sparse data. Fan *et al.* presented HP-SpMM [8], which employs a hybrid-parallel strategy to address the load imbalance issue in GNNs, particularly in graph-sampling training modes. Our DTC-SpMM is orthogonal to the CUDA-core-based works mentioned above, as we focus on systematic optimization for integrating Tensor Cores. Nevertheless, DTC-SpMM derives valuable insights from the techniques and optimization principles established in the aforementioned works.

Using TCs to accelerate SpMM has gained significant traction in recent years. While some studies have focused on TC-based Sparse GEneral Matrix-Matrix multiplication (SpGEMM) where all input and output matrices are sparse, they cannot be applied to SpMM due to different computation patterns [54, 65]. For TC-based SpMM, Gray *et al.* [12] developed a new *block-sparse* routine that allows exploiting TC for nonzero sub-matrices. cuSPARSE implements this method based on the Blocked-Ellpack (BELL) format, known as Block-SpMM [44]. Chen *et al.* proposed VectorSparse [4] which is more fine-grained compared with block sparsity for Volta architecture under FP16 precision. Castro *et al.* proposed CLASP [3], a column-vector pruning-aware implementation, which extends VectorSparse to Ampere architecture. Li *et al.* presented Magicube [36], which integrates a novel compression format called SR-BCRS and several crucial online optimizations. Although significant progress has been made, these methods take advantage of the pruning techniques and the generated regular sparse matrices with constraints on the distribution of non-zero elements. Thus, they struggle to accelerate general SpMM with irregular input sparse matrices (unstructured) like those in SC and GNNs.

Sun *et al.* [49] and Dun *et al.* [7] split sparse weight matrices into two segments: dense and sparse. They employed a *block-sparse* routine to process dense parts with TCs and CUDA cores for sparse segments, respectively. Our approach is orthogonal to theirs and can enhance the performance of their dense parts segment.

Zheng *et al.* introduced SparTA [66], which targets unstructured weight sparsity in DNN models. SparTA is the first to partition matrices into 2:4 structured sparse and unstructured sparse components to harness both sparse TCs and CUDA cores. Xia *et al.* presented Flash-LLM [59] targeting unstructured weight sparsity in the KV cache stage of Large Language Model inference. Flash-LLM introduces the Load-as-Sparse-Compute-as-Dense mode and employs double buffering to overlap the reading of dense feature matrices and TC computations. Our DTC-SpMM, in contrast, incorporates sparse double buffering to overlap sparse tile loading with computation. Given the design of reducing memory footprint rather than computation, Flash-LLM performs notably well on low sparsity (60%-90%) and memory-bounded tall-and-skinny SpMM (with N = 8, 16, and 32, etc.). Both SparTA and Flash-LLM demonstrate effectiveness in handling smaller weight matrices typical in DL, with thousands to tens of thousands of rows and 60%-90% sparsity. Nevertheless, our focus in this study is on GNN and SC matrices, which are notably larger, characterized by millions of rows and over 95% sparsity. This poses challenges for both SparTA and Flash-LLM.

Wang *et al.* proposed TC-GNN [57], the first work to accelerate irregular GNN computation with TF32 TC. They focused on optimizing general SpMM (key in GNN workload). As the state-of-the-art TC-based general SpMM implementation, TC-GNN has been integrated with SparseTIR [63]. We refer to the SpMM from TC-GNN [57] as TCGNN-SpMM.

2.3 Overview of TCGNN-SpMM

Given the novel techniques it incorporates, understanding TCGNN-SpMM's original design is crucial.

Sparse Graph Translation and the storage format. Sparse Graph Translation (SGT) employed in TC-GNN identifies non-zero tiles and consolidates non-zero elements from those tiles into a reduced number of "dense" tiles, known as TC blocks. SGT transforms the irregular sparse matrices into condensed ones so that they can be efficiently processed on TCs. TC blocks are designed to have the same shape as the matrix *fragment* using the C-level *WMMA* API (i.e., 16×8). As shown in Figure 1, the sparse matrix is first divided into row windows and compressed towards the "left side". This

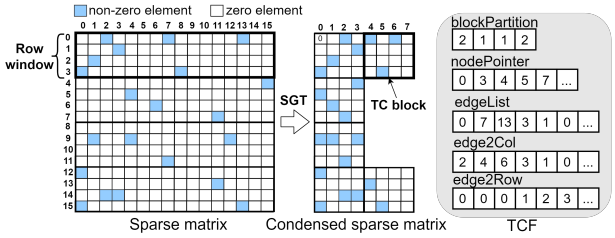

Figure 1. Sparse Graph Translation and TCF format.

process assigns a new compressed column index to each non-zero element. A specialized format is devised to store the condensed sparse matrix. We call it TC-GNN-Compressed-Format (TCF). In total, five arrays are required: *blockpartition*, *nodePointer*, *edgeList*, *edgeToColumn*, and *edgeToRow*. *blockpartition* records the number of TC blocks in each row window. *nodePointer* denotes the starting index of each row, analogous to the *RowOffset* in CSR. *edgeList* and *edgeToColumn* store the original and compressed column indices of the non-zero elements, respectively. *edgeToRow* contains the row indices of the non-zero elements.

Parallel strategy and algorithm routine Figure 2 illustrates the TCGNN-SpMM scheme. TCGNN-SpMM operates with a condensed sparse matrix. Independent thread blocks are allocated to compute each output row window of matrix *C*. Each thread block follows an iterative approach, where it processes one TC block from the corresponding row window of matrix *A* in each iteration. The iterative process comprises three main steps: ❶*FetchSparse*, ❷*ScatterFetchDense*, and ❸*TCCompute*. The threads within a thread block initially

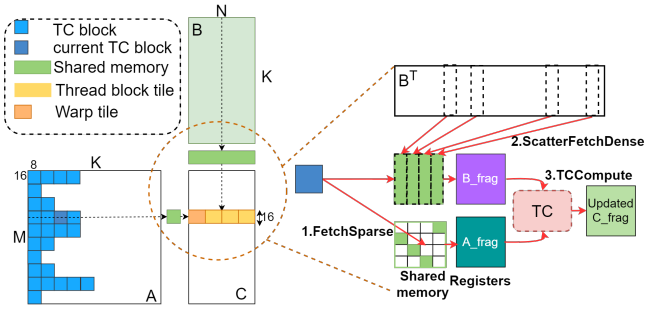

Figure 2. Overview of the TCGNN-SpMM design.

fetch the row indices and compressed column indices of the non-zero elements within a TC block (Figure 1). They then reconstruct the sparse tile in shared memory and collaborate to retrieve data from scattered positions of matrix *B*, corresponding to the original column indices of this TC block as it is compressed. The fetched data is firstly stored in shared memory and then transferred to the register *fragment* for TC computation. The warps within a thread block then call the C-level *WMMA* APIs to leverage TCs for computation and accumulation of partial results.

Table 1. 8 representative matrices. 1: from TC-GNN [57]; 2: from SNAP [34]; 3: from OGB [20]; 4: from DGL [53].

Type	Name	Abbr.	M&K	NNZ	AvgRowL
I	YeastH[1]	YH	3,138,114	6,487,230	2.07
	OVCAR-8H[1]	OH	1,889,542	3,946,402	2.09
	Yeast[1]	Yt	1,710,902	3,636,546	2.13
	DD[1]	DD	334,925	1,686,092	5.03
	web-BerkStan[2]	WB	685,230	7,600,595	11.09
II	reddit[4]	reddit	232,965	114,848,857	492.99
	ddi[3]	ddi	4267	2,140,089	501.54
	protein[3]	protein	132,534	79,255,038	598.00

3 Gaps and Opportunities

This section describes our methods for conducting a performance analysis on the current state-of-the-art TC-based general SpMM routine and highlights the observed performance gaps. We first expand the dataset used in [57] based on the following observations: while the number of rows in the original matrices spans a wide range, reaching millions, the number of non-zero elements is only in the millions. Thus the average row length (*AvgRowL*) is relatively small, ranging from 2 to 12. However, *AvgRowL* significantly affects the SpMM efficiency [62]. Besides, the total floating-point operation is $2 \times N \times NNZ$, which depends on the number of non-zero elements. Therefore, we supplement several widely-used benchmark matrices to extend non-zero elements to the billions and expand *AvgRowL* to nearly 600.

The matrices in Table 1 represent the typical workload of GNNs as they serve as benchmarking matrices in GNN-related works [10, 21–23, 53, 55–57]. The matrices are categorized into two types for analysis: **Type I**, which has a small *AvgRowL*, and **Type II**, which has a large *AvgRowL*.

We conduct an in-depth analysis covering both algorithmic and micro-architectural aspects. Our tests are performed on RTX4090 (Ada Lovelace) and RTX3090 (Ampere) with CUDA 12.1. We show the results of RTX4090, a newer generation GPU compared to RTX3090.

3.1 Observed performance gaps

We have identified four key performance gaps in the TC-based SpMM routine, which can be attributed to the complexities in incorporating TCs into SpMM.

Observation 1: High memory consumption due to the inefficient storage format. SpMM can be implemented with various storage formats that may result in significantly different performance, particularly due to memory complexity. As a memory-bound kernel, the theoretical performance upper-bound of SpMM is mostly determined by memory access efficiency [18]. Under the constraint of a fixed amount of total floating-point operations, storage formats with lower memory complexity imply higher computational density and higher roofline performance upper-bound.

The TCF format in TC-GNN requires $\lceil M/16 \rceil + M + 1 + NNZ \times 3$ elements to represent matrix *A*, comprising $\lceil M/16 \rceil$ for *blockpartition*, $M+1$ for *nodePointer*, and NNZ each for

edgeList, *edgeToColumn*, and *edgeToRow*. In comparison, CSR requires $M+1+NNZ$ elements [17]. For the 8 matrices, TCF, on average, consumes 168.41% more memory space than CSR, which suggests that TCF is not memory-efficient and may impact performance.

Observation 2: Low density of TC blocks. TC-GNN introduces SGT to condense sparse matrices, reducing the number of TC blocks by half. Condensing is crucial for accelerating SpMM using TC, as it transforms unstructured sparse matrices into a relatively structured form that can fit the format required by TCs. We define an important metric to measure the degree of condensing, namely the average number of non-zero elements in TC blocks (***MeanNnzTC***). A higher value of *MeanNnzTC* indicates a higher degree of condensing. The value of *MeanNnzTC* has implications in two aspects. First, it affects the total computational workload. When using TC, the workload of SpMM is determined by the number of TC blocks (***NumTCBlocks***) instead of *NNZ*. With a fixed *NNZ*, a larger *MeanNnzTC* implies a smaller workload. Second, it impacts data reuse efficiency. A TC block (16×8) loads 8 corresponding rows from *B*. Non-zero elements in the same column within a TC block can reuse one global memory transaction. Therefore, a higher value of *MeanNnzTC* indicates a higher data reuse rate, resulting in fewer global transactions.

Table 2. Measured key indicator values for TCGNN-SpMM.

	Dataset	*MeanNnzTC*	*#IMAD/ #HMMA*	TC Pipeline Utilization
Type I	YH	9.79	13.72	4.19%
	OH	9.66	13.69	4.31%
	Yt	10.69	13.80	3.97%
	DD	12.97	13.43	6.64%
	WB	26.9	15.16	6.09%
Type II	reddit	16.53	98.54	0.46%
	ddi	25.88	46.67	0.90%
	protein	14.80	63.90	1.47%

Table 2 shows the measured *MeanNnzTC* values after applying SGT. For most matrices, *MeanNnzTC* remains below 16. A *MeanNnzTC* value less than 16 indicates that, on average, each column in the TC blocks contains fewer than two non-zero elements, leading to limited data reuse benefits. The statistical outcomes suggest that SGT has not achieved the expected level of condensation.

Observation 3: Low Tensor Core pipeline utilization. TC pipeline utilization is crucial in assessing the effectiveness of TCs [60]. We employed the NVIDIA Nsight Compute (NCU) kernel profiler to obtain the utilization. The results in Table 2 show that the pipeline utilization consistently remains below 8%.

The significantly low utilization can be attributed to two main factors. Firstly, there is redundancy in shared memory utilization. Similar to TC-based GEMM [51], TCGNN-SpMM loads data tiles from matrices *A* and *B* into shared memory before transferring them to the register file. In TC-based GEMM, this enables data reuse in both the row-wise (for matrix *A*) and column-wise direction (for matrix *B*), thereby reducing the number of global memory transactions. However, due to the irregular and compressed nature of sparse *A*, column-wise reuse (i.e., reusing tiles of matrix *B* through shared memory) cannot be achieved. Secondly, there is a huge amount of coordinate calculations involved. During the *FetchSparse* and *ScatterFetchDense* stages, each thread performs extensive computations to determine the memory access indices. To quantify this, we measured the number of executed instructions. Table 2 shows that TCGNN-SpMM executes over 10× more Integer Multiply-Add (*IMAD*) instructions (for coordinate computations) than *HMMA* instructions (for TC computations).

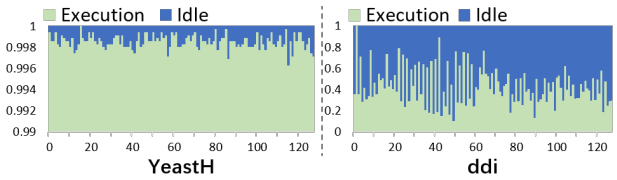

Figure 3. The relative execution and idle time of all 128 SMs on RTX4090. We show two representative matrices.

Observation 4: Input-adaptive workload imbalance. Load imbalance on GPUs refers to the uneven distribution of computation tasks among different processing units, such as SMs or warps, which leads to some units being overloaded while others remain idle [62]. Numerous studies show that load imbalance greatly impacts the performance of CUDA-core SpMM and propose various methods to mitigate this issue [6, 8, 11, 62].

In CUDA-Core SpMM, tasks are divided based on rows in the sparse matrix, where the non-zero elements serve as the fundamental unit. With the introduction of TCs, tasks are divided into row windows (16 rows), where the TC block is the unit. This change highlights the need for further in-depth research into load balancing, which is not considered in TCGNN-SpMM. To assess the load situation, we collect execution times for each SM (128 in total) on RTX4090 in TCGNN-SpMM. Figure 3 depicts the results for two representative matrices, *YeastH* and *ddi*. For *ddi*, many SMs are idle, while this phenomenon is less prominent in *YeastH*. Our statistical analysis reveals that load imbalance remains significant, and it is closely related to the input sparse matrix. Matrices with a significant disparity in the number of TC blocks within their row windows often encounter severe load imbalance.

4 Design of DTC-SpMM

4.1 Design overview

As shown in Figure 4, DTC-SpMM comprises several key components to harness the computational power of TCs

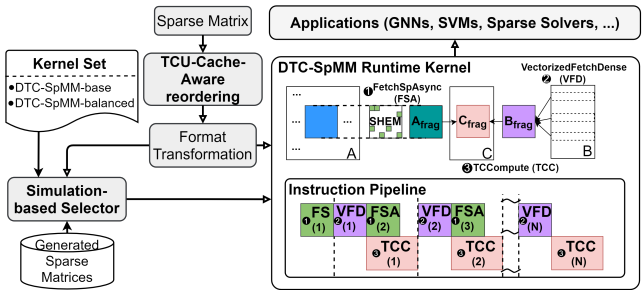

Figure 4. The design overview of DTC-SpMM.

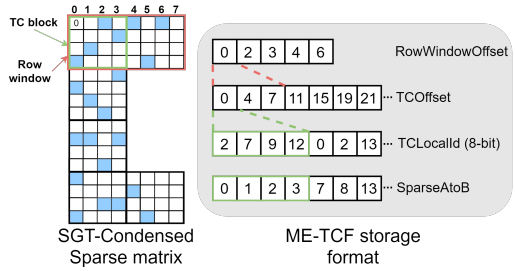

Figure 5. The design of Memory-efficient-TCF (ME-TCF).

effectively in SpMM. Firstly, DTC-SpMM introduces an offline two-level ❶ TCU-Cache-Aware reordering module to rearrange the input sparse matrix, enhancing the density of TC blocks and improving data locality. Secondly, the ❷ format conversion module efficiently transforms the reordered sparse matrix into the memory-efficient ME-TCF format designed specifically for TCs. DTC-SpMM also includes a ❸ simulation-based Selector to automatically select the runtime kernel for the current input matrix, including *DTC-SpMM-base* and *DTC-SpMM-balanced*, addressing input-adaptive workload imbalance issues. Finally, DTC-SpMM incorporates a highly optimized ❹ runtime SpMM kernel. A series of techniques, including shared-memory bypassing and sparse double buffering, are designed to optimize memory access and increase the TC pipeline utilization. Modules ❶, ❷, and ❸ are accelerated by highly parallel CUDA kernels to reduce overhead, and detailed discussions are provided in Section 6.

4.2 Memory-efficient storage format

As the foundation of DTC-SpMM, we design a memory-efficient format for TC-based SpMM routine based on TC-GNN's SGT, which we call ME-TCF. ME-TCF utilizes a total of four arrays to represent an SGT-condensed sparse matrix (Figure 5). ❶*RowWindowOffset* indicates the starting index of each row window in the *TCOffset* array, consisting of $\lceil M/16 \rceil + 1$ elements. ❷*TCOffset* holds the starting index of each TC block in the *TCLocalId* array, consisting of $NumTCBlock + 1$ elements. ❸ *TCLocalId* stores the local index of each nonzero element in all TC blocks. We design this array to utilize low-precision (8-bit) integers instead of 32-bit integers, capitalizing on the small-size nature of TC blocks (16×8). The largest local index (127) will remain within the range of an unsigned 8-bit integer (255). This array requires $NNZ/4$ elements. ❹*SparseAtoB* holds the original column indices of each column in TC blocks, necessitating $NumTCBlock \times 8$ elements. In total, ME-TCF requires $\lceil M/16 \rceil + NumTCBlock \times 9 + NNZ/4 + 2$ elements to represent A. Compared to the three 32-bit integer arrays in TCF, we utilize a single 8-bit array to store the indices of non-zero elements, significantly reducing the storage overhead.

4.3 TCU-Cache-Aware reordering

As discussed in Obs.2, SGT's effectiveness heavily depends on the original distribution of nonzero elements in the sparse matrix, which limits its condensing ability.

Reordering is a promising approach to enhance the performance of CUDA-Core SpMM. Existing reordering techniques, such as METIS [28] and Louvain [46], are designed to improve cache behavior and are not specifically optimized for computing SpMM on TCs, thereby limiting their effectiveness. To address this issue, we propose a novel two-level hierarchical reordering method called TCU-Cache-Aware (TCA) reordering. Alg.1 shows the TCA algorithm, and Figure 6 provides an example. The first hierarchy groups similar rows to increase the density of TC blocks, and the second hierarchy performs reclustering of row clusters to enhance cache performance.

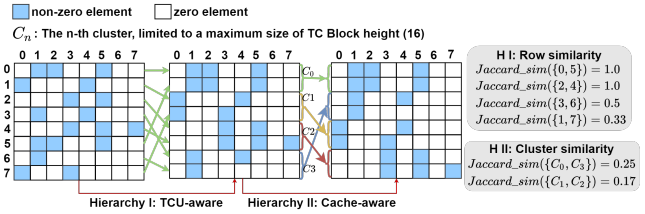

Figure 6. The design of TCU-Cache-Aware reordering. Hierarchy I groups rows into clusters that match the size of TC blocks to improve density. Hierarchy II further groups clusters to enhance L2 cache locality.

Hierarchy I (TCU-Aware). Inspired by [23, 26], we leverage the Jaccard index to measure the similarity between two rows based on their non-zero elements' column indices. We first identify row pairs with high Jaccard similarity with Locality-Sensitive Hashing (LSH) (line 2). The candidate row pairs are first organized in a priority queue (line 3). During pair merging, the row pair with the highest similarity is dequeued (line 6), and their clusters are merged (line 8). Row pair merging continues until the queue becomes empty. To maximize the density of TC blocks, we enforce a small cluster size limit of 16 (BLOCK_HEIGHT), as it matches the TC block size (16×8). A larger cluster size limit (e.g., 64

Algorithm 1 TCU-Cache-Aware (TCA) reordering

Input: Sparse Matrix A, BLOCK_HEIGHT, SM_NUM
Output: Sparse Matrix $A_{reordered}$
1: // Hierarchy I: group rows into clusters.
2: row_pairs = MinhashLSH(A); ▷ Row pairs with high Jaccard similarity.
3: Q = MakePriorityQueue(row_pairs);
4: RC_pool = InitCluster(A); RC_set = {};
5: **while** !$Q.empty()$ **do**
6: $row_i, row_j = Q.get()$; ▷ Dequeue the most similar row pair.
7: C_i, C_j = FindCluster(row_i, row_j); $RC\_pool.remove(C_i, C_j)$;
8: C = Merge(C_i, C_j); ▷ Merge the corresponding two row clusters.
9: **if** $C.size() < BLOCK\_HEIGHT$ **then**
10: $RC\_pool.add(C)$;
11: **else**
12: $RC\_set.add(C)$; ▷ Limit the row cluster size to BLOCK_HEIGHT.
13: **end if**
14: **end while**
15: // Hierarchy II: regroup into cluster of clusters.
16: $cluster\_pairs$ = MinhashLSH(RC_set); ▷ Candidate cluster pairs.
17: Q_c = MakePriorityQueue($cluster\_pairs$);
18: CC_pool = InitCluster(RC_set); CC_set = {};
19: **while** !$Q_c.empty()$ **do**
20: $C_i, C_j = Q_c.get()$; ▷ Dequeue the most similar cluster pair.
21: CC_i, CC_j = FindCluster(C_i, C_j); $CC\_pool.remove(CC_i, CC_j)$;
22: CC = Merge(CC_i, CC_j); ▷ Merge the two cluster of clusters.
23: **if** $CC.size() < SM\_NUM$ **then**
24: $CC\_pool.add(CC)$;
25: **else**
26: $CC\_set.add(CC)$; ▷ Limit the cluster size to SM_NUM.
27: **end if**
28: **end while**
29: $A_{reordered} = Reid(A, CC\_set)$

Algorithm 2 DTC-SpMM runtime kernel pseudo code

Input: Sparse Matrix A, Matrix B
Output: Matrix C
1: $RowWindowId = blockIdx.x$;
2: $Start = A.RowWindowOffset[RowWindowId]$;
3: $End = A.RowWindowOffset[RowWindowId + 1]$;
4: __shared__ $ATile[2][16 \times 8]$; ▷ Sparse buffers in SHEM.
5: __shared__ $AtoBTile[2][8]$; ▷ Index buffers in SHEM.
6: // Pre loop.
7: FetchSp($A, ATile, AtoBTile, 0$);
8: // Main loop.
9: **for** $i \leftarrow Start + 1$ **to** End **step** 1 **do**
10: // Vectorized fetch current B data to registers.
11: B_{frag} = VFetchDense($B, AtoBTile, i - 1$);
12: // Prefetch next A tile to SHEM with *cp.async*.
13: FetchSpAsync($A, ATile, AtoBTile, i$);
14: // Transfer current A tile from SHEM to register.
15: A_{frag} = ATlieToAReg($ATile, i - 1$);
16: // Tensor Core computation with *mma*.
17: C_{frag} = TCCompute($A_{frag}, B_{frag}, C_{frag}$);
18: cp.async.wait_group(0); ▷ Asynchronous transaction barrier.
19: **end for**
20: // Epilogue loop.
21: B_{frag} = VFetchDense($B, AtoBTile, End - 1$);
22: A_{frag} = ATlieToAReg($ATile, End - 1$);
23: C_{frag} = mma($A_{frag}, B_{frag}, C_{frag}$);
24: StoreCRemapping(C, C_{frag}); ▷ Write C_{frag} to C with register remapping.

[23]) results in the grouping of low-similarity rows, thereby decreasing the density of TC blocks. If a cluster has more than 16 rows, it ceases to participate in pair merging and is preserved (lines 9-13).

Hierarchy II (Cache-Aware). However, such a small cluster size limit may adversely impact L2 cache efficiency. The GPU's L2 cache is shared among SMs, and a wave of thread blocks is concurrently executed on the SMs. As discussed in Section 2.3, a thread block is assigned to a row window (i.e., a row cluster). However, neighboring clusters may have significantly different nonzero element distributions, as exemplified by *C0* and *C1* in Figure 6. The concurrent thread blocks may access distinct positions in matrix B, resulting in a low L2 cache hit rate. To address this problem, we propose the cache-aware hierarchy to enhance the L2 cache locality. We regroup the row clusters into clusters of clusters using the same steps employed in the TCU-aware hierarchy (lines 19-28). We deduplicate the column indices of all nonzero elements within a row cluster and calculate the Jaccard similarity between row clusters with these indices. As depicted in Figure 6, *C0* and *C3* are grouped together due to their relatively high Jaccard similarity.

4.4 DTC-SpMM runtime kernel optimizations

The DTC-SpMM runtime kernel scheme is diagrammed in Figure 4, and the CUDA pseudo-code is in Alg. 2. Our DTC-SpMM kernel adopts a tiling-based approach, encompassing three primary steps during each iteration. ❶ *FetchSpAsync* asynchronously copies the sparse tile of A needed for the next iteration. ❷ *VFetchDense* vectorizes the loading of corresponding data from B into registers. ❸ *TCCompute* utilizes ptx-level *mma* instructions for Tensor Core computation. We introduce three unique optimizations to enhance execution efficiency, addressing the low utilization issue observed in TCGNN-SpMM.

4.4.1 Shared-Memory bypassing.

Due to the irregular and compressed nature, reusing data tiles of matrix B through shared memory becomes infeasible. As a result, there is redundancy in storing B tiles in shared memory, harming the TC pipeline utilization. Thus, we bypass the shared memory and directly store B tiles to the register file.

Necessity of low-level PTX programming. TCGNN-SpMM relies on the C-level *WMMA* APIs. In TCGNN-SpMM, as illustrated in Figure 7, warps execute *LDG.32* (Load Global) and *STS* (Store within Local or Shared Window) instructions to perform scattered fetch operations and store in shared memory (indicated by the gray line). Subsequently, the *wmma::load_matrix_sync* function is employed to load tiles from shared memory into register fragments. However, the *wmma::load_matrix_sync* function is designed specifically for loading tiles stored in a contiguous memory layout with regular stride values. Therefore, it does not directly support scattered fetch operations from global memory.

To bypass the use of shared memory, we use lower-level PTX programming techniques. As depicted in the orange line in Figure 7, in our proposed design, warps directly execute *LDG.128* (vectorized) instructions to retrieve data from the appropriate positions in matrix B and store it directly into the register file. This approach eliminates the need for *STS* instructions and the *wmma::load_matrix_sync* function.

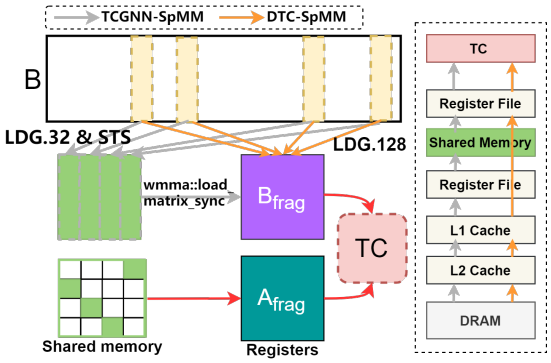

Figure 7. Instruction pipeline and data movement of TCGNN-SpMM and DTC-SpMM.

Original register distribution. Threads within a warp collectively hold operand matrices, and their distribution across the 32 threads must be managed explicitly before using the ***mma*** instruction (see Section 2.1). For TF32 precision, we use ***mma.m16n8k4***. Figure 8(a) illustrates the register distribution.

Vectorized global memory accesses and register remapping. In many workloads, such as GNNs, Matrix B is typically stored in the row-major order. However, the matrix B_{frag} required for the ***mma*** instruction is stored in the column-major order. As shown in Figure 8(b), there are two thread arrangements in the *VFetchDense* stage that can ensure the column-major distribution of elements in B_{frag}: strided-access and sequential-access.

With strided-access, threads directly access the corresponding elements in B according to the column-major distribution. In this case, neighboring threads access scattered memory addresses. On the other hand, sequential-access ensures that neighboring threads access adjacent addresses within a row, but it requires an additional warp-transpose operation (*__shfl__sync*) to exchange elements among threads to satisfy the column-major distribution [36]. In our implementation, we adopt strided-access. We perform **micro-benchmarking** [25] on RTX4090 for global memory accesses and instruction latency. Results show that both strided and sequential access achieve coalesced memory accesses [42]. The granularity of global memory access (1 sector) on RTX4090 remains 32 bytes (8 float elements). In both strided- or sequential-access, 4 sectors are required. The latency of the ***HMMA*** instruction and ***__shfl__sync*** is **16.0** and **10.7** cycles, respectively. Therefore, the online overhead introduced by warp-transpose in sequential access is significant.

To enable vectorized instructions (e.g., float4) during B fetching, we have devised a **register remapping** strategy, illustrated in Figure 8(c). Although vectorized accesses enhance bandwidth utilization, integrating them with *mma* instructions without using shared memory poses challenges.

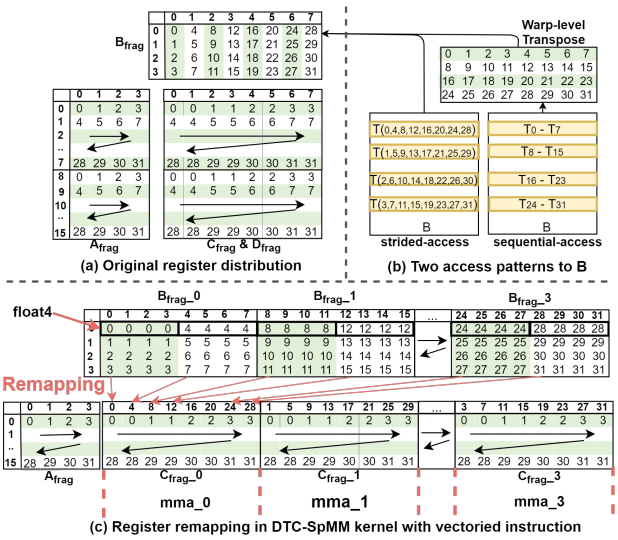

Figure 8. (a) The original register distribution of ***mma.m16n8k4***, (b) Two thread arrangements to scattered fetch B, and (c) Register remapping in DTC-SpMM kernel with vectorized memory accesses to B. 0-31 are lane IDs of threads in a warp.

For instance, in the case of float4, thread 0 would obtain four consecutive values, contrary to the original distribution depicted in Figure 8(a), where thread 0, 4, 8, and 12 hold these four consecutive values. To reconcile this distribution discrepancy without introducing online overhead, we preserve the distribution of B_{frag} and perform a one-time remapping when writing C_{frag} back to C at the end of the loop.

4.4.2 Sparse double buffering.

We devise a pipeline arrangement, as illustrated in Figure 9, to overlap memory and TC pipelines, thereby further enhancing TC pipeline utilization. Due to the absence of support for asynchronous copying from global memory to registers in GPU architectures up to Ada Lovelace, *VFetchDense* (i.e., fetching B) cannot be prefetched (due to shared-memory bypassing). Hence, we mitigate the overhead of *FetchSparse* by overlapping TC computation with the subsequent *FetchSparse* phase (i.e., loading sparse A tiles) through asynchronous prefetching. To implement this arrangement, we design two buffers in shared memory to store sparse A tiles, and we use the asynchronous ptx instruction, *cp.async* to implement *FetchSparseAsync*.

4.4.3 Index-precomputing.

We aim to reduce the number of *IMAD* instructions involved in coordinate calculations during the *FetchSparse* and *VFetchDense* stages. By precomputing the indices as much as possible in advance, we can eliminate the need for extensive computations during runtime, thus reducing the overall number of instructions executed. This technique ensures that a larger portion of the executed instructions is dedicated to *HMMA* (TC instructions), leading to increased pipeline utilization.

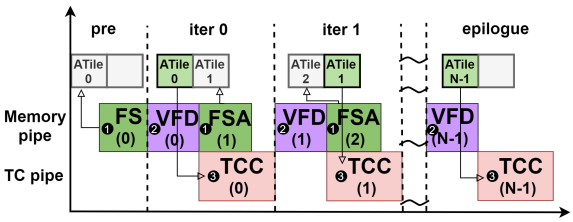

Figure 9. Pipeline arrangements with sparse double buffering. We prefetch sparse A tile in shared memory buffers and overlap *FetchSparseAsync* with TC computation.

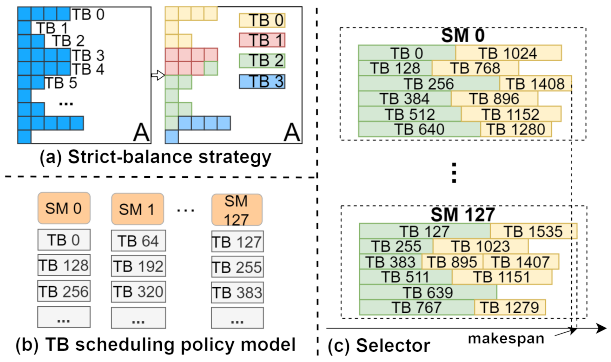

Figure 10. Diagram of (a) Strict-balance strategy, (b) thread block (TB) scheduling policy model of RTX4090, and (c) makespan calculation on one SM.

4.5 Simulation-based Selector design

As highlighted in Obs.4, load imbalance remains a significant issue. To address this challenge, we propose an input-adaptive simulation-based Selector.

4.5.1 Balanced DTC-SpMM runtime kernel. Building upon the runtime kernel discussed in Section 4.4, we devise a corresponding balanced runtime version. Our design incorporates a strict-balance strategy, which aims to achieve an evenly distributed workload among thread blocks by evenly allocating TC blocks. This strategy ensures that each SM receives a similar number of computation tasks. Figure 10 illustrates the comparison between the original workload distribution and that under our strict-balance strategy. Without our strategy, each thread block is responsible for a single row window and all the TC blocks within it, only writing the final accumulation results to the global memory. In contrast, with the strict-balance strategy, TC blocks assigned to a single thread block (32 in our implementation) can originate from different row windows, resulting in a complete balance of computational workload. However, it introduces atomic operations and additional global memory write operations, leading to online overhead.

4.5.2 Simulation-based Selector. The Selector takes into account various factors, including the properties of the input sparse matrix and the underlying hardware architecture, to determine the optimal utilization of the balanced runtime kernel. We draw insights from the traditional **Multiway Number Partitioning (MNP)** problem, which is known to be NP-hard. The MNP problem involves partitioning a set of jobs among multiple processors to minimize the **makespan**, representing the finish time of the last job.

In the context of our work, we refine this concept by characterizing the makespan as the maximum cumulative sum of TC blocks. To effectively estimate the makespan, we initiate our analysis by investigating the thread block (TB) scheduling policy of the RTX4090. As the TB scheduler employed by NVIDIA is proprietary and lacks publicly available information, the specific policy remains undisclosed and presents an intriguing area for exploration. Similar to previous studies [11, 24, 35], we employ an acknowledged scheduling policy model, which is considered to be effective in modeling NVIDIA GPUs' TB scheduling strategy, expressed as follows:

$$sm\_idx = 2\,(block\_idx \bmod 64) + \frac{block\_idx}{64} \bmod 2. \quad (1)$$

With this scheduling policy model, we can compute the makespan without executing the actual DTC-SpMM kernel. Figure 10 provides an example of makespan calculation on a single SM without the strict-balance strategy. The occupancy of the DTC-SpMM kernel on RTX4090 is 6, meaning that one SM can run 6 thread blocks concurrently. The first 6 thread blocks are scheduled on this SM, with each block's length representing the number of TC blocks in its corresponding row window. As one thread block completes its computation (e.g., block 128), the next block (e.g., block 768) is scheduled. The final makespan is determined by taking the maximum makespan among all 128 SMs. It is worth noting that the makespan with the strict-balance strategy can be calculated as $NumTCBlocks/(128 \times 6)$. The Selector is based on a crucial parameter, approximation ratio (AR), calculated as the makespan without the strict-balance strategy normalized by that with the strategy. The AR measures the improvement achieved by the strict-balance strategy in reducing the makespan. We have chosen a threshold value of 1.2 for the AR in the Selector, based on offline experimental results with 1000 generated sparse matrices. These matrices have uniformly distributed nonzeros, resulting in a naturally balanced workload. Our observations show a 22.4% performance degradation when using the strict-balance strategy. Thus, the threshold of 1.2 is set accordingly. While the chosen threshold value may not be universally optimal, we find that it works effectively with our set of matrices. For every input sparse matrix, we compute the AR; and if the calculated AR exceeds 1.2, the balanced-DTC-SpMM runtime kernel is launched.

5 Performance Evaluation

Baselines. We compare DTC-SpMM with ❶ the most widely used cuSPARSESpMM [37] from the vendor's cuSPARSE library (v12.1); The algorithm is configured at *CUSPARSE_SPMM_ALG_DEFAULT*, and the sparse format is *CUSPARSE_FORMAT_CSR*; ❷ TCGNN-SpMM [57], the state-of-the-art implementation that introduces TF32 TCs to general SpMM; ❸ Sputnik [11], a well-designed sparse library for sparsity in DL; ❹ SparseTIR [63], a recent sparse tensor compiler with TVM, targeting GNNs and sparse DNNs.

We also compare with recent TC-based SpMM implementations, including ❺ Block-SpMM from cuSPARSE (12.1) which leverages TF32 TCs and targets structured SpMM; The algorithm is configured as *CUSPARSE_SPMM_ALG_DEFAULT*, and the sparse format is *CUSPARSE_FORMAT_BLOCKED_ELL*; ❻ VectorSparse [4], which leverages fine-grained 1-D vector sparsity; ❼ Flash-LLM [59] and ❽ SparTA [66] that targets unstructured sparse weight matrices in DL.

Datasets. In addition to the 8 representative real-world matrices detailed in Table 1, our testing extends to matrices sourced from the SuiteSparse collection [32]—a widely referenced compilation of sparse matrix benchmarks derived from diverse applications. We specifically consider matrices with at least one million non-zeros. Due to Sputnik's indices computation relying on int32, certain matrices surpass the limit, leading to a segmentation fault. Furthermore, TCGNN-SpMM cannot handle non-square matrices. These matrices are consequently excluded, resulting in a final set of 414 matrices. The comprehensive information of the matrices is detailed in [19].

Environments. We test on two recent Nvidia GPU architectures: RTX4090 (Ada Lovelace with Compute Capability 8.9 and 24 GB of global memory) and RTX3090 (Ampere; 8.6; 24 GB). We use NVCC from CUDA 12.1 to compile the codes. All experiments are executed 1000 times, and the average results are reported.

5.1 Overall performance comparison

We measure the average performance with different N settings (the number of columns of matrix B), including 128, 256, and 512.

Figure 11 depicts the measured performance on RTX4090. The speedup values are normalized to cuSPARSE-SpMM, represented by the red dashed line. Our DTC-SpMM achieves the highest speedup on all 8 matrices, and the relative speedup is even higher (up to 3.29×) on Type II matrices with higher computational complexity. While TCGNN-SpMM is well-optimized for Type I matrices (with small *AvgRowL*), it fails to achieve speedups on Type II matrices characterized by long rows. Table 3 summarizes the geometric mean speedups we achieved over baseline methods across the SuiteSparse matrices. DTC-SpMM achieves significant speedups for most matrices and experiences slowdown only for a small fraction of the matrices. On RTX4090, DTC-SpMM achieves speedups over TCGNN-SpMM across all matrices, and for cuSPARSE, it achieves over 1.5× speedups on 79.5% of the matrices. Compared to SparseTIR and Sputnik, DTC-SpMM also attains speedups of 1.57× and 1.46×, respectively. On RTX3090, the results exhibit a similar trend, but overall acceleration is slightly lower than on the 4090, and DTC-SpMM experiences a slowdown on a relatively higher proportion of matrices.

Table 3. Summary of performance comparison on two GPUs. The percentage represents the portion of matrices out of the total 414 from SuiteSparse.

GPU			vs cuSPARSE	vs TCGNN	vs SparseTIR	vs Sputnik
RTX 4090	speedup	>1.5x	79.47%	79.71%	47.58%	38.65%
		1.0-1.5x	16.43%	20.29%	47.10%	58.45%
		0.9-1.0x	2.17%	0.00%	4.35%	1.21%
		0.5-0.9x	1.93%	0.00%	0.97%	1.69%
	Geomean speedup		**2.16x**	**3.25x**	**1.57x**	**1.46x**
RTX 3090	speedup	>1.5x	77.54%	91.55%	44.69%	19.81%
		1.0-1.5x	18.36%	8.45%	28.74%	68.12%
		0.9-1.0x	2.66%	0.00%	20.77%	8.70%
		0.5-0.9x	1.45%	0.00%	5.80%	3.38%
	Geomean speedup		**1.98x**	**3.25x**	**1.48x**	**1.29x**

5.2 Comparison with SpMM using TCs

Comparison with methods targeting structured sparsity. Figure 12 illustrates the speedup DTC-SpMM achieved over Block-SpMM and VectorSparse. In Block-SpMM, sparse matrices are organized in block-wise tiles and stored in the Blocked-Ellpack (BELL) format. In VectorSparse, matrices are organized in vector-wise tiles with Column Vector Sparse Encoding (CVSE). We initially transform matrices into BELL format with different block sizes (32 and 64) and CVSE format with different vector lengths (4 and 8). Across all matrices, DTC-SpMM demonstrates significant advantages (i.e., 1.14×-23.51× over Block-SpMM and 1.89×-4.95× over VectorSparse). Additionally, the necessity to pad and fill all rows of blocks in the BELL format can lead to out-of-memory (OOM) issues when applied to large-scale matrices. Thus, the state-of-the-art TC-based structured SpMM struggles to efficiently handle the highly unstructured sparse matrices in GNNs and scientific computing which we focused on.

Comparison with methods targeting unstructured weight sparsity. Table 4 shows the execution time of Flash-LLM, SparTA, and DTC-SpMM. Flash-LLM performs format conversion on matrices stored in uncompressed form (dense storage), making it prone to OOM issues when dealing with large sparse matrices like YeastH. SparTA utilizes the SpMM kernel from cuSPARSElt to leverage sparse TCs but is limited to matrices with row and column counts not exceeding 50,000. On larger datasets like *reddit* and *protein*, DTC-SpMM significantly outperforms Flash-LLM (by more than 8 times).

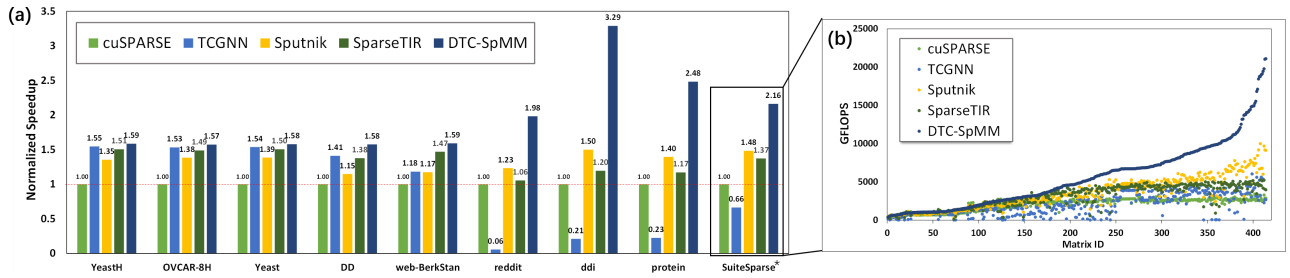

Figure 11. Performance comparison on RTX4090. (a) Speedups on 8 representative matrices (normalized to cuSPARSE-SpMM (red dashed line)). (b) Measured throughputs on 414 matrices from SuiteSparse (sorted by GFLOPS of DTC-SpMM). SuiteSparse* shows the geomean speedups across SuiteSparse matrices.

Table 4. The execution time of Flash-LLM, SparTA, and DTC-SpMM on RTX4090 with $N = 128$ (unit: ms). SparTA reports "Not Supported" and Flash-LLM reports "OOM" on other not-shown matrices.

Dataset	Flash-LLM (v1)	Flash-LLM (v2)	SparTA	Ours
ddi	0.070	0.113	**0.049**	0.068
protein	30.006	30.006	Not Supported	**3.70**
reddit	90.210	90.212	Not Supported	**5.95**

SparTA and Flash-LLM excel in handling small weight matrices typical in DL, with thousands to tens of thousands of rows and 60%-90% sparsity. However, due to the lack of condensing design leading to lower TC block density, their performance is hindered on larger and sparser matrices in GNNs and SC scenarios, characterized by millions of rows and over 95% sparsity.

5.3 Breakdown study

Effectiveness of ME-TCF storage format. As mentioned in Obs.1, the TCF format in TCGNN-SpMM stores twice the number of column coordinates of non-zero elements compared to CSR, resulting in greater storage consumption (168.41% more on average). In contrast, our designed ME-TCF format utilizes 8-bit low-precision integers to store the column coordinates of non-zero elements, effectively reducing memory complexity. For the 8 original matrices, the storage space with ME-TCF is slightly lower than that of the CSR (6.42% less on average). However, after applying the TCU-Cache-Aware reordering, the number of TC blocks decreases, resulting in reduced space requirements for *SparseAtoB* in ME-TCF, making it more memory-efficient than the CSR format (30.10% less on average). As reordering methods gain further attention in sparse computing, the memory complexity of ME-TCF is expected to decrease further, while the memory complexity of CSR remains unchanged.

Effectiveness of TCU-Cache-Aware reordering. We first evaluate the detailed performance improvement provided by TCU-Cache-Aware reordering. The results are shown

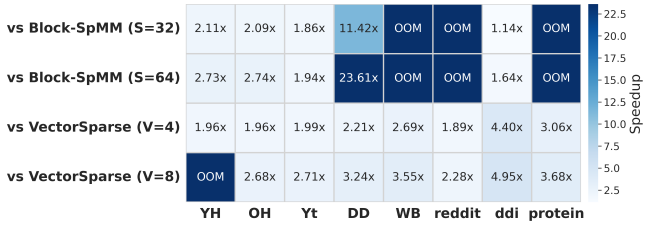

Figure 12. Speedups of DTC-SpMM over Block-SpMM and VectorSparse on RTX4090 with $N = 128$.

in Figure 13(b). With the reordered matrices, DTC-SpMM demonstrates average performance gains of 23.23%. Note that the improvement becomes more pronounced when the average row length is larger. On the other hand, both cuSPARSESpMM and DTC-SpMM benefit from TCU-Cache-Aware reordering, with DTC-SpMM achieving greater performance gains. This validates that TCU-Cache-Aware reordering is more effective and specifically designed for introducing TCs. The performance gains primarily come from the

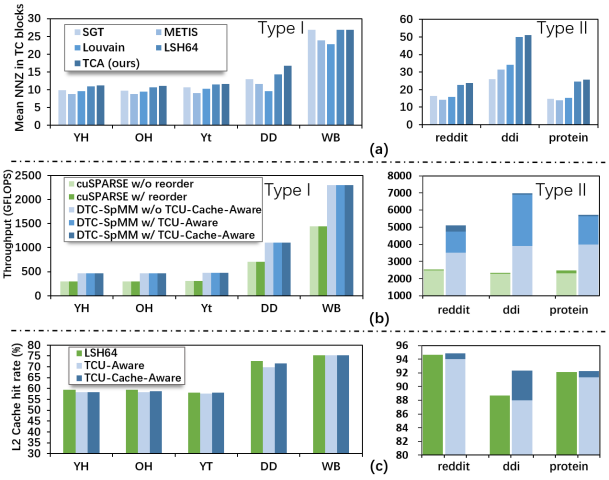

Figure 13. Comparison on (a)*MeanNnzTC*, (b) detailed breakdown throughput improvement, and (c) L2 cache hit rate.

increased density of TC blocks (*MeanNnzTC*). We compare our approach with three commonly used reordering methods, including graph-partition-based METIS [28], modularity-based Louvain [46], and LSH64 [23]. Figure 13(a) presents the changes in *MeanNnzTC*. Compared to SGT from TC-GNN, our approach largely improves *MeanNnzTC* (1.13× and 1.72× for Type I and II, respectively), surpassing the improvements brought by METIS, Louvain, and LSH64. Figure 13(c) shows the L2 cache hit rate improvement brought by our two-hierarchy design. Only with TCU-Aware hierarchy, the hit rate is lower than LSH64 (1.36% lower on average). Cache-Aware further improves L2 cache hit rate and surpasses LSH64 (0.01% higher on average).

Effectiveness of runtime kernel optimizations. In Figure 14, the TC pipeline utilization and the ratio of executed *IMAD* to *HMMA* instructions (*#IMAD/#HMMA*) are illustrated. For Type I and II matrices, DTC-SpMM demonstrates TC pipeline utilization 11.35% and 1760.25% higher than TCGNN-SpMM, respectively, while exhibiting a *#IMAD/#HMMA* ratio 38.39% and 89.37% lower, respectively. Ablation studies reveal that basic DTC-SpMM with only the ME-TCF format achieves 6.10% and 921.24% higher TC pipeline utilization than TCGNN-SpMM. Further ablation experiments demonstrate that SMB increases pipeline utilization by 12.16%, IP is effective, particularly for matrices with large average row lengths, SDB enhances pipeline utilization by 4.83%, and VFD further boosts utilization by 4.99%. Consequently, the detailed ablation study confirms the effectiveness of individual runtime kernel optimizations as well as their combined impact.

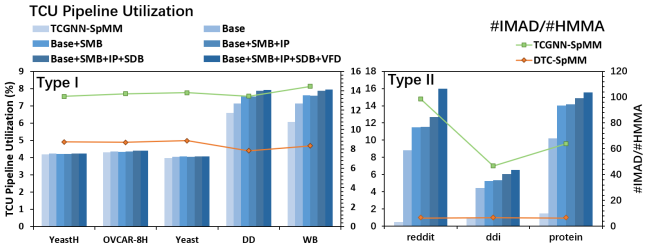

Figure 14. Comparison on TC pipeline utilization and executed instructions. Base: Basic DTC-SpMM with ME-TCF format and no optimizations; SMB: Shared-memory-bypassing; IP: index-precomputing; SDB: sparse double buffering; VFD: VectorizedFetchDense.

Effectiveness of workload balance design. Figure 15(b) illustrates the workload distribution among 128 SMs in RTX4090 of *DTC-SpMM-base* (i.e., without balancing design) and *DTC-SpMM-balanced* on *reddit* and *ddi*. It can be observed that the strict-balance strategy effectively mitigates the previously severe workload imbalance. Furthermore, when applying the strict-balance strategy, DTC-SpMM demonstrates remarkable improvements of 15.82% and 54.31% on *reddit* and *ddi* respectively, as shown in Figure 15. Hence, the strict-balance strategy proves to be beneficial for addressing heavily imbalanced workloads.

For Type I matrices, represented by *YeastH*, where load imbalance is less pronounced due to limited TC blocks in a row window resulting from small row length, the benefits of strict-balance are not as prominent. Our proposed Selector can accurately differentiate between these scenarios and select appropriate load distribution strategies for different input matrices.

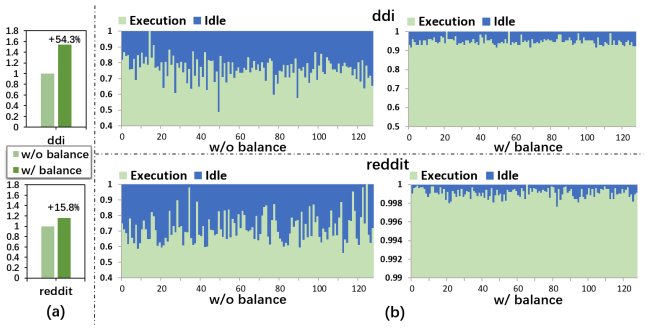

Figure 15. Effectiveness of workload balancing design. (a) Enhancement in throughput due to the balancing design on the *reddit* and *ddi* datasets. (b) Changes in workload distribution with and without workload balancing design. We collect the relative execution and idle time of all 128 SMs on RTX4090.

5.4 Case study: End-to-end GNN training

The Graph Convolutional Neural Network (GCN) model [31] is one of the most widely used GNN models, consisting of several GraphConv layers. The GraphConv layer performs the following computation:

$$H_{l+1} = \sigma\left[(A \times H_l) \times w_l + b_l\right], \tag{2}$$

where σ represents the activation function; A denotes the adjacency matrix; H stands for the feature matrix; w indicates the weight matrix; and b represents the bias term. The operation $A \times H$ corresponds to a typical SpMM operation.

Leveraging PyTorch [45] and its CUDA Extension feature, we implement a dual-layer GCN model using DTC-SpMM, denoted as DTC-GCN. We evaluate the end-to-end training time of the model across four graph datasets. In addition to *YeastH* and *protein*, which are already used for analysis in previous sections, we expand our evaluation to include the Illinois Graph Benchmark (IGB) dataset [30]. IGB is a recently released and valuable tool for GNN research, offering a variety of graph structures, including both homogeneous and heterogeneous graphs, at different scales. We specifically choose the homogeneous graph datasets *IGB-tiny* and *IGB-small* [30]. We compare against three popular GNN training frameworks, Deep Graph Library (DGL) [53], Pytorch-Geometric (PyG) [10] and TC-GNN [57]. For

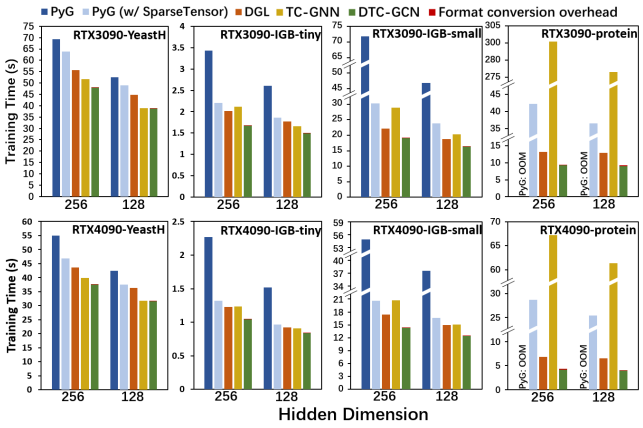

Figure 16. End-to-end training time of GCN models (200 epochs). We excluded the format conversion time for TC-GNN, as it does not utilize GPU acceleration, resulting in relatively higher times.

PyG versions above 1.6.0, GNN training can be performed in two modes: "Gather-Scatter" and "SparseTensor". The "SparseTensor" mode leverages SpMM kernels from the torch-sparse library, resulting in lower memory footprint and faster execution times. The hidden layer dimensions are configured at 128 and 256. Figure 16 shows the training time of DTC-GCN, which includes the format conversion time. On RTX4090, DTC-GCN achieves 1.26×, 1.91× and 2.21× geometric mean speedups over DGL, PyG (w/ SparseTensor) and TC-GNN, respectively. The geometric mean speedups are 1.22×, 1.81×, and 2.69× on RTX3090. When excluding the results on the *protein* dataset (due to the weak performance of TC-GNN's SpMM kernel), DTC-GCN achieves geometric mean speedups of 1.16× and 1.19× over TC-GNN on RTX4090 and RTX3090, respectively.

6 Overhead and Limitation

We assess the potential overheads in the design of DTC-SpMM using two representative matrices (*YeastH* and *protein*) as examples. ❶ **Format Conversion Overhead:** Converting matrices to ME-TCF from CSR format may introduce overhead. To mitigate this, we develop GPU kernels to accelerate this process. The conversion overhead is 1.48× and 14.50× of a single SpMM execution ($N = 128$), respectively. Using highly-optimized GPU kernels, our format conversion outperforms TC-GNN (which lacks GPU acceleration) by 101.00× and 72.21×, respectively. ❷ **Reordering Overhead (Optional):** TCU-Cache-Aware reordering, based on [23, 26], achieves significant improvements but involves computationally expensive Minhash and Jaccard calculations. We compute them with GPU kernels from MinHashCuda and [9], along with a batching technique to enhance GPU utilization. These efforts reduce reordering time from hours [23] to minutes. Reordering can be an offline preprocessing step, and DTC-SpMM achieves significant speedups even without it (Figure 11). ❸ **Selector Overhead:** The Selector execution time accounts for 42.0% and 24.8% of a single SpMM execution time, respectively.

Many real-world applications require iterative SpMM execution [5, 14, 58], where the sparse matrix A remains unchanged for thousands of SpMM operations. When applied to these scenarios, both the format conversion and Selector overhead of DTC-SpMM are negligible. Besides, for sparse computing libraries (e.g., DGL [53] and PyG [10]) and sparse matrix collections (e.g., IGB [30], OGB [20], SNAP [34], and SuiteSparse [32]), all three overhead sources can be effectively mitigated, enabling DTC-SpMM to achieve good acceleration. These libraries can perform reordering and format conversion once on the stored sparse matrices, providing significant performance benefits to numerous applications built on them, such as sparse matrix factorization [27] and GNN training and hyper-parameter tuning [47, 64]. However, due to format conversion, DTC-SpMM may not be suitable for a small number of scenarios with varying input sparse matrices in each SpMM execution (e.g., graph sampling in GNN [13]). Systems with lighter overhead, like cuSPARSE [37] and HP-SpMM [8], are more suitable for such cases.

7 Conclusion

In this paper, we have examined the state-of-the-art techniques to optimize general SpMM with TCs and identified four key performance gaps. To close these gaps, we have proposed DTC-SpMM, a novel approach with systematic optimizations tailored to harness TCs for accelerating general SpMM. These optimizations encompass both high-level algorithm design and low-level instruction pipeline and memory access optimizations. Extensive experiments on modern GPUs have demonstrated that DTC-SpMM achieves remarkable speedups compared to both state-of-the-art Tensor-Core SpMM and the widely-used CUDA-core SpMM. While initially designed for NVIDIA GPUs and targeting TF32 precision, our insights and optimizations can be extended to support other precisions, facilitating general SpMM acceleration on parallel devices equipped with matrix computing units.

Acknowledgments

We extend our thanks to the anonymous reviewers and our shepherd, Vikram Sharma Mailthody, for their valuable feedback and support. This work was partially supported by National Natural Science Foundation of China under Grant No. 62272122, a Hong Kong RIF grant under Grant No. R6021-20, and Hong Kong CRF grants under Grant No. C2004-21G and C7004-22G.

References

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)*, 54(9):1–38, 2021.

[2] AMD. AOCL: AMD optimizing CPU libraries, 2020. Accessed on July 26, 2023.

[3] Roberto L Castro, Diego Andrade, and Basilio B Fraguela. Probing the efficacy of hardware-aware weight pruning to optimize the spmm routine on ampere gpus. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 135–147, 2022.

[4] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. Efficient tensor core-based gpu kernels for structured sparsity under reduced precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[5] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[6] Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. Heuristic adaptability to input dynamics for spmm on gpus. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 595–600, 2022.

[7] Ming Dun, Xu Zhang, Huawei Cao, Yuan Zhang, Junying Huang, and Xiaochun Ye. Adaptive sparse deep neural network inference on resource-constrained cost-efficient gpus. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2023.

[8] Ruibo Fan, Wei Wang, and Xiaowen Chu. Fast sparse gpu kernels for accelerated training of graph neural networks. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 501–511. IEEE, 2023.

[9] Alex Fender, Brad Rees, and Joe Eaton. Rapids cugraph. In *Massive Graph Analytics*, pages 483–493. Chapman and Hall/CRC, 2022.

[10] Matthias Fey and Jan Eric Lenssen. Fast Graph Representation Learning with PyTorch Geometric, May 2019.

[11] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[12] Scott Gray, Alec Radford, and Diederik P Kingma. Block-sparse gpu kernels, 2017.

[13] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[14] Mohammed Heyouni and Azeddine Essai. Matrix krylov subspace methods for linear systems with multiple right-hand sides. *Numerical Algorithms*, 40:137–156, 2005.

[15] Mert Hidayetoğlu, Carl Pearson, Vikram Sharma Mailthody, Eiman Ebrahimi, Jinjun Xiong, Rakesh Nagi, and Wen-mei Hwu. At-scale sparse deep neural network inference with efficient gpu implementation. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2020.

[16] Mert Hidayetoglu, Carl Pearson, Vikram Sharma Mailthody, Eiman Ebrahimi, Jinjun Xiong, Rakesh Nagi, and Wen-mei W Hwu. Efficient inference on gpus for the sparse deep neural network graph challenge 2020. *CoRR*, 2020.

[17] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V Çatalyürek, Srinivasan Parthasarathy, and P Sadayappan. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 66–79, 2018.

[18] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019.

[19] HPMLL. DTC-SpMM_ASPLOS24. https://github.com/HPMLL/DTC-SpMM_ASPLOS24.git, 2024.

[20] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020.

[21] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. Featgraph: A flexible and efficient backend for graph neural network systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13. IEEE, 2020.

[22] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. Gespmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2020.

[23] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. Understanding and bridging the gaps in current gnn performance optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–132, 2021.

[24] Muhammad Huzaifa, Johnathan Alsop, Abdulrahman Mahmoud, Giordano Salvador, Matthew D Sinclair, and Sarita V Adve. Inter-kernel reuse-aware thread block scheduling. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(3):1–27, 2020.

[25] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[26] Peng Jiang, Changwan Hong, and Gagan Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 376–388, 2020.

[27] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. A high-performance parallel algorithm for nonnegative matrix factorization. *ACM SIGPLAN Notices*, 51(8):1–11, 2016.

[28] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

[29] Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Ryan Robinett, and Sid Samsi. Sparse deep neural network graph challenge. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[30] Arpandeep Khatua, Vikram Sharma Mailthody, Bhagyashree Taleka, Tengfei Ma, Xiang Song, and Wen-mei Hwu. Igb: Addressing the gaps in labeling, features, heterogeneity, and size of public graph datasets for deep learning research. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4284–4295, 2023.

[31] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[32] Scott P Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019.

[33] Mariia Krainiuk, Mehdi Goli, and Vincent R Pascuzzi. oneapi open-source math library interface. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 22–32. IEEE, 2021.

[34] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1–20, 2016.

[35] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. Locality-aware cta clustering for modern gpus. *ACM SIGARCH Computer Architecture News*, 45(1):297–311, 2017.

[36] Shigang Li, Kazuki Osawa, and Torsten Hoefler. Efficient quantized sparse matrix operations on tensor cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[37] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.

[38] NVIDIA. NVIDIA volta gpu architecture whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017. Accessed on July 27, 2023.

[39] NVIDIA. NVIDIA ampere GA102 GPU architecture whitepaper. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020. Accessed on July 27, 2023.

[40] NVIDIA. NVIDIA ampere GA102 GPU architecture whitepaper. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, 2020. Accessed on July 27, 2023.

[41] NVIDIA. NVIDIA ada gpu architecture whitepaper. https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf, 2023. Accessed on July 27, 2023.

[42] NVIDIA. NVIDIA CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2023. Accessed on July 5, 2023.

[43] NVIDIA. PTX ISA: CUDA Toolkit documentation, 2023. Accessed on July 27, 2023.

[44] NVIDIA. Accelerating matrix multiplication with block-sparse format and NVIDIA tensor cores. https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with-block-sparse-format-and-nvidia-tensor-cores/, Publication date not provided. Accessed on July 27, 2023.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[46] Xinyu Que, Fabio Checconi, Fabrizio Petrini, and John A Gunnels. Scalable community detection with the louvain algorithm. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 28–37. IEEE, 2015.

[47] Min Shi, Yufei Tang, Xingquan Zhu, Yu Huang, David Wilson, Yuan Zhuang, and Jianxun Liu. Genetic-gnn: Evolutionary architecture search for graph neural networks. *Knowledge-Based Systems*, 247:108752, 2022.

[48] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. Dissecting tensor cores via microbenchmarks: Latency, throughput and numeric behaviors. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):246–261, 2022.

[49] Yufei Sun, Long Zheng, Qinggang Wang, Xiangyu Ye, Yu Huang, Pengcheng Yao, Xiaofei Liao, and Hai Jin. Accelerating sparse deep neural network inference using gpu tensor cores. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2022.

[50] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Comput. Surv.*, 55(6), dec 2022.

[51] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023.

[52] F Vazquez, EM Garzon, and JJ Fernandez. A matrix approach to tomographic reconstruction and its implementation on gpus. *Journal of Structural Biology*, 170(1):146–151, 2010.

[53] Minjie Yu Wang. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*, 2019.

[54] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. Dual-side sparse tensor core. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1083–1095. IEEE, 2021.

[55] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In *15th USENIX symposium on operating systems design and implementation (OSDI 21)*, pages 515–531, 2021.

[56] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. {MGG}: Accelerating graph neural networks with {Fine-Grained}{Intra-Kernel}{Communication-Computation} pipelining on {Multi-GPU} platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 779–795, 2023.

[57] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. TC-GNN: Bridging sparse GNN computation and dense tensor cores on GPUs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 149–164, 2023.

[58] Zeyi Wen, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. Thundersvm: A fast svm library on gpus and cpus. *The Journal of Machine Learning Research*, 19(1):797–801, 2018.

[59] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. *arXiv preprint arXiv:2309.10285*, 2023.

[60] Da Yan, Wei Wang, and Xiaowen Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 634–643. IEEE, 2020.

[61] Da Yan, Wei Wang, and Xiaowen Chu. Optimizing batched winograd convolution on gpus. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 32–44, 2020.

[62] Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing*, pages 672–687. Springer, 2018.

[63] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 660–678, New York, NY, USA, 2023. Association for Computing Machinery.

[64] Yingfang Yuan, Wenjun Wang, George M Coghill, and Wei Pang. A novel genetic algorithm with hierarchical evaluation strategy for hyperparameter optimisation of graph neural networks. *arXiv preprint arXiv:2101.09300*, 2021.

[65] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. Accelerating sparse matrix–matrix multiplication with gpu tensor cores. *Computers & Electrical Engineering*, 88:106848, 2020.

[66] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. SparTA: Deep-Learning model sparsity via Tensor-with-Sparsity-Attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 213–232, 2022.