TC-GNN: Accelerating Sparse Graph Neural Network Computation Via Dense Tensor Core on GPUs

Anonymous Author(s)

ABSTRACT

Recently, graph neural networks (GNNs), as the backbone of graph-based machine learning, demonstrate great success in various domains (*e.g.*, e-commerce). However, the performance of GNN is usually unsatisfactory due to its highly sparse and irregular graph-based operations. To this end, we propose, **TC-GNN**, the first GPU Tensor Core Unit (TCU) based GNN acceleration framework. The core idea is to reconcile the "Sparse" GNN computation with "Dense" TCU. We conduct an in-depth analysis of the sparse operations in mainstream GNN computing frameworks. We introduce a novel sparse graph translation technique to facilitate TCU processing of sparse GNN workload. We also implement an effective CUDA core and TCU collaboration design to fully utilize GPU resources. We fully integrate TC-GNN with the Pytorch framework for ease of programming. Rigorous experiments show an average 1.70× speedup over the state-of-the-art Deep Graph Library framework across various mainstream GNN models and dataset settings.

1 INTRODUCTION

Over the recent years, with the increasing popularity of the graph-based learning, graph neural networks (GNNs) have become to dominate the computing of essential tasks across various domains, including the e-commerce, financial services, *etc.* Compared with standard methods for graph analytics, such as random walk [13, 35] and graph laplacians [6, 22, 23], GNNs highlight themselves with significantly higher accuracy [17, 38, 40] and better generality [14]. From the computation perspective, GNNs feature an interleaved execution phase of both graph operations (scatter-and-gather [12]) at the ***Aggregation*** phase and Neural Network (NN) operations (matrix multiplication) at the ***Update*** phase. Our experimental studies further show that the aggregation phase which involves highly sparse computation on irregular input graphs generally takes more than 80% running time for both GNN training and inference. Existing GNN frameworks, *e.g.*, Deep Graph Library [39] and Pytorch-Geometric [9], are mostly built upon the popular NN frameworks that are originally optimized for dense operations, such as general matrix-matrix multiplication (GEMM). To support sparse computations in GNNs, their common practice is to incorporate sparse primitives (such as cuSPARSE [25]) for their backend implementations. However, cuSPARSE leverages the sparse linear algebra (LA) algorithm which involves lots of high-cost indirect memory accesses on non-zero elements of a sparse matrix. Therefore, cuSPARSE cannot enjoy the same level of optimizations (*e.g.*, data reuse) as its dense counterpart, such as cuBLAS [27]. Moreover, cuSPARSE is designed to only utilize on CUDA core. Therefore, it cannot benefit from the recent technical advancement on GPU hardware features, such as Tensor Core Unit (TCU), which can significantly boost the GPU performance of dense LA algorithms (*e.g.*, the linear transformation and convolution) in most conventional deep-learning applications.

This work focuses on exploring the potentials of TCU for accelerating such GNN-based graph learning. We remark that making TCU effective for general GNN computing is a non-trivial task. Our initial study shows that naively applying the TCU to sparse GNN computation would even result in inferior performance compared with the existing sparse implementations on CUDA core. There are several challenges. First, directly resolving the sparse GNN computing problem with the pure dense GEMM solution is impractical due to the extremely large memory cost ($O(N^2)$, where N is the number of nodes). Besides, traversing the matrix tiles already known to be filled with all-zero elements would cause excessive unnecessary computations and memory access. Second, simply employing TCU to process non-zero matrix tiles of the sparse graph adjacency matrix would still waste most of the TCU computation and memory access efforts. Because TCU input matrix tiles are defined with fixed dimension settings (*e.g.*, $Height(16) \times Width(8)$), whereas the non-zero elements of a sparse graph adjacency matrix are distributed irregularly. Thus, it requires intensive zero-value padding to satisfy such a rigid input constraint. Third, even though the recent CUDA release update enables TCU to exploit the benefit of certain types of sparsity [26], it only supports blocked SpMM, where non-zero elements must be first fit into well-shaped blocks and the number of blocks must be the same across different rows. Such an input restriction makes it hard to handle highly irregular sparse graphs from the real-world GNN applications efficiently.

To this end, we introduce, **TC-GNN**, the first TCU-based GNN acceleration design on GPUs. Our key insight is to *let the input sparse graph fit the dense computation of TCU*. ***At the input level***, instead of exhaustively traversing all sparse matrix tiles and determine whether to process each tile, we develop a new sparse graph translation (SGT) technique that can effectively identify those non-zero tiles and condense non-zero elements from these tiles into a fewer numbers of "dense" tiles. Our major observation is that neighbor sharing is very common among nodes in real-world graphs. Therefore, applying SGT can effectively merge the unnecessary data loading of the shared neighbors among different nodes to avoid the high-cost memory access. Our SGT is generally applicable towards any kind of sparse pattern of input graphs and can always yield the correct results as the original sparse algorithm. ***At the kernel level***, for efficiently processing GNN sparse workloads, TC-GNN exploits the benefits of CUDA core and TCU collaboration. The major design idea is that the CUDA core which is more excel at fine-grained thread-level execution would be a good candidate for managing memory-intensive data access. While TCU which is more powerful in handling simple arithmetic operations (*e.g.*, multiplication and addition) can be well-suited for compute-intensive GEMM on dense tiles generated from SGT. ***At the framework level***, we integrate TC-GNN with the popular Pytorch [34] framework. Thereby, users only need to interact with their familiar Pytorch programming environment by using TC-GNN APIs. This can significantly reduce

SC '21, November 14–19, 2021, St. Louis, MO
2021. ACM ISBN 978-1-4503-XXXX-X/18/06.
https://doi.org/978-1-4503-XXXX-X/18/06

extra learning efforts meanwhile improving the user productivity and code portability.

To conclude, we summarize our contributions as follow,

- We conduct a detailed analysis (§3) of several existing solutions (*e.g.*, SpMM on CUDA core) and identify the potentials of using TCU for accelerating the sparse GNN workloads.
- We introduce a sparse graph translation technique (§4). It can make the sparse and irregular GNN input graphs easily fit the dense computing on TCU for acceleration.
- We build a TCU-tailored GPU kernel with effective CUDA core and TCU collaboration (§5). It consists of a novel two-level workload mapping strategy for computation optimization and a TCU-optimized dataflow design for memory access optimization.
- We deliver an end-to-end GNN framework design with seamless integration with the popular Pytorch framework for high programmability and configurability.
- Extensive experimental results show the significant speedup (average 1.70×) over the state-of-the-art GNN computing framework, Deep Graph Library, across various mainstream GNN models and dataset settings.

2 BACKGROUND AND RELATED WORK

2.1 Graph Neural Networks

Graph neural networks (GNNs) is an effective tool for graph-based machine learning. The detailed computing flow of GNNs is illustrated in Figure 1.

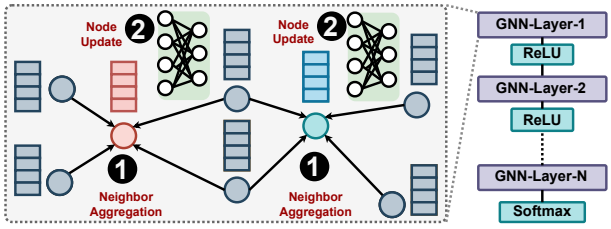

Figure 1: GNN General Computation Flow.

GNNs basically compute the node feature vector (embedding) for node v at layer $k+1$ based on the embedding information at layer k ($k \geq 0$), as shown in Equation 1,

$$\begin{aligned} a_v^{(k+1)} &= \textbf{Aggregate}^{(k+1)}(h_u^{(k)} | u \in \mathbf{N}(v) \cup h_v^{(k)}) \\ h_v^{(k+1)} &= \textbf{Update}^{(k+1)}(a_v^{(k+1)}) \end{aligned} \tag{1}$$

where $h_v^{(k)}$ is the embedding vector for node v at layer k; $a_v^{(k+1)}$ is the aggregation results through collecting neighbors' information (*e.g.*, node embeddings); $\mathbf{N}(v)$ is the neighbor set of node v. The aggregation method and the order of aggregation and update could vary across different GNNs. Some methods [14, 17] just rely on the neighboring nodes while others [38] also leverage the edge properties that are computed by applying vector dot-product between source and destination node embeddings. The update function is generally composed of standard NN operations, such as a single fully connected layer or a multi-layer perceptron (MLP) in the form of $w \cdot a_v^{(k+1)} + b$, where w and b are the weight and bias parameter, respectively. The common choices for node embedding dimensions are 16, 64, and 128, and the embedding dimension may change across different layers.

After passing through several iterations of aggregation and update (*i.e.*, several GNN layers), we will get the output feature embedding of each node, which can usually be used for various downstream graph-based deep learning tasks, such as node classification [7, 11, 15] and link prediction [5, 18, 37].

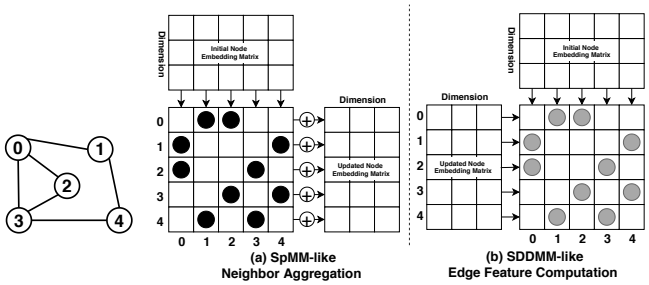

Figure 2: SpMM-like and SDDMM-like Operation in GNNs. Note that "→" indicates loading data; "⊕" indicates neighbor embedding accumulation.

The sparse computing in the aggregation phase is generally formalized as the sparse-matrix dense-matrix multiplication (SpMM), as illustrated in Figure 2(a), and is handled by many sparse libraries (*e.g.*, cuSPARSE [25]) in many state-of-the-art GNN frameworks [39]. These designs only count on GPU CUDA cores for computing, which waste the modern GPUs with diverse computing units, such as the Tensor Core Unit (TCU). Specifically, we formalized the neighbor aggregation as SpMM-like operations (Equation 2)

$$\hat{\mathbf{X}} = (\mathbf{F}_{N\times N} \odot \mathbf{A}_{N\times N}) \cdot X_{N\times D}) \tag{2}$$

where $\mathbf{A}$ is the graph adjacency matrix stored in CSR format. $\mathbf{X}$ is a node feature embedding matrix stored in dense format. N is the number of nodes in the graph, and D is the size of node feature embedding dimension; $\odot$ is the elementwise multiplication and $\cdot$ is the standard matrix-matrix multiplication; $\mathbf{F}$ is the edge feature matrix in CSR format and can be computed by SDDMM-like operations (Equation 3), as illustrated in Figure 2(b).

$$\mathbf{F} = (\mathbf{X}_{N\times D} \cdot \mathbf{X}_{N\times D}^T) \odot \mathbf{A}_{N\times N} \tag{3}$$

Note that the computation of F is optional in GNNs, which is generally adopted by Attention-based Graph Neural Network in Pytorch [36] for identifying more complicated graph structural information. Other GNNs, such as Graph Convolutional Network [17], Graph Isomorphism Network [40], only use the graph adjacency matrix for neighbor aggregation.

2.2 GPU Tensor Core

In the most recent GPU architectures (since Volta [30]), NVIDIA announced a new type of computing unit, Tensor Core Unit (TCU), for accelerating dense deep-learning operations (*e.g.*, Dense GEMM). A GPU Streaming-Multiprocessor (w/ TCU) is illustrated in Figure 3. Note that FP64, FP32, INT, and SFU are for double-precision, single-precision, integer, and special function unit, respectively.

Different from scalar-scalar computation on CUDA Cores, TCU provides a tile-based matrix-matrix computation primitives on register fragments, which can deliver more than 10× throughput improvement. In particular, Tensor Cores support the compute primitive of $\mathbf{D} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$, where $\mathbf{A}$ and $\mathbf{B}$ are required to be a certain

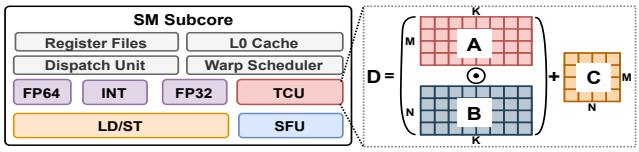

Figure 3: A Subcore of GPU SM with TCU.

type of precision (*e.g.*, half, TF-32), while **C** and **D** are stored in 32-bit single-precision floating-point. Depending on the data precision and GPU architecture version, the matrix size of $\mathbf{A}(M \times K)$, $\mathbf{B}(K \times N)$, and $\mathbf{C}(M \times N)$ should follow some principles [28]. For example, TF-32 TCU computing requires $M = N = 16$ and $K = 8$. In the most recent CUDA release (CUDA>=11.0) on Ampere ($sm >= 80$), TF-32 serves as a good alternative of float/double on TCU-based GPU computing for modern deep-learning applications, according to the in-depth research study [31] from NVIDIA.

Different from the CUDA cores that operates at the thread level (*e.g.*, allowing the "if" branch among threads), TCU supports only the operation at the warp level (*e.g.*, forbidding the "if" branch among threads within a warp). Before calling TCU, all registers in a warp need to collaboratively store matrix tiles into a new memory hierarchy *Fragment* [33], which allows data sharing across registers. This intra-warp sharing provides opportunities for fragment-based memory optimizations.

Listing 1: CUDA WMMA APIs for TCU.

```
wmma::fragment<matrix_a, M, N, K, tf32, row_major> a_frag;
wmma::load_matrix_sync(a_frag, A, M);
wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
wmma::store_matrix_sync(C, c_frag, N, mem_row_major);
```

TCU can be utilized in several ways. The simplest way is to call cuBLAS [27] by using the `cublasSgemmEX` API. The second way is to call the Warp Matrix Multiply-Accumulate (WMMA) (`nvcuda::wmma`) API [32] in CUDA C++ to operators Tensor Core directly. There are four major types of operations (Listing 1). Specifically, we use `wmma::fragment` to define the input matrix tile for TCU computation. Each fragment consists of thread-local registers from a warp of threads. `wmma::load_matrix_sync` loads the input matrix tiles from global/shared memory to register fragments. `wmma::mma_sync` executes the matrix multiplication on loaded matrix tiles in register fragments. Finally, `wmma::store_matrix_sync` moves the results from register fragments to global/shared memory.

Since the appearance of the Tensor Core, research efforts have been devoted to accelerating high-performance computing workloads with TCU. Ahmad et al. [2] process the batched small-size GEMM on Tensor Core for acceleration. Ang Li and Simon Su [21] leverage 1-bit GEMM capability on Turing GPU Tensor core for accelerating binary Neural Network inference. Boyuan et al. [8] introduce GEMM-based scientific computing on TCU with extended-precision and high performance. These prior efforts that mostly use the TCU in the dense applications that TCU are initially designed for, while TC-GNN jumps out of the scope defined by TCU designer by accelerating the sparse GNN operations using TCU.

3 MOTIVATION

In this section, we will discuss the major technical thrust for us to leverage TCU for accelerating sparse GNN computation. We use the optimization of SpMM as the major example in this discussion, and the acceleration of SDDMM would also benefit from similar optimization principles. We first characterize the existing GNN computation solutions, including SpMM on CUDA core, Dense GEMM on CUDA core/TCU, and a hybrid Sparse-Dense solution. Then we give insights based on pros/cons analysis and our motivation.

3.1 SpMM on CUDA core

As the major components of sparse linear algebra operation, SpMM has been incorporated in many off-the-shelf libraries [1, 3, 4, 25]. Among those implementations, the close-sourced cuSPARSE [25] developed by NVIDIA can deliver the state-of-the-art performance for GPU-based sparse matrix computation. cuSPARSE has also been widely adopted by the many popular GNN computing framework, such as Deep Graph Library (DGL) [39], as the backend for the sparse neighbor aggregation operations. To understand its characters, we choose DGL as the platform and profile one layer (*neighbor aggregation + node update*) of a GCN [17] model on NVIDIA RTX3090 GPU. We report two key kernel matrices for only neighbor aggregation kernel, including L1/texture cache hit rate (*Cache*) and the achieved Streaming-Multiprocessor (SM) occupancy (*Occ.*). From Table 1, we have several observations.

Table 1: Profiling of GCN Sparse Operations.

Dataset	Node	Edges	Dim	Aggr. (%)	Update (%)	Cache(%)	Occ.(%)
Cora	3,327	9,464	3703	88.56	11.44	37.22	15.06
Citeseer	2,708	10,858	1,433	86.52	13.47	38.18	15.19
Pubmed	19,717	88,676	500	94.39	5.55	37.22	16.24

First, the aggregation phase usually dominates the overall execution of the GNN execution. From these three commonly used GNN datasets, we can see that the aggregation phase usually takes more than 80% of the overall execution time, which demonstrates the key performance bottleneck of the GNNs is improve the performance of the sparse neighbor aggregation.

Second, sparse operations in GNNs show very low memory performance. The column *Cache* of Table 1 shows GNN sparse operations could not well benefit from GPU cache system, thus, showing low cache-hit ratio (around 37%) and indicating frequent high-cost global memory access.

Third, sparse operations of GNNs show very inefficient computation. As described in the column *Occupancy* of Table 1, sparse operations of GNNs could hardly keep the GPU busy because 1) its low computation intensity (the number of non-zero elements in the sparse matrix is generally small); 2) its highly-irregular memory access for fetching rows of dense matrix during the computation, resulting in memory bound computation; 3) it currently can only leverage CUDA core for computation, which naturally has limited throughput performance.

On the other side, this study also points out several potential directions of improving the SpMM performance on GPUs, such as improving the computation intensity (*e.g.*, assigning more workload to each thread/warp/block), boosting memory access efficiency (*e.g.*, crafting specialized memory layout for coalesced memory access), and breaking the computation performance ceiling (*e.g.*, using TCU).

3.2 Dense GEMM on CUDA Core/TCU

While the Dense GEMM is mainly utilized for dense NN computations (*e.g.*, linear transformation and convolution), it can also be leveraged for GNN aggregation under some circumstances. For example, when an input graph has very limited number of nodes, we can directly use the dense adjacency matrix of the graph and accelerate the intrinsically sparse neighbor aggregation computation

Table 2: Comparison among Sparse GEMM, Dense GEMM, Hybrid Sparse-Dense, and TC-GNN.

Solution	Mem. Consumption	Effective Mem. Access	Computation Intensity	Effective Computation
Sparse GEMM (§3.1)	Low	Low	Low	High
Dense GEMM (§3.2)	High	High	High	Low
Hybrid Sparse-Dense (§3.3)	High	Low	Low	High
TC-GNN	**Low**	**High**	**High**	**High**

on CUDA core/TCU by calling cuBALS/CUTLASS. However, such assumption may not hold even for medium-size graphs in the real-world GNN applications. As shown in Table 3, for these selected

Table 3: Medium-size Graphs in GNNs.

Dataset	# Nodes	# Edges	Memory	Eff.Comp
OVCR-8H	1,890,931	3,946,402	14302.48 GB	0.36%
Yeast	1,714,644	3,636,546	11760.02 GB	0.32%
DD	334,925	1,686,092	448.70 GB	0.03%

datasets, the memory consumption of their dense graph adjacent matrix (as a 2D `float` array) would easily exceed the memory constraint of the today's GPU (<100GB). Second, even if we assume the dense adjacent matrix can fit into the GPU memory, the extremely low effective computations (last column of Table 3) would also be a major obstacle for us to achieve high performance. We measure the effective computation as $\frac{nnz}{N \times N}$, where nnz is the number of the non-zero elements (indicating edges) in the graph adjacent matrix and N is the number of nodes in the graph. The number of nnz is tiny in comparison with the $N \times N$. Therefore, most of the computations and memory access on those zero elements are wasted in such GEMM-based solution on CUDA Core/TCU.

3.3 Hybrid Sparse-Dense Solution

Another type of work [19, 26] takes the path of mixing the *sparse control* (tile-based iteration) with *Dense GEMM computation*. They first apply convolution-like (2D sliding window) operation on the adjacent matrix and traverse all possible blocks that contain non-zero elements. Then, for all identified non-zero tiles, they invoke GEMM on CUDA Core/TCU for computation. However, this strategy has two shortcomings. **First**, sparse control itself would cause high overhead. Based on our empirical study, the non-zero elements are highly scattered on the adjacent matrix of a sparse graph. Therefore, traversing all blocks in a super large adjacent matrix would be time consuming. **Second**, the identified sparse tiles would still waste lots of computations. The irregular edge connections of the real-world graphs could hardly fit into these fixed-shape tile frame. Therefore, most of the dense tiles would still have very low occupation (very few non-zero elements in each tile).

Inspired by the above studies, we make several key design choices in order to achieve high-performance sparse GNN operations. ***1) At the algorithm level***, we choose the hybrid sparse-dense solution as the starting point. This can give us more flexibility for optimizations at the sparse control (*e.g.*, traversing less tiles) and dense computation (*e.g.*, increasing the effective computation/memory access when processing each tile), ***2) At the GPU kernel level***, we employ the shared memory as the key space for GPU kernel-level data management. It can help us to re-organize the irregular GNN input data in a more "regularized" way such that both the memory access efficiency and computing performance can be well improved. ***3) At the hardware level***, we choose TCU as our major computing unit since it can bring significantly higher computing throughput performance in comparison with CUDA Core. This also indicates great potential of using TCU for harvesting more performance gains. Finally, we crystallize our idea into TC-GNN that effectively coordinates the execution of GNN sparse operations on dense TCU. We show a brief qualitative comparison among TC-GNNand the above three solutions in Table 2 and we justify these benefits through detailed discussion of TC-GNN in the next two sections. Note that *Memory Consumption* is the size of memory used by the sparse/dense graph adjacency matrix; The *Effective Memory Access* is the ratio between the size of the accessed data that is actually involved in the later computation and the total size of data being accessed; The *Computation Intensity* is the ratio of the number of computing operations versus the data being accessed; The *Effective Computation* is the ratio between the operations for generating the final result and the total operations.

4 TC-GNN

In this section, we will detail TC-GNN, the first design that accelerates the GNN sparse operations on dense Tensor Core Unit (TCU) of GPU. It includes three major algorithmic designs: *Sparse Graph Translation*, *Sparse (SpMM-like) Neighbor Aggregation*, and *Sparse (SDDMM-like) Edge Feature Computing*.

4.1 TCU-Aware Sparse Graph Translation

As the major component of TC-GNN, we propose a novel sparse graph translation (SGT) technique to facilitate the TCU acceleration of GNNs. our core idea is that *the pattern of the graph sparsity can be well tuned for TCU computation through an effective graph structural manipulation meanwhile guaranteeing the output correctness*. As

Algorithm 1: TCU-aware Sparse Graph Translation.

```
input  : Graph adjacent matrix A in CSR (nodePointer, edgeList).
output : Result of winPartition and edgeToCol.
/* Compute the total number of row windows.                    */
numRowWin = ceil(numNodes/winSize);
for winId in numRowWin do
    /* Get the edgeIndex range of the current rowWindow.       */
    winStart = nodePointer[winId * winSize];
    winEnd = nodePointer[(winId + 1) * winSize];
    /* Sort the edges of the current rowWindow.                */
    eArray = Sort(winStart, winEnd, edgeList);
    /* Deduplicate edges of the current rowWindow.             */
    eArrClean = Deduplication(nIdArray);
    /* Count the num. TCblocks of the current rowWindow.       */
    winPartition[winId] = ceil(eArrClean.size/TC_BLK_w);
    /* Create mapping from edges to columnID in TC Blocks.     */
    for eIndex in [winStart, winEnd] do
        eid = edgeList[eIndex];
        edgeToCol[eIndex] = eArrClean[eid];
    end
end
```

exemplified in Figure 4(a) and (b), we take the regular graph in CSR format as the input and condense the columns of each row window (circled with red-colored rectangular box) to build a set of TCU

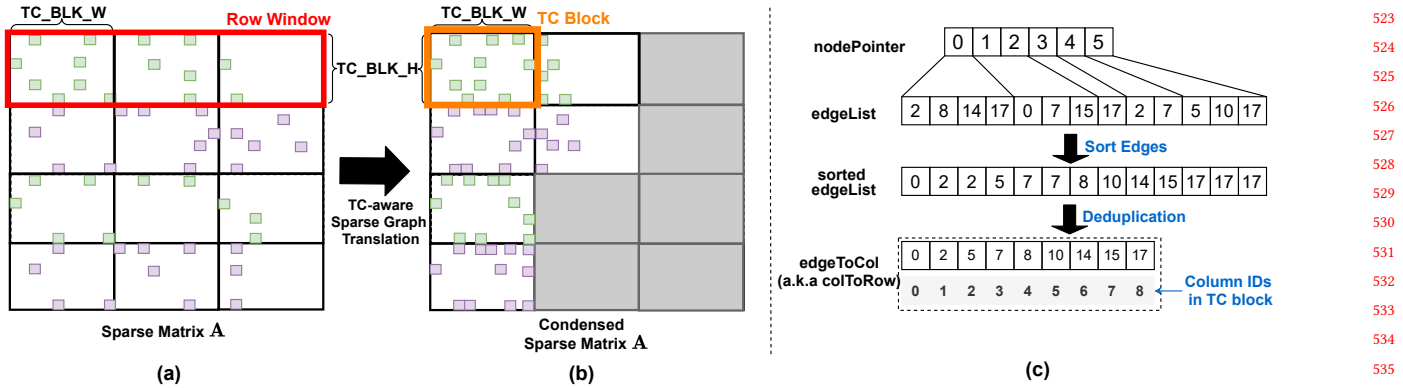

Figure 4: Sparse Graph Translation. Note that the grey-colored area indicates the TCU blocks that will be directly skipped.

blocks (*TC_block*) (*a.k.a.*, the input operand shape of a single MMA instruction), as circled with orange-colored rectangular box. Note that in this paper we demonstrate the use of standard MMA shape for TF-32 of TCU on Ampere GPU architecture, and other MMA shapes [28] can also be used if different computation precision (*e.g.*, `half` and `INT8`) and GPU architecture (*e.g.*, Turing) are specified.

Our sparse graph translation scheme takes several steps for processing each row window, as detailed in Algorithm 1 and visualized in Figure 4(c). Note that *winPartition* is an array for maintaining the number of TC blocks in each row window. *edgeToCol* is an array for maintaining the mapping between the edges and their corresponding position in graph after SGT. We choose the size of the row window (*winSize*=*TC_BLK_H*) and column width (*TC_BLK_W*) according to TCU MMA specification (*e.g.*, *TC_BLK_H*=16, *TC_BLK_W*=8 in TF-32). After condensing the graph within each row window, the time complexity of sliding the *TC_block* can be reduced from $O(\frac{N}{TC\_BLK\_W})$ to only $O(\frac{nnz_{unique}}{TC\_BLK\_W})$, where N is total number of nodes in the graph and nnz_{unique} is the size of the unique neighbor within the current row window, which equals *eArrClean.size* in Algorithm 1. Besides, the density (computation intensity) of each identified TCU block can be largely improved. Considering the case in Figure 4, after the sparse graph translation, we can achieve 2× higher density on individual TCU blocks (Figure 4(b)) compared with the original one (Figure 4(a)). Note that SGT is applicable for both the SpMM and SDDMM in GNN sparse operations, and it can be easily parallelized because the processing of individual row windows are independent from each other. Besides, the sparse graph translation only needs to execute once and its result can be reused across many epochs/rounds of GNN training/inference.

4.2 TCU-tailored GNN Computation

Neighbor Aggregation The major part of GNN sparse computing is the neighbor aggregation, which can generally be formalized as SpMM operations by many state-of-the-art frameworks [39]. And they employ the cuSPARSE [25] on CUDA core as a black-box technique for supporting sparse GNN computation. In contrast, our TC-GNN design targets at TCU for the major neighbor aggregation computation which demands a specialized algorithmic design. TC-GNN focuses on maximizing the net performance gains by gracefully batching the originally highly irregular SpMM as dense GEMM computation and solving it on TCU effectively.

Algorithm 2: TCU-tailored Neighbor Aggregation.

input : Condensed graph structural information (*nodePointer*, *edgeList*, *edgeToCol*, *winPartition*) and node feature embedding matrix (**X**).
output: Updated node feature embedding matrix ($\hat{\mathbf{X}}$).

```
/* Traverse through all row windows.                          */
for winId in numRowWindows do
    /* Get the number of TC blocks of the row window.         */
    numTCblocks = winPartition[winId] ;
    /* Get edge range of TC blocks of the row window.         */
    edgeRan = GetEdgeRange(nodePointer, winId);
    for TCblkId in numTCblocks do
        /* Get a chunk of edgeList in current TC block.       */
        edgeChunk = GetChunk(edgeList, edgeRan, TCblkId);
        /* Get neighbor node Ids in current TC block.         */
        colToNId = GetNeighbors(edgeChunk, edgeToCol);
        /* Initiate a dense tile (ATile).                     */
        ATile = InitSparse(edgeChunk, winId);
        /* Initiate a dense tile (XTile).                     */
        XTile, colId = FetchDense(colToNId, X);
        /* Compute XnewTile via GEMM on Tensor Core.          */
        XnewTile = TCcompute(ATile, XTile);
        /* Store XnewTile of X̂.                               */
        X̂ = StoreDense(XNewTile, winId, colId);
    end
end
```

As illustrated in Algorithm 2, the node aggregation processes all TC blocks from each row window. *nodePointer* and *edgeList* are directly from graph CSR, while *edgeToCol* and *winPartition* are generated from SGT discussed in the previous section. Note that **InitSparse** is to initialize a sparse tile in dense format according to the translated graph structure of the current TC block. Meanwhile, **FetchDense** returns a dense node embedding matrix tile *XTile* for TCU computation, and the corresponding column range *colId* (embedding dimension range) of matrix **X**. This is to handle the case that the width of one *XTile* could not cover the full-width (all dimensions) of **X**. Therefore, the *colId* will be used to put the current TCU computation output to correct location in updated node embedding matrix $\hat{\mathbf{X}}$.

Edge Feature Computing Previous research efforts [36, 38] have demonstrated the great importance of incorporating the edge

Algorithm 3: TCU-tailored Edge Feature Computing.

```
input : Condensed graph structural information (nodePointer,
        edgeList, edgeToCol, winPartition) and node feature
        embedding matrix (X).
output: Edge Feature List (edgeValList).
/* Traverse through all row windows.                          */
for winId in numRowWin do
    /* Get the number of TC blocks of the row window.         */
    numTCblocks = winPartition[winId] ;
    /* Get edge range of TC blocks of the row window.         */
    edgeRan = GetEdgeRange(nodePointer, winId);
    for TCblkId in numTCblocks do
        /* Get a chunk of edgeList in current TC block.       */
        edgeChunk = GetChunk(edgeList, edgeRan, TCblkId);
        /* Get neighbor node Ids in current TC block.         */
        colToNId = GetNeighbors(edgeChunk, edgeToCol);
        /* Fetch a dense tile (XTile_A).                      */
        XTile_A = FetchDenseRow(winId, TCblkId, X);
        /* Fetch a dense tile (XTile_B).                      */
        XTile_B = FetchDenseCol(colToNId, edgeToCol, X);
        /* Compute edgeValTile via GEMM on Tensor Core.       */
        edgeValTile = TCcompute(XTile_A, XTile_B);
        /* Store edgeValTile to edgeValList.                  */
        StoreSparse(edgeValList, edgeValTile,
                    edgeList, edgeToCol);
    end
end
```

feature for a better GNN model algorithmic performance (*e.g.*, accuracy and F1-score). The underlying building blocks to generate edge features is the Sparse Dense-Dense Matrix Multiplication (SDDMM)-like operation. In TC-GNN, we support SDDMM with the collaboration of the above sparse graph translation and TCU-tailored algorithm design, as described in Algorithm 3. The overall algorithm structure and inputs are similar to the above neighbor aggregation. The major difference is the output. In the case of neighbor aggregation, our output is the updated dense node embedding matrix ($\hat{\mathbf{X}}$), where edge feature computing will generate a sparse output with the same shape as the graph edge lists. Note that fetching the $XTile_A$ only needs to consecutively access the node embedding matrix $\mathbf{A}$ by rows, while fetching the $XTile_B$ requires first computing the TCU block column-id to node-id ($colToNId$) to fetch the corresponding neighbor node embeddings from the same node embedding matrix $\mathbf{X}$.

5 IMPLEMENTATION

In this section, we detail TC-GNNby mapping the above algorithmic design to low-level primitives (*e.g.*, warp/block) and customizing shared memory layout. We discuss two key techniques: *two-level workload mapping* and *TCU-optimized dataflow design.*

5.1 Two-level Workload Mapping

Different from previous work [9, 39] focusing on CUDA core only, TC-GNN highlights itself with CUDA core and TCU collaboration through effective two-level workload mapping. The idea is based on the fact that CUDA Cores work in SIMT fashion and are operated by individual threads, while TCU designated for GEMM computation requires the collaboration from a warp of threads (32 threads). Our key design principle is to mix these two types of computing units as a single GPU kernel, which can efficiently coordinate the kernel execution at different levels of execution granularity.

In TC-GNN, we operate CUDA cores by thread blocks and manage TCU by thread warps. Specifically, threads running CUDA cores from the same thread block will load data (*e.g.*, edges) from the global memory to shared memory. Note that in our design we assign each row window (discussed in Section 4.1) to one thread block. The number of threads in each blocks should be divisible by the number of threads in each warp (32) for better performance. Once threads running on CUDA core (CUDA-core threads) finish the data loading, threads from each warp (TCU threads) will operate TCU for GEMM computation (including loading the data from the shared memory to thread-local registers (fragments), applying GEMM computation on data in registers, accumulating results on registers, and storing the final results back to global memory). Note that there would be large overlap of the CUDA-core threads and TCU threads, both of which are threads from the same blocks but running at different time frame. In general, we use more CUDA-core threads than TCU threads considering that global memory access demanding more parallelization. There are two major benefits of such a two-level workload mapping strategy. First, threads from the same block can work together to improve the memory access parallelization to better utilize memory bandwidth. Second, warps from the same block can reuse the loaded data, including the information (*e.g.*, column index mapping) of the translated graph and the tiles from the dense node embedding matrix. Therefore, redundant high-cost global memory operations can largely be avoided.

5.2 TCU-Optimized Dataflow Design

As the major technique to improve the GPU performance, shared memory is customized for our TCU-based sparse kernel design for re-organizing data layout for dense TCU computation and reducing the redundant global memory traffic. Our design takes the TCU specialty into careful consideration from two aspects, 1) the input matrix tile size of the TCU, which is M(16)×N(16)×K(8) in case of TF-32, and 2) the tile fragment layout for fast computation. The common practice of the loaded tile A and B are stored in row-major and column-major for better performance. Next we will detail our TCU-optimized dataflow design for both neighbor aggregation and edge feature computation.

Neighbor Aggregation As visualized in Figure 5(a) and detailed in Listing 2, shared memory is mainly used for caching several most frequently used information, including the tile of sparse matrix A (`sparse_A`), the column-id of the sparse matrix $\mathbf{A}$ to row-id of node embedding matrix $\mathbf{X}$ (`sparse_AToX_index`), and the dense tile of $\mathbf{X}$ (`dense_X`). When handling each TCU block, we assign all threads from the same block of threads for loading the sparse tile while allowing several warps to concurrently load the dense row tile from the matrix X. The reasons for enforcing such caching are two-folds. First, it can bridge the gap between the sparse graph data and the dense GEMM computing that requires continuous data layout. For example, the adjacent matrix $\mathbf{A}$ is input as CSR format that **cannot** be directed feed to TCU GEMM computation, therefore, we use a shared memory `sparse_A` to initialize its equivalent dense tile. Similarly, we cache rows of $\mathbf{X}$ according to the columns of $\mathbf{A}$ to the row of $\mathbf{X}$ mapping after our sparse graph translation, where originally scattered columns of $\mathbf{A}$ (rows of $\mathbf{X}$) are condensed. Second, it can enable the data reuse on `sparse_AToX_index` and `sparse_A`.

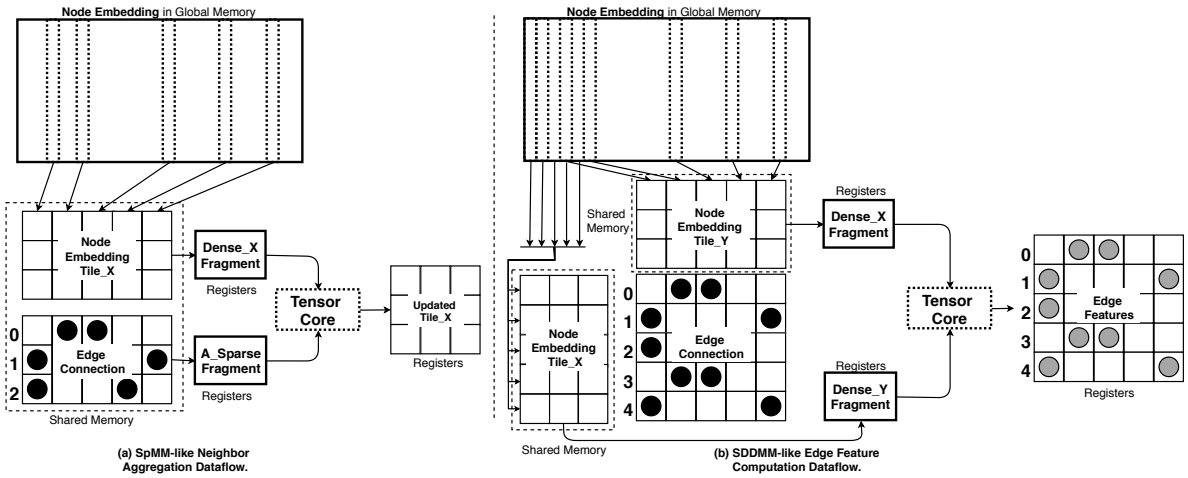

Figure 5: TCU-Optimized Dataflow Design for (a) Neighbor Aggregation and (b) Edge Feature Computing in GNNs.

Listing 2: Implementation of Neighbor Aggregation.

```
1  __shared__ float sparse_A[BLK_H * BLK_W];
2  __shared__ unsigned sparse_AToX_index[BLK_H];
3  __shared__ float dense_X[warpPerblock * BLK_W * BLK_H];
4
5  for (i = 0; i < num_TC_blocks; i++){
6      tid = threadIdx.x; // thread id.
7      wid = tid % 32;    // warp id.
8      // Assigning dummy value for handling corner cases.
9      if (wid == 0 && laneid < BLK_W)
10         sparse_AToX_index[laneid] = numNodes + 1;
11     // Loading edges and initialize sparse_A.
12     for (eIdx = n_start+tid; eIdx < n_end; eIdx += threadPerBlock){
13         col = edgeToColumn[eIdx];
14         // if the edge in the current TC_block frame of column.
15         if (i*BLK_W <= col && col < (i+1)*BLK_W){
16             unsigned row = edgeToRow[eIdx] % BLK_H;
17             // set the edge of the sparse_A.
18             sparse_A[row*BLK_W + col%BLK_W] = 1;
19             // map columns of sparse_A to rows of dense_X.
20             sparse_AToX_index[col % BLK_W] =  edgeList[eIdx];
21         }
22     }
23     // Initialize dense_X by column-major store,
24     // Threads of a warp for fetching a dense_X.
25     // each warp identify by wid.
26     for (i = tid; i < BLK_W*BLK_H; i += warpSize){
27         // TC_block_col to dense_tile_row.
28         dense_rowIdx = sparse_AToX_index[i%BLK_W];
29         // dimIndex of the dense tile.
30         dense_dimIdx = i / BLK_W;
31         source_idx = wid * BLK_W*BLK_H + i;
32         target_idx = dense_rowIdx*embedding_dim + wid*dimPerWarp
33                    + dense_dimIdx;
34         dense_X[source_idx] = in_mat[target_idx];
35     }
36     // Call wmma load A_frag, X_frag from shared memory
37     // Compute and accumulate. Store to global memory of X_hat.
38 }
```

Because in general the BLK_H (16) cannot cover all dimension of a node embedding (*e.g.*, 64), multiple warps will be initiated of the same block to operate TCU in parallel to working on non-overlapped dense tiles while using the same sparse tile.

Edge Feature Computation Similar to the shared memory design in neighbor aggregation, for edge feature computing, as visualized in Figure 5(b) and detailed in Listing 3 at the next page, the shared memory is utilized for sparse tile A `sparse_A`, the column-id of sparse **A** to row-id of the matrix **X** `sparse_AToX_index`, and the dense tile `dense_X` from the matrix **X**. We assign all threads from the same block of threads for loading the sparse tile while allowing several warps to concurrently load the dense row tile from the matrix **X**. Compared with dataflow design in neighbor aggregation, edge feature computing demonstrates several differences. First, the sizes of `sparse_A` is different. In the neighbor aggregation computation, the sparse matrix **A** is used as one operand in the SpMM-like computation, therefore, the minimal processing granularity is 16×8, while in edge feature computing by following SDDMM-like operation, the sparse matrix **A** is served as the output matrix, thus, maintaining the minimum processing granularity is 16×16. To reuse the same translated sparse graph as SpMM, we need to recalculate the total number of TC blocks (Line 9). Second, iterations along the node embedding dimension would be different. Compared with neighbor aggregation, edge feature computing requires the result accumulation along the node embedding dimension. Therefore, the result will only be output until all iterations have finished. In neighbor aggregation case, node embedding dimension is divided among several warps, each of which will output their aggregation result to non-overlapped embedding dimension range in parallel. Third, the output format has changed. Compared with SpMM-like neighbor aggregation which directly output computing result as an updated dense matrix $\hat{\mathbf{X}}$, SDDMM-like edge feature computing requires a sparse format (the same shape as `edgeList`) output for compatibility with neighbor aggregation and memory space. Therefore, one more step of dense-to-sparse translation is employed after the result accumulation.

6 EVALUATION

In this section, we comprehensively evaluate TC-GNN on various GNN models and graph datasets.

Benchmarks: We choose the two most representative GNN models widely used by previous work [9, 24, 39] on *node classification* tasks, which can cover different types of aggregation. Specifically, 1) Graph Convolutional Network (**GCN**) [17] is one of the most popular GNN model architectures. It is also the key backbone network for many other GNNs, such as GraphSAGE [14], and differentiable pooling (Diffpool) [41]. Therefore, improving the performance of GCN will also benefit a broad range of GNNs. For GCN evaluation, we use the setting: *2 layers with 16 hidden dimensions*, which is also the setting from the original paper [17]. 2)

Listing 3: Implementation of Edge Feature Computation.

```
__shared__ float sparse_A[BLK_H*BLK_H];
__shared__ unsigned sparse_AToX_index[BLK_H];
__shared__ float dense_X[BLK_H*BLK_W];
__shared__ float dense_Y[BLK_W*BLK_H];

// Processing TC_blocks along the column dimension of Sparse A.
// The block step here is 2, which is 16 = 8 + 8.
// In order to reuse the edgeToColumn in SpMM.
num_TC_blocks = (blockPartition[bid]*BLK_W + BLK_H - 1)/BLK_H;
// dimension iteration for covering all dimension.
DimIterations =  (embedding_dim+BLK_W-1)/BLK_W;
// traversing all TC blocks in the current row window.
for (i = 0; i < num_TC_blocks; i++){
    if (wid == 0 && laneid < BLK_H)
        sparse_AToX_index[laneid] = numNodes + 1;

    for (idx = tid; idx < BLK_H*BLK_H; idx += threadPerBlock)
        sparse_A[idx] = numEdges + 1;

    // Initialize sparse_A by using BLK_H (16) threads from the
    // warp-0. Specifically, we fetch all neighbors of the
    // current nodes, then to see whether it can fit into
    // current TC_block frame of column.
    for (eIdx = tid+n_start; eIdx < n_end; eIdx += threadPerBlock){
        // condensed column id in sparse_A.
        col = edgeToColumn[eIdx];
        // if the edge in the current TC_block frame of column.
        if (i*BLK_H <= col && col < (i+1)*BLK_H){
            // reverse indexing the row Id of the edge.
            row = edgeToRow[eIdx] % BLK_H;
            // set the edge of the sparse_A.
            sparse_A[row*BLK_H + col%BLK_H] = eIdx;
            // map sparse_A colId to dense_X rowId.
            sparse_AToX_index[col%BLK_H] = edgeList[eIdx];
        }
    }
    // traverse all dimension of the same sparse tile.
    for (dim_iter = 0; dim_iter < DimIterations; dim_iter++){
        // Initialize dense_X by row-major store
        // Threads of a block for fetching a dense_X.
        for (i = tid; i < BLK_H*BLK_W; i += threadPerBlock){
            dense_rowIdx = i / BLK_W;
            dense_dimIdx = i % BLK_W;
            target_idx = i;
            source_idx = dense_rowIdx*embedding_dim \
                        + dim_iter*BLK_W + dense_dimIdx;
             // boundary check for padding.
            if (source_idx >= numNodes*embedding_dim)
                dense_X[target_idx] = 0;
            else
                dense_X[target_idx] = in_mat[source_idx];
        }
        // Initialize dense_Y by column-major store,
        // Threads of a warp for fetching a dense_Y.
        for (i=tid; i < BLK_W*BLK_H; i += threadPerBlock){
            // TC_block_col to dense_tile_row.
            dense_rowIdx = sparse_AToX_index[i%BLK_H];
            // dimIndex of the dense tile.
            dense_dimIdx = i / BLK_W;
            target_idx = i;
            source_idx = dense_rowIdx*embedding_dim \
                        + dim_iter*BLK_W + dense_dimIdx;
            // boundary check for padding.
            if (source_idx >= numNodes*embedding_dim)
                dense_Y[target_idx] = 0;
            else
                dense_Y[target_idx] = in_mat[source_idx];
        }
        __syncthreads();
        // Call wmma load dense_X and dense_Y from shared memory
        // Compute and accumulate.
  }
        // Store to EdgeValList.
}
```

Attention-based Graph Neural Network in Pytorch (**AGNN**) [36]. AGNN differs from GCN and GIN in its aggregation function, which compute edge feature (via embedding vector dot-product between source and destination vertices) before the node aggregation. AGNN is also the reference architecture for many other recent GNNs for better model algorithmic performance. For AGNN evaluation, we use the setting: *4 layers with 32 hidden dimensions.*

Table 4: Datasets for Evaluation.

Type	Dataset	#Vertex	#Edge	Dim.	#Class
I	Citeseer	3,327	9,464	3703	6
	Cora	2,708	10,858	1433	7
	Pubmed	19,717	88,676	500	3
	PPI	56,944	818,716	50	121
II	PROTEINS_full	43,471	162,088	29	2
	OVCAR-8H	1,890,931	3,946,402	66	2
	Yeast	1,714,644	3,636,546	74	2
	DD	334,925	1,686,092	89	2
	YeastH	3,139,988	6,487,230	75	2
III	amazon0505	410,236	4,878,875	96	22
	artist	50,515	1,638,396	100	12
	com-amazon	334,863	1,851,744	96	22
	soc-BlogCatalog	88,784	2,093,195	128	39
	amazon0601	403,394	3,387,388	96	22

Baselines: We choose several baseline implementations for comparison. 1) Deep Graph Library (**DGL**) [39] is the state-of-the-art GNN framework on GPUs, which is built with the high-performance CUDA-core based cuSPARSE [25] library as the back-end and uses Pytorch [34] as its front-end programming interface. DGL significantly outperforms other existing GNN frameworks [9] over various datasets on many mainstream GNN model architectures. Therefore, we make an in-depth comparison with DGL. 2) Pytorch-Geometric (**PyG**) [9] is another GNN framework in which users can define their edge convolutions when building customized GNN aggregation layers. PyG leverages torch-scatter [10] library (highly-engineered CUDA-core kernel) as the backend support, which highlights its performance on batched small graph settings; 3) Blocked-SpMM [26] (**bSpMM**) is for accelerating SpMM on TCU. It is included in the most recent update (March. 2021) on cuSPARSE library (CUDA 11.2). bSpMM requires the sparse matrix with Blocked-Ellpack (Blocked-ELL) storage format for computation. Its computation on non-zero blocks can be seen as the hybrid sparse-dense solution discussed in Section 3.3. Note that the bSpMM has not been incorporated in any existing GNN frameworks.

Datasets: We cover three types of datasets, which have been used in previous GNN-related work [9, 24, 39]. **Type I** graphs are the typical datasets used by previous GNN algorithm papers [14, 17, 40]. They are usually small in the number of nodes and edges, but rich in node embedding information with high dimensionality. **Type II** graphs [16] are the popular benchmark datasets for graph kernels and are selected as the built-in datasets for PyG [9]. Each dataset consists of a set of small graphs, which only have intra-graph edge connections without inter-graph edge connections. **Type III** graphs [17, 20] are large in terms of the number of nodes and edges. These graphs demonstrate high irregularity in its structures, which are challenging for most of the existing GNN frameworks. Details of these datasets are listed in Table 4.

Platforms & Metrics: TC-GNN backend is implemented with C++ and CUDA C, and TC-GNN front-end is implemented in Python. Our major evaluation platform is a server with an 8-core 16-thread Intel Xeon Silver 4110 CPU and an NVIDIA RTX3090 GPU. To measure the performance speedup, we calculate the averaged latency of 200 end-to-end results.

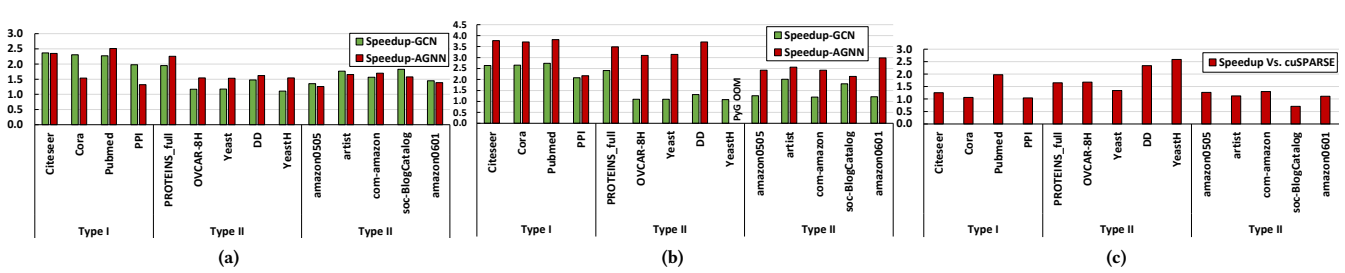

Figure 6: Speedup over (a) DGL and (b) PyG on GCN and AGNN; (c) Speedup over cuSPARSE bSpMM on TCU.

6.1 Compared with DGL

In this section, we first conduct a detailed experimental analysis and comparison with DGL on end-to-end GNN training. As shown in Figure 6(a), TC-GNN achieves 1.70× speedup on average compared to DGL over three types of datasets across GCN and AGNN model on end-to-end training. We next provide detailed analysis for each type of dataset.

Type I Graphs: The performance improvements against DGL is significantly higher for GCN (average 2.23×) compared to AGNN (average 1.93×). The major reason is their different GNN computation patterns. For GCN, it only consists of a neighbor aggregation phase (SpMM-like operation) and a node update phase (GEMM-like operation). Whereas in the AGNN, the aggregation phase would also require an additional edge attention value (feature) computation based on SDDMM-like operations. Compared with SpMM-like operations, edge attention computation in SDDMM is more sensitive to the irregular sparse graph structure because of much more intensive computations and memory access. Therefore, the performance improvement is relatively lower on AGNN.

Type II Graphs: TC-GNN achieves GCN (1.38×) and AGNN (1.70×) on the Type II graphs. Speedup on Type II graphs is relatively lower compared with Type I. Because Type II datasets consisting of a set of small graphs with very dense intra-graph connections but no inter-graph edges. This would lead to a lower benefit from the sparse graph translation that would show more effectiveness on highly irregular and sparse graphs by shrinking the TCU iteration space. Meanwhile, such a clustered graph structure would also benefit cuSPARSE due to more efficient memory access, *i.e.*, the fewer number of irregular data fetching from the sparse matrix. In addition, for AGNN, TC-GNN can still demonstrate evident performance benefits over the DGL (CUDA core only), which can mainly contribute to our TCU-based SDDMM-like operations design that can fully exploit the power of GPU through an effective TCU and CUDA core collaboration.

Type III Graphs: The speedup is also evident (average 1.59× for GCN and average 1.51× for AGNN) on graphs with a large number of nodes and edges and irregular graph structures. The reason is the high overhead global memory access can be well reduced through our spare graph translation. Besides, our dimension-split strategy further facilitates efficient workload sharing among warps through improving the data spatial/temporal locality. On the dataset *artist* and *soc-BlogCatalog*, which have a higher average degree within Type III datasets, we notice a better speedup performance for both GCN and AGNN. This is because 1) more neighbors per node can lead to higher density of non-zero elements within each tile/fragment. Thus, it can fully exploit the computation benefits of each TCU GEMM operation; 2) it can also facilitate more efficient memory access. For example, in AGNN, fetching one dense row (node embedding of one node) x from the dense matrix X can be reused more times by applying dot-product between x and many columns of (node embedding of neighbors) the dense matrix X^T.

6.2 Compared with Other Baselines

Comparison with PyG We further compare TC-GNN with PyG [9], which is another popular GNN computing framework built on the highly engineered torch-scatter [10] library running on CUDA core. As shown in Figure 6(b), TC-GNN can outperform PyG with an average 1.76× speedup on GCN and an average 2.82× speedup on AGNN. For GCN, TC-GNN achieves significant speedup on datasets with high-dimensional node embedding, such as *Yeast*, through 1) effective TCU acceleration through a TCU-aware sparse graph translation. 2) reducing the synchronization overhead by employing our highly parallelized TCU-tailored algorithm design. PyG, however, achieves inferior performance because 1) its underlying GPU kernel can only leverage CUDA core, thus, intrinsically bounded by the CUDA core computing performance; 2) its kernel implementation heavily rely on the high-overhead atomic operations for thread-level synchronization, therefore, preventing itself to achieve high-enough performance.

Compared with cuSPARSE bSpMM on TCU Most recently, cuSPARSE introduce a new block-wise SpMM [26](bSpMM) that can exploit the NVIDIA TCU capability, whereas previous cuSPARSE SpMM kernel can only leverage CUDA core. We compare our TC-GNN SpMM kernel with cuSPARSE bSpMM to demonstrate the performance advantage of TC-GNN compared with the state-of-the-art hybrid sparse-dense solution on TCU. As shown in Figure 6(c), TC-GNN can outperform PyG with an average 1.76× speedup on neighbor aggregation. Our sparse graph translation technique can maximize the non-zero density of each non-zero tile and significantly reduce the number of non-zero tiles to be processed. However, bSpMM in cuSPARSE has to comply with the strict input sparse pattern as indicates in their official API documentation [29]. For example, all rows in the arrays must have the same number of non-zero blocks. Therefore, more redundant computations (on padding those non-structural zero blocks) in bSpMM leads to a inferior performance compared with our TC-GNN design.

6.3 Additional Studies

Sparse Graph Translation To demonstrate the effectiveness of our sparse graph translation (SGT) technique, we conduct a

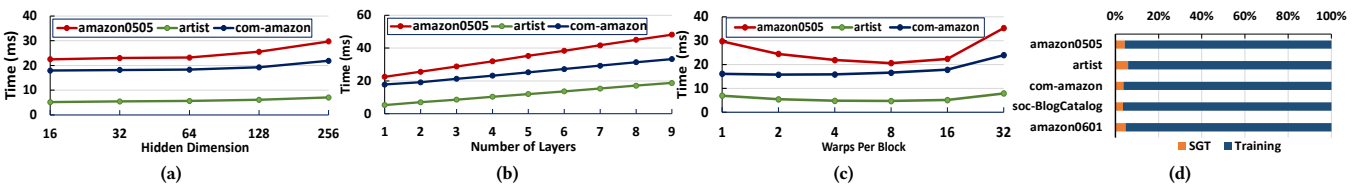

Figure 7: Optimization Analysis. (a) The impact of hidden dimension size; (b) the impact of the number of layers; (c) the impact of the number of warps per block; (d) the overhead analysis of sparse graph translation.

quantitive analysis in terms of the total number of TCU blocks between graphs w/o SGT and the graphs w/ SGT applied. Note that in the SpMM-based aggregation, the size of TCU Block is 16×8 since it serves as one of the operand in TCU GEMM. While in SDDMM-based edge feature computation, the size of TCU Block is 16×16 since it serves as the resulting matrix of TCU GEMM. As shown

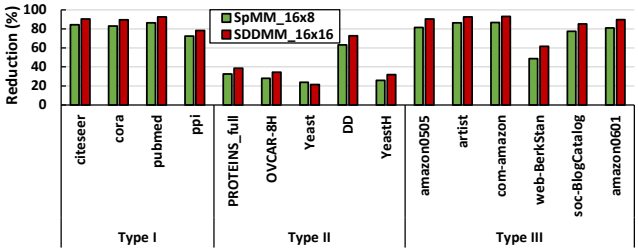

Figure 8: Effectiveness of Sparse Graph Translation.

in Figure 8, across all types of datasets, our SGT technique can significant reduce the number of traversed TCU-blocks. The major reason is that SGT can largely improve the density of non-zero elements within each TCU Block. In contrast, the graphs w/o SGT would demonstrate a large number of highly-sparse TCU blocks. What also worth noticing is that on the Type II graphs, such a reduction benefit is lower. This is because Type II graph consists of a set of small subgraphs that only maintain the intra-subgraph connections, which already maintains relatively dense columns within each TC block.

The size of hidden dimensions: In this experiment, we analyze the impact of the GNN model architecture in terms of the size of the hidden dimension for GCN. As shown in Figure 7(a), we observe that with the increase of hidden dimension of GCN, the running time of TC-GNN is also increased due to more computation (*e.g.*, additions) and memory operations (*e.g.*, data movements) during the aggregation phase and a larger size of node embedding matrix during the node update phase. Meanwhile, we also notice that TC-GNN can effectively handle higher hidden dimensions by launching more warps per thread-block. Therefore, the TCU computation can be efficiently parallelized across different dimensions.

The number of layers: In this experiment, we analyze the impact of the number of layers in terms of the size of the hidden dimension for GCN. As shown in Figure 7(b), we observe that with more number of layers, the running time of TC-GNN also increases correspondingly due to more computations and memory access. Meanwhile, we also notice that on datasets with larger number of nodes and edges such as *amazon*0505 that the layer increase will lead to a more pronounced time increase.

The number of warps per block: According to our profiling, loading the sparse graph data (*e.g.*, edge lists) from the global memory to initialize the TCU input matrix tiles would dominate the overall execution time. One effective way to reduce its impact is to improve the memory access parallelism by assigning more threads. Therefore, one key control variable in this study is the number of the virtual warps used in TC-GNN. Note that here we refer to "virtual" warps because all of them will be responsible for loading data but only part of them will actually take part in TCU computation. As shown in Figure 7(c), with the increase of the number of warps, the overall performance for training per epoch would first decrease due to the better parallelism for loading the graph data. However, the number of warps per block would decrease the overall performance under certain circumstances, such as the cases at 32. All three settings suffer from evident performance degradation. This is because the global memory access contention will become severe, thus, leading to lower execution performance. What also worth noticing is that different datasets would have different "optimal" choices of the warp-per-block parameter. For example, on the *com-amazon* dataset, 2 warps per block can deliver the best performance, while *amaon0505* requires 8 warps per block. Based on our large profiling and empirical study, we observe that the selection of this parameter should consider the average edge per row window (*avg.edges*), which can be easily get during the preprocessing. When we set the $warpPerBlock = \lfloor \frac{avg.edge}{32} \rfloor$, we can approach the optimal performance. For instance, the average edges per row window is 88 for *com-amazon*, it reaches the best performance when we have 2 warps per block.

Overhead Analysis: We further evaluate the overhead of our TC-aware sparse graph translation technique. Here we use the training for illustration, and the inference in real GNN application setting would also use the same graph structure many times [14, 17] while only changing the node embeddings input. As shown in Figure 7(d), its overhead is consistently tiny (average 4.43%) compared with overall training time. We thus conclude that such one-time overhead can be amortized during the GNN computation, which demonstrates its applicability in real-world GNN applications.

7 CONCLUSION

In this paper, we introduce, **TC-GNN**, the first GNN acceleration framework on TCU of GPUs. We introduce a novel sparse graph translation technique to gracefully fit the sparse GNN workload on dense TCU. Our TCU-tailored GPU kernel design maximizes the TCU performance gains for GNN computing through effective CUDA core and TCU collaboration and a set of memory/data flow optimizations. Our seamless integration with the popular Pytorch framework further facilitate the end-to-end GNN computing with high programmability. Extensive experiments show the performance advantage of TC-GNN over the state-of-the-art Deep Graph Library framework across various GNN models and datasets.

REFERENCES

[1] [n.d.]. *Intel Math Kernel Library. Reference Manual.* Intel Corporation. Santa Clara, USA.

[2] A. Abdelfattah, S. Tomov, and J. Dongarra. [n.d.]. Fast Batched Matrix Multiplication for Small Sizes Using Half-Precision Arithmetic on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS).*

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, PA.

[4] Wieb Bosma, John Cannon, and Catherine Playoust. 1997. The Magma algebra system. I. The user language. *J. Symbolic Comput.* (1997). Computational algebra and number theory (London, 1993).

[5] Hsinchun Chen, Xin Li, and Zan Huang. 2005. Link prediction approach to collaborative filtering. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL).* IEEE.

[6] De Cheng, Yihong Gong, Xiaojun Chang, Weiwei Shi, Alexander Hauptmann, and Nanning Zheng. 2018. Deep feature learning via structured graph Laplacian embedding for person re-identification. *Pattern Recognition* (2018).

[7] Alberto Garcia Duran and Mathias Niepert. 2017. Learning graph representations with embedding propagation. In *Advances in neural information processing systems (NeurIPS).*

[8] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. 2021. EGEMM-TC: Accelerating Scientific Computing Tensor Cores with Extended Precision. *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)* (2021).

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

[10] Matthias Fey and Jan E. Lenssen. 2019. PyTorch Extension Library of Optimized Scatter Operations. https://github.com/rusty1s/pytorch_scatter

[11] Jaume Gibert, Ernest Valveny, and Horst Bunke. 2012. Graph embedding in vector spaces by node attribute statistics. *Pattern Recognition* (2012).

[12] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[13] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM international conference on Knowledge discovery and data mining (SIGKDD).*

[14] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems (NeurIPS).*

[15] Riesen Kaspar and Bunke Horst. 2010. *Graph classification and clustering based on vector space embedding.* World Scientific.

[16] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. 2016. Benchmark Data Sets for Graph Kernels. http://graphkernels.cs.tu-dortmund.de

[17] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)* (2017).

[18] Jérôme Kunegis and Andreas Lommatzsch. 2009. Learning spectral graph transformations for link prediction. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML).*

[19] Süreyya Emre Kurt, Aravind Sukumaran-Rajam, Fabrice Rastello, and P. Sadayyappan. 2020. Efficient Tiled Sparse Matrix Multiplication through Matrix Signatures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20).* IEEE Press, Article 87, 14 pages.

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

[21] Ang Li and Simon Su. 2020. Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2020).

[22] Dijun Luo, Chris Ding, Heng Huang, and Tao Li. 2009. Non-negative laplacian embedding. In *2009 Ninth IEEE International Conference on Data Mining (ICDM).*

[23] Dijun Luo, Feiping Nie, Heng Huang, and Chris H Ding. 2011. Cauchy graph embedding. In *Proceedings of the 28th International Conference on Machine Learning.*

[24] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC).*

[25] Nvidia. [n.d.]. CUDA Sparse Matrix library (cuSPARSE). developer.nvidia.com/cusparse. developer.nvidia.com/cusparse

[26] Nvidia. [n.d.]. cuSPARSE Blocked SpMM. https://developer.nvidia.com/blog/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/

[27] Nvidia. [n.d.]. Dense Linear Algebra on GPUs. developer.nvidia.com/cublas. developer.nvidia.com/cublas

[28] NVIDIA. [n.d.]. Improved Tensor Core Operations. https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html#tensor-operations. https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html#tensor-operations

[29] Nvidia. [n.d.]. NVIDIA Blocked-Sparse API. https://docs.nvidia.com/cuda/cusparse/index.html#cusparse-generic-function-spmm). https://docs.nvidia.com/cuda/cusparse/index.html#cusparse-generic-function-spmm)

[30] Nvidia. [n.d.]. NVIDIA Volta. https://en.wikipedia.org/wiki/Volta_(microarchitecture). https://en.wikipedia.org/wiki/Volta_(microarchitecture)

[31] NVIDIA. [n.d.]. TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x. blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/.

[32] Nvidia. [n.d.]. Warp Matrix Multiply-Accumulate (WMMA). docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma. docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma

[33] NVIDIA. 2017. Programming Tensor Cores in CUDA 9. https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS).*

[35] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD).*

[36] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. 2018. Attention-based graph neural network for semi-supervised learning. (2018).

[37] Tomasz Tylenda, Ralitsa Angelova, and Srikanta Bedathur. 2009. Towards time-aware link prediction in evolving social networks. In *Proceedings of the 3rd workshop on social network mining and analysis.*

[38] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations (ICLR).*

[39] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019). https://arxiv.org/abs/1909.01315

[40] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations (ICLR).*

[41] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. 2018. Hierarchical Graph Representation Learning with Differentiable Pooling. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS).*