TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs

1 Introduction

Over the recent years, with the increasing popularity of graph-based learning, graph neural networks (GNNs) [13, 28, 33] become dominant in the computing of essential tasks across various domains, including e-commerce, financial services, *etc*. Compared with standard methods for graph analytics, such as random walk [9,11,27] and graph laplacians [3,16,17], GNNs highlight themselves with significantly higher accuracy [13,30,33] and better generality [10]. From the computation perspective, GNNs feature an interleaved execution phase of both graph operations (scatter-and-gather [8]) 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 [31] and PyTorch Geometric [6], 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 strategy is to incorporate sparse primitives (such as cuSPARSE [20]) 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 [21]. Moreover, cuSPARSE is designed to only utilize CUDA cores. 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 TCUs 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 cores. 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 TCUs to process non-zero matrix tiles of the sparse graph adjacency matrix would still waste most of the TCU computation and memory access efforts. This is 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***, although the recent CUDA release update enables TCUs to exploit the benefit of certain types of sparsity [19], 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 in real-world GNN applications.

To this end, we introduce, **TC-GNN**, the first TCU-based GNN acceleration design on GPUs. Our key insight is to *let the sparse input graph fit the dense computation of TCUs*. ***At the input level***, instead of exhaustively traversing all sparse matrix tiles and determining 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 number 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 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 GPU kernel level***, for efficiently process-

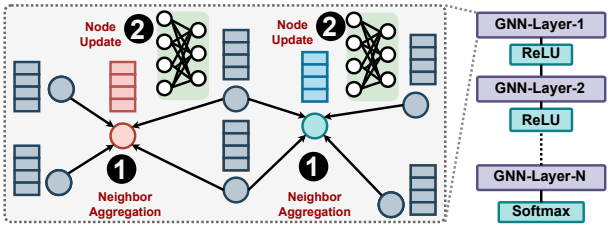

Figure 1: GNN General Computation Flow.

ing 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 [26] framework. Thereby, users only need to interact with their familiar PyTorch programming environment by using TC-GNN APIs. This can significantly reduce extra learning efforts meanwhile improving user productivity and code portability across different platforms.

2 Background and Related Work

2.1 Graph Neural Networks

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

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

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 [10, 13] just rely on the neighboring nodes while others [30] 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 parameters, respectively. The common choices for node embedding dimensions are 16, 64, and 128, and the embedding dimension may change across different layers. After several iterations of aggregation and update (*i.e.*, several GNN layers),

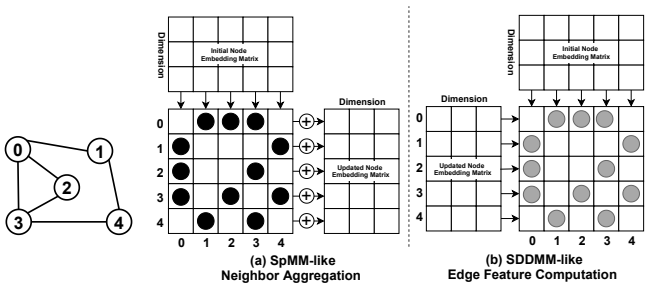

Figure 2: (a) SpMM-like and (b) SDDMM-like Operation in GNNs. Note that "$\rightarrow$" indicates loading data; "$\oplus$" indicates neighbor embedding accumulation.

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 [4, 7, 12] and link prediction [2, 14, 29].

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 [20]) in many state-of-the-art GNN frameworks [31, 32]. 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 Sampled Dense-Dense Matrix Multiplication (SDDMM)-like operations (Equation 3 and Figure 2(b)).

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

The computation of F is optional in GNNs, which is generally adopted by Attention-based Graph Neural Network [28] for identifying more complicated graph structural information.

2.2 GPU Tensor Core

In the most recent GPU architectures (since Volta [23]), 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 units, respectively.

Different from scalar computation on CUDA cores, TCU provides tile-based matrix-matrix computation primitives on

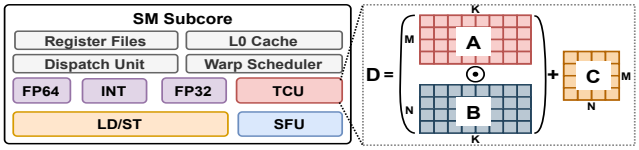

Figure 3: A Subcore of GPU SM with TCUs.

Listing 1: WMMA APIs for TCUs in CUDA C.

```
wmma::fragment<matrix_a, M, N, K, tf32, row_major> a_frag;
// Load tiles (global/shared mem. -> register fragments).
wmma::load_matrix_sync(a_frag, A, M);
// Execute GEMM on loaded tiles on register fragments.
wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
// Move results (register fragments -> global/shared mem).
wmma::store_matrix_sync(C, c_frag, N, mem_row_major);
```

register fragments, which can deliver more than 10× throughput improvement. In particular, TCU supports 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 type of precision (*e.g.*, TF-32), while $\mathbf{C}$ and $\mathbf{D}$ are stored in FP32. Depending on the data precision and GPU architecture version, the matrix size (MMA shape) of $\mathbf{A}(M \times K)$, $\mathbf{B}(K \times N)$, and $\mathbf{C}(M \times N)$ should follow some principles [22]. For example, TF-32 TCU computing requires $M = N = 16$ and $K = 8$. In the recent CUDA release (>=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 NVIDIA's in-depth studies [24]. TCU can be utilized in several ways. The simplest way is to call cuBLAS [21] by using the `cublasSgemmEX` API. The second way is to call the Warp Matrix Multiply-Accumulate (WMMA) (`nvcuda::wmma`) API [25] in CUDA C to operate TCUs directly with four major operations (Listing 1). Since the appearance of the TCU, research efforts have been devoted to accelerating high-performance computing workloads with TCUs [1, 5, 15]. These prior efforts use the TCUs in the dense applications that TCU is initially designed for, while TC-GNN jumps out of the scope defined by TCU designers and accelerates the sparse GNN operations using TCUs.

3 TC-GNN

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 effective graph structural manipulation meanwhile guaranteeing output correctness*. Specifically, we condense the highly-scattered neighbor ids without losing key information (*e.g.*, edge connections). As 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 (in the red-colored rectangular box) to build TCU blocks (*TC_block*) (*a.k.a.*, the input operand shape of a single MMA instruction), in the orange-colored rectangular box. In this paper, we demonstrate the use of standard MMA shape for TF-32 of TCU on Ampere GPU architecture, and other MMA shapes [22] can be used for different computation precision (*e.g.*, `half`) and GPU architecture (*e.g.*, Turing).

Algorithm 1: TCU-aware Sparse Graph Translation.

input : Graph adjacent matrix $\mathbf{A}$ (*nodePointer*, *edgeList*).
output : Result of *winPartition* and *edgeToCol*.

```
/* Compute the total number of row windows.          */
numRowWin = ceil(numNodes/winSize);
for winId in numRowWin do
    /* 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(eArray);
    /* #TC blocks in the current rowWindow.          */
    winPartition[winId] =
      ceil(eArrClean.size/TC_BLK_w);
    /* Edges-to-columnID mapping in TC Blocks.       */
    for eIndex in [winStart, winEnd] do
        eid = edgeList[eIndex];
        edgeToCol[eIndex] = eArrClean[eid];
    end
end
```

SGT takes several steps for processing each row window, as detailed in Algorithm 1 and visualized in Figure 4(c). 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 the 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. 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)). SGT is applicable for both the SpMM and SDDMM in GNN sparse operations and can be easily parallelized because the processing of individual row windows is independent. In most cases, SGT only needs to execute once and its result can be reused across many epochs/rounds.

TCU-tailored GNN Computation 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 [31]. And they employ the cuSPARSE [20] on CUDA cores 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 highly irregular SpMM as

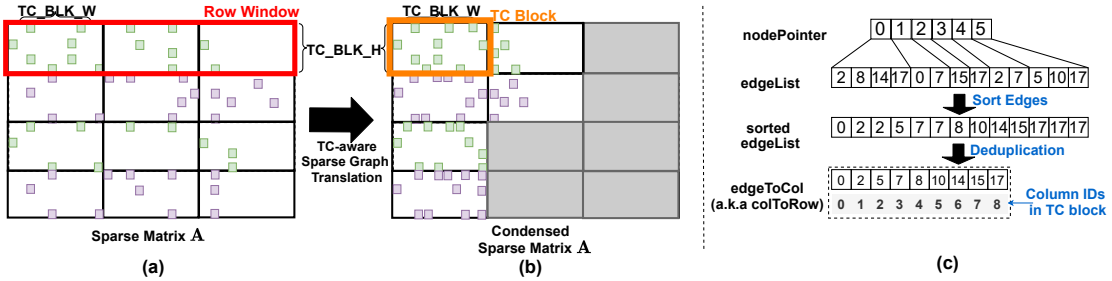

Figure 4: Illustration of Sparse Graph Translation. Note that the grey area indicates the TCU block that will be directly skipped.

Algorithm 2: TC-GNN Neighbor Aggregation.

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

```
/* Traverse through all row windows.                          */
for winId in numRowWindows do
    /* #TC blocks of the row window.                          */
    numTCblocks = winPartition[winId] ;
    /* Edge range of TC blocks of the row window.             */
    edgeRan = GetEdgeRange(nodePointer, winId);
    for TCblkId in numTCblocks do
        /* The edgeList chunk in current TC block.            */
        edgeChunk = GetChunk(edgeList, edgeRan, TCblkId);
        /* 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 TCU GEMM.                     */
        XnewTile = TCcompute(ATile, XTile);
        /* Store XnewTile of X̂.                               */
        X̂ = StoreDense(XNewTile, winId, colId);
    end
end
```

Algorithm 3: TC-GNN Edge Feature Computation.

input : Condensed graph structural information (*nodePointer*, *edgeList*, *edgeToCol*, *winPartition*) and node embedding matrix ($\mathbf{X}$).
output : Edge Feature List (*edgeValList*).

```
/* Traverse through all row windows.                          */
for winId in numRowWin do
    /* #TC blocks in the row window.                          */
    numTCblocks = winPartition[winId] ;
    /* Edge range of TC blocks of the row window.             */
    edgeRan = GetEdgeRange(nodePointer, winId);
    for TCblkId in numTCblocks do
        /* EdgeList chunk in current TC block.                */
        edgeChunk = GetChunk(edgeList, edgeRan, TCblkId);
        /* 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 TCU GEMM.                  */
        edgeValTile = TCcompute(XTile_A, XTile_B);
        /* Store edgeValTile to edgeValList.                  */
        StoreSparse(edgeValList, edgeValTile,
                    edgeList, edgeToCol);
    end
end
```

dense GEMM computation and solving it on TCU effectively (Algorithm 2). Previous research [28, 30] has also demonstrated the great importance of incorporating the edge feature for a better GNN model algorithmic performance (*e.g.*, accuracy). The underlying building block to generate edge features is the SDDMM-like operation. TC-GNN supports SDDMM with the collaboration of the above sparse graph translation and TCU-tailored algorithm design (Algorithm 3).

Two-level Workload Mapping Different from previous work [6, 31] focusing on CUDA cores 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 (§12) to one thread block. The number of threads in each block should be divisible by the number of threads in each warp (32) for better performance. Once threads running on CUDA cores 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).

TCU-optimized Dataflow Design 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. As visualized in Figure 5(a) for neighbor aggregation, shared memory is mainly used for

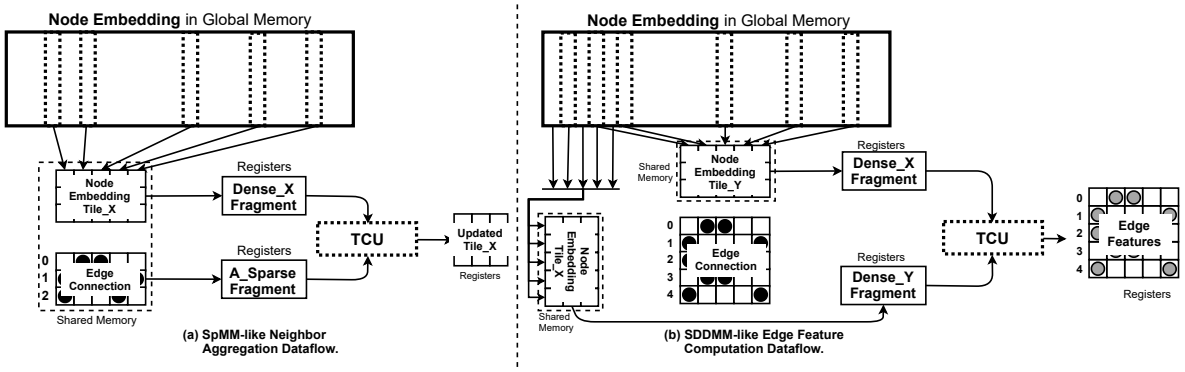

Figure 5: TCU-optimized Dataflow Design for (a) Neighbor Aggregation and (b) Edge Feature Computing in GNNs.

Table 1: Datasets for evaluation.

Type	Dataset	Abbr.	#Vertex	#Edge	Dim.	#Class
I	Citeseer	CR	3,327	9,464	3703	6
	Cora	CO	2,708	10,858	1433	7
	Pubmed	PB	19,717	88,676	500	3
	PPI	PI	56,944	818,716	50	121
II	amazon0505	AZ	410,236	4,878,875	96	22
	artist	AT	50,515	1,638,396	100	12
	com-amazon	CA	334,863	1,851,744	96	22
	soc-BlogCatalog	SC	88,784	2,093,195	128	39
	amazon0601	AO	403,394	3,387,388	96	22

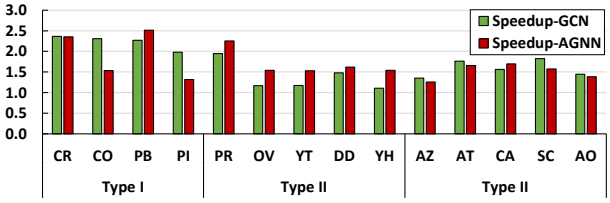

Figure 6: Speedup over DGL on GCN and AGNN.

caching several most frequently used information, including the tile of sparse matrix A (`sparse_A`), the column-id of the sparse matrix **A** to row-id of node embedding matrix **X** (`sparse_AToX_index`), and the dense tile of **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*. Similar to the shared memory design in neighbor aggregation, for edge feature computing, as visualized in Figure 5(b), 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**.

4 Evaluation

We choose two representative GNN models widely used by previous work [6, 18, 31] on *node classification* tasks. Specifically, 1) Graph Convolutional Network (**GCN**) [13] is one of the most popular GNN model architectures. We use the setting: *2 layers with 16 hidden dimensions per layer*; 2) Attention-based Graph Neural Network (**AGNN**) [28]. AGNN differs from GCN in its aggregation function, which compute edge feature (via embedding vector dot-product between source and destination vertices) before the node aggregation. For AGNN, we use: *4 layers with 32 hidden dimensions per layer*. We choose Deep Graph Library (**DGL**) [31] as our baseline, which is the state-of-the-art GNN framework on GPUs. We cover two types of datasets (Table 1) from previous GNN-related work [6, 18, 31]. TC-GNN backend is implemented with C++ and CUDA C, and the 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. We calculate the averaged latency of 200 end-to-end runs.

Figure 6 shows that TC-GNN achieves 1.70× speedup on average compared to DGL over two types of datasets across GCN and AGNN model on end-to-end training. The performance improvements against DGL are significantly higher for GCN (on average 2.23×) compared to AGNN (on average 1.93×) on type I graphs. 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 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 (SDDMM) is more sensitive to the irregular sparse graph structure because of much more intensive computations and memory access. Thus, the performance improvement is relatively lower. The speedup is also evident (on average 1.59× for GCN and average 1.51× for AGNN) on Type II 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.

References

[1] A. Abdelfattah, S. Tomov, and J. Dongarra. Fast batched matrix multiplication for small sizes using half-precision arithmetic on gpus. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

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

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

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

[5] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. Egemm-tc: Accelerating scientific computing tensor cores with extended precision. *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, 2021.

[6] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds (ICLR)*, 2019.

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

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

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

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

[11] Zexi Huang, Arlei Silva, and Ambuj Singh. A broader picture of random-walk based graph embedding. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 685–695, 2021.

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

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

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

[15] Ang Li and Simon Su. Accelerating binarized neural networks via bit-tensor-cores in turing gpus. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2020.

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

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

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

[19] Nvidia. Accelerating matrix multiplication with block sparse format and nvidia tensor cores. developer.nvidia.com/blog/accelerating-matrix-multiplication-with\-block-sparse-format-and-nvidia-tensor-cores/.

[20] Nvidia. Cuda sparse matrix library (cusparse). developer.nvidia.com/cusparse.

[21] Nvidia. Dense linear algebra on gpus. developer.nvidia.com/cublas.

[22] NVIDIA. Improved tensor core operations. https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html#tensor-operations.

[23] Nvidia. Nvidia volta. https://en.wikipedia.org/wiki/Volta_(microarchitecture).

[24] NVIDIA. Tensorfloat-32 in the a100 gpu accelerates ai training, hpc up to 20x.

[25] Nvidia. Warp matrix multiply-accumulate (wmma). docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma.

[26] 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. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

[27] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2014.

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

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

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

[31] 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. Deep graph library: Towards efficient and scalable deep learning on graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[32] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An efficient runtime system for gnn acceleration on gpus. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, 2021.

[33] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.