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

Yuke Wang, Boyuan Feng, Zheng Wang,
Guyue Huang, Yufei Ding

Department of Computer Science
University of California, Santa Barbara

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Graphs are Everywhere, GNNs are Useful Hammer!

Social Networks

Financial Services

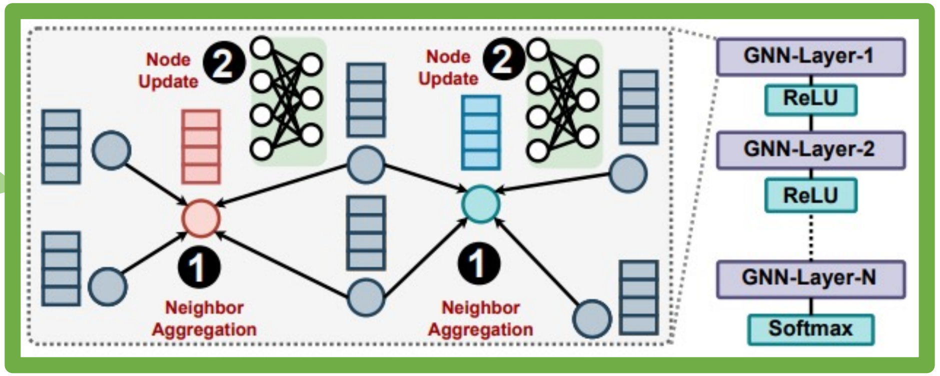
Power Grid

Molecular Biology

Image Credit: Google

Background

- **Graph Neural Network Basics.**



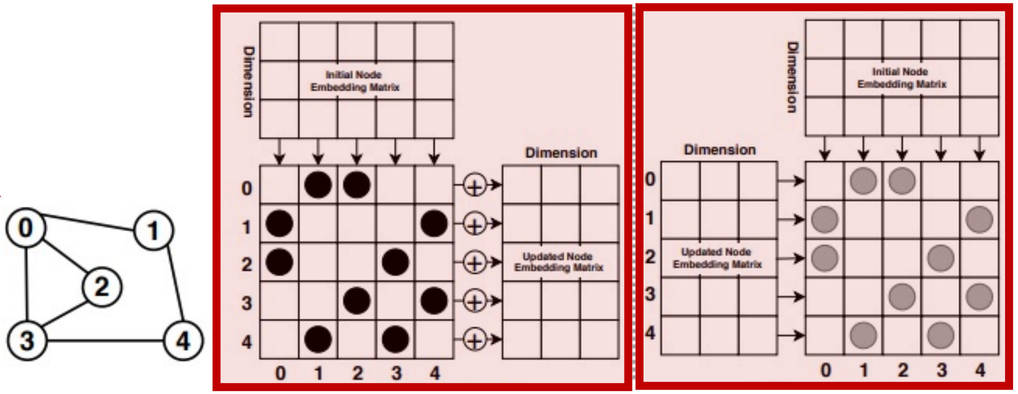
$$a_v^{(k+1)} = \boldsymbol{Aggregate}^{(k+1)}(h_u^{(k)} | u \in \mathrm{N}(v) \cup h_v^{(k)})$$
$$h_v^{(k+1)} = \boldsymbol{Update}^{(k+1)}(a_v^{(k+1)})$$

- **Basic computation in GNNs.**
  - Neighbor aggregation (SpMM-like).

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

  - Edge feature computation (SDDMM-like).

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



Background

- **GPU Tensor Cores (TCs).**
  - TC supports the compute primitive of D = A × B + C.
  - Matrix tile A, B and C are certain precision (e.g., tf-32, fp-16)

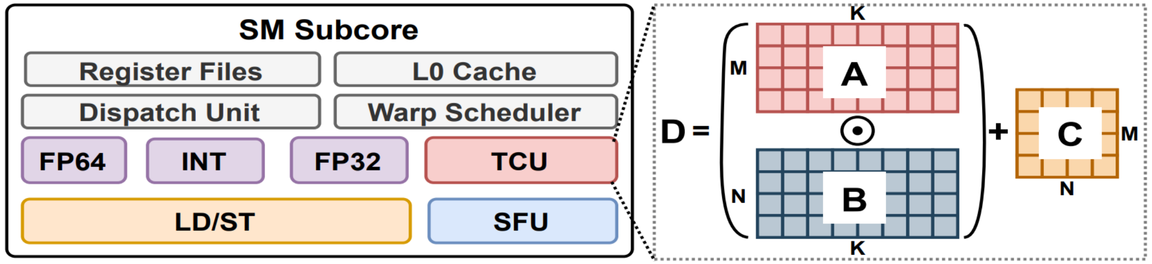


Figure 3: A Subcore of GPU SM with TCUs.

- **Programming of TCs.**
  - cuBLAS cublasSgemmEX APIs with limited precision option (e.g., INT8, FP16)
  - Warp Matrix Multiply-Accumulate (WMMA) (nvcuda::wmma) API in CUDA C.

Listing 1. Basic WMMA APIs for TCU in CUDA C.

```
// define the register fragment for matrix A (1-bit).
wmma::fragment<matrix_a, M, N, K, b1, row_major> a_frag;
// load a tile of matrix A to register fragment.
wmma::load_matrix_sync(a_frag, A, M);
// matrix-matrix multiplication (1-bit x 1-bit -> 32-bit)
wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
// store the C matrix tile from register to matrix C.
wmma::store_matrix_sync(C, c_frag, N, mem_row_major);
```

Challenges

- Existing deep-learning frameworks are optimized for dense neural network operations.

Lack of efficient support for sparse graph neural network computation.

- Existing major sparse computation kernels (e.g., cuSPARSE) leverage CUDA cores.

Underutilize the latest GPU with new hardware feature that can offer high-performance computation.

- Existing Tensor-Core based kernels (e.g., Block-SpMM) rely on rigid input sparsity pattern (e.g., block sparsity).

Limits its applicability towards different sparse inputs settings.

Motivations

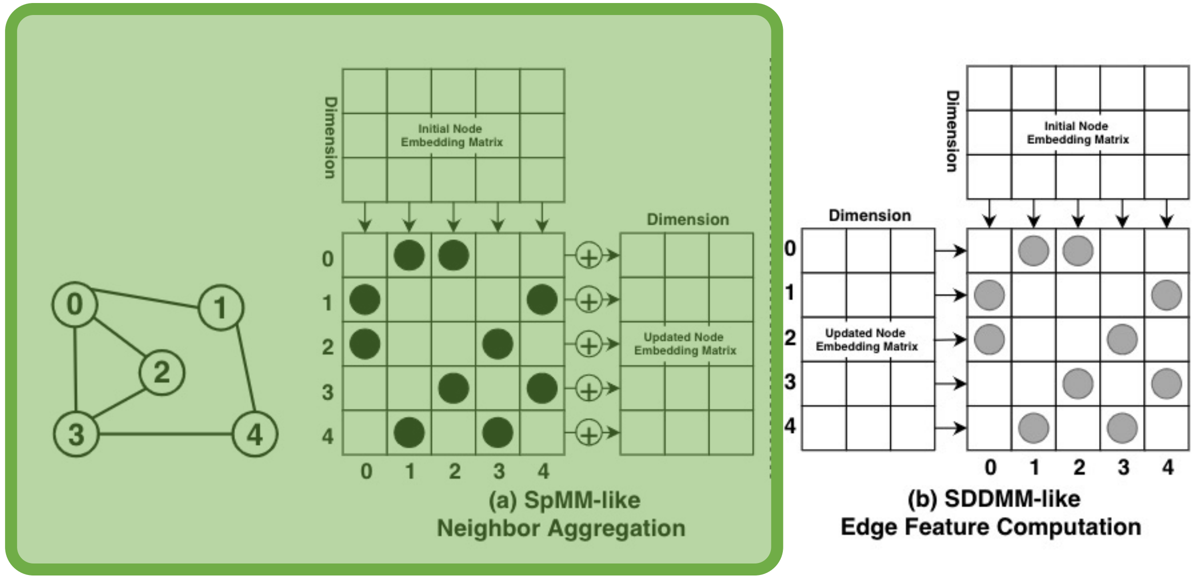


Figure 2. SpMM-like and SDDMM-like Operation in GNNs. Note that "→" indicates loading data; "⊕" indicates neighbor embedding accumulation.

Apply separate optimization on one direction only would hardly work ☹

GPUs fit GNNs

Sparse MM on CUDA core

Profiling of GCN Sparse Operations.

Dataset	Aggr. (%)	Update (%)	Cache(%)	Occ.(%)
Cora	88.56	11.44	37.22	15.06
Citeseer	86.52	13.47	38.18	15.19
Pubmed	94.39	5.55	37.22	16.24

GNNs fit GPUs

Dense MM on CUDA core

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%

Question:

How could we match the sparse GNN workload with GPUs to achieve high computation efficiency and better utilization of GPU resources?

TC-GNN Overview

- The first TC-based GNN acceleration design on GPUs.
- At the input level technique.
- At the kernel level innovation.
- At the framework level design.

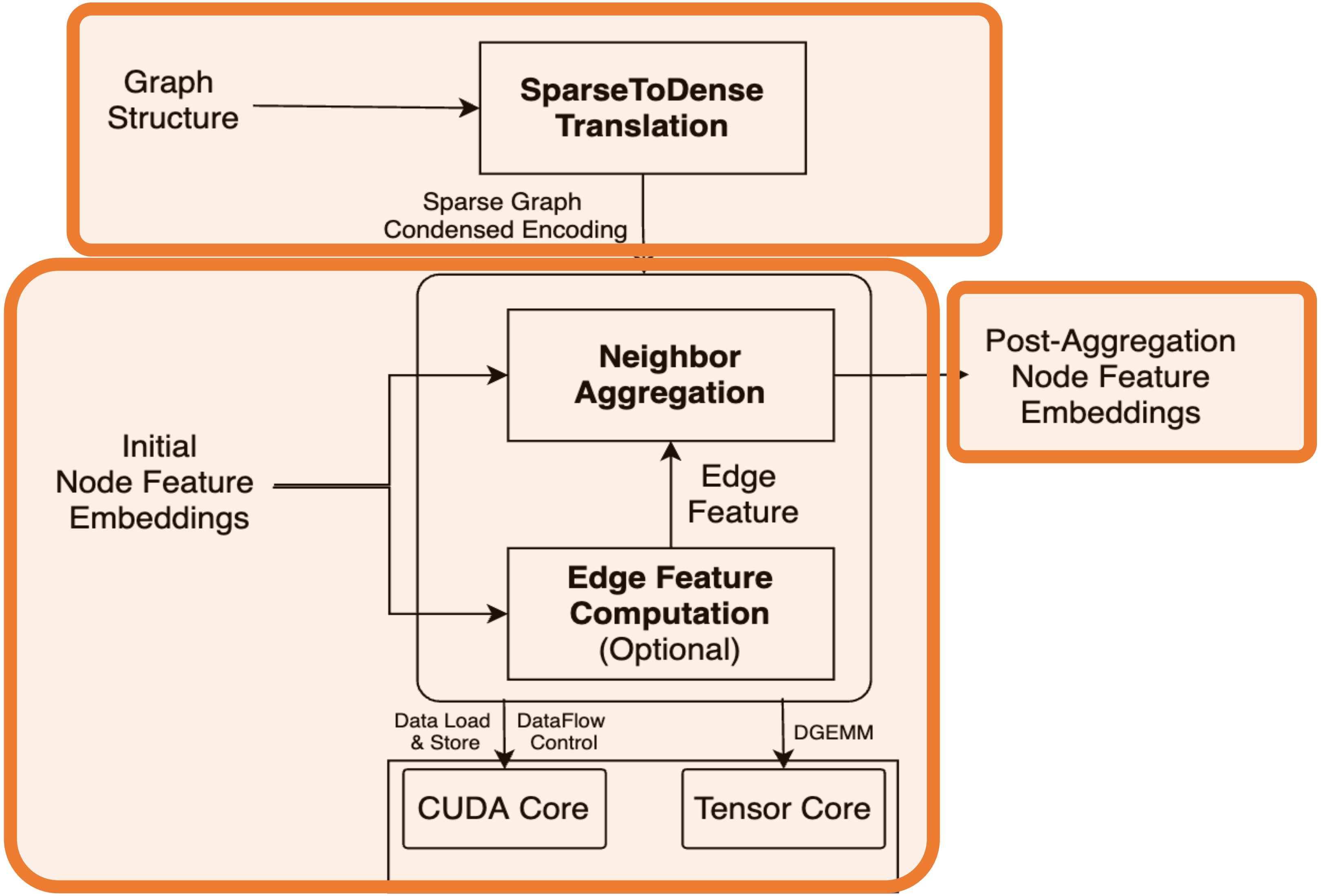
"Let the input sparse graph fit the dense computation of Tensor Core"

Sparse graph translation (SGT) technique condense non-zero elements from sparse tiles into a fewer number of "dense" tiles

TC-GNN exploits the benefits of CUDA core and tensor core collaboration.

TC-GNN integrates with the popular Pytorch framework.

Overall Design



Overall Design

Load TC-GNN Module

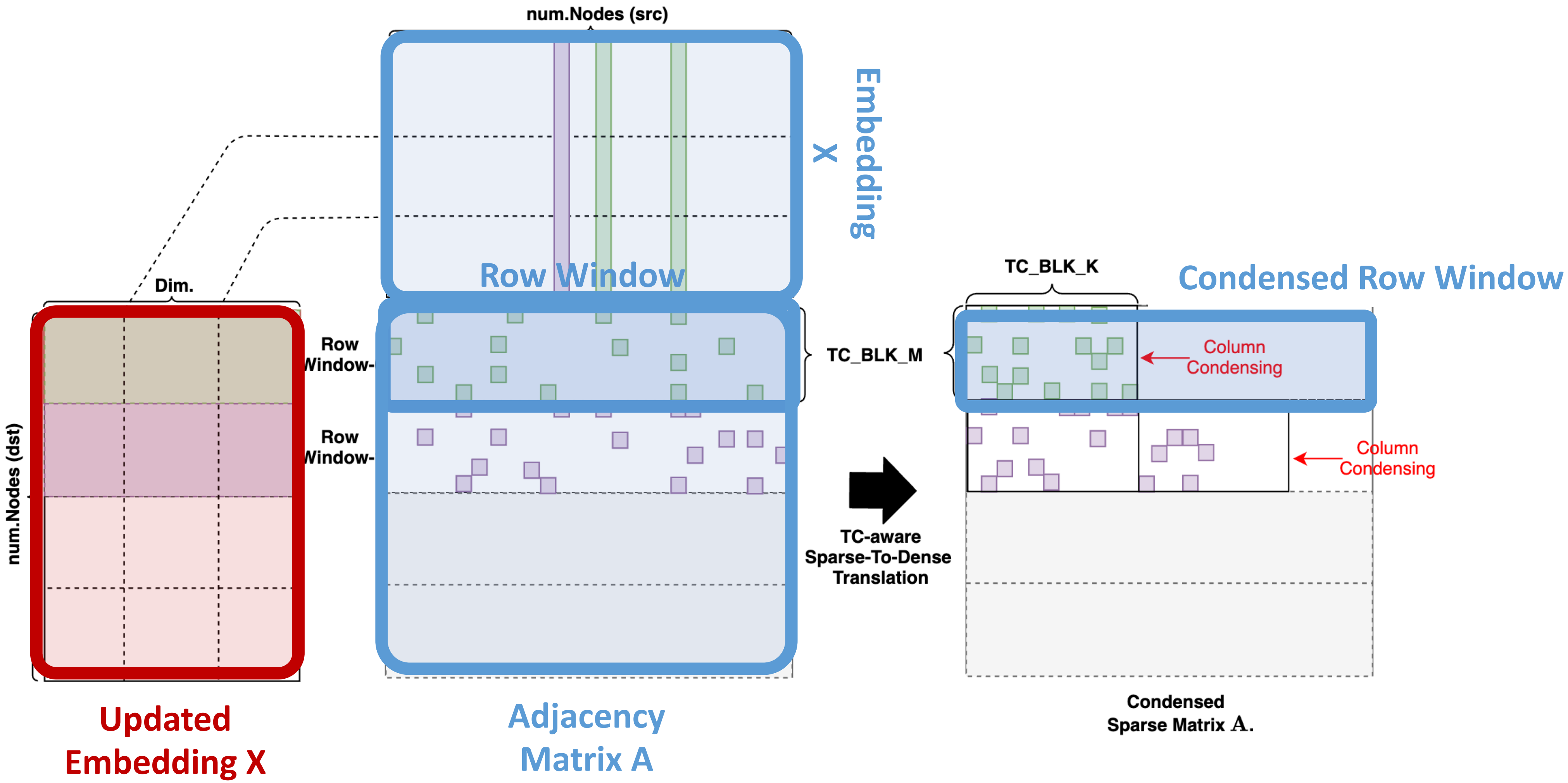
```
import TCGNN, torch
# include other packages ...
class GCN(torch.nn.Module):
    def __init__(self, inDim, hiDim, outDim):
        self.layer1 = TCGNN.GCNConv(inDim, hiDim)
        self.layer2 = TCGNN.GCNConv(hiDim, outDim)
        self.softmax = torch.nn.Softmax()

    def forward(self, tiledGraph, param):
        tiled_adj, X = tiledGraph.adj, tiledGraph.X
        X = self.layer1(X, tiledAdj, param)
        X = self.ReLU(X)
        X = self.layer2(X, tiledAdj, param)
        X = self.softmax(X)
        return X
# Define a two-layer GCN model in TC-GNN.
model = GCN(inDim=100, hiDim=16, outDim=10)
# Load graph and extract input information
rawGraph, info = TCGNN.Loader(graphFilePath)
# Generate TCU tile and runtime configuration.
tiledGraph, config = TCGNN.Preprocessor(rawGraph, info)
# Run model through forward computation.
predict_y = model(tiledGraph, config)
# Compute loss and accuracy.
# Gradient backpropagation for training.
```

Post-Aggregation Node Feature Embeddings

Load graphs by TC-GNN

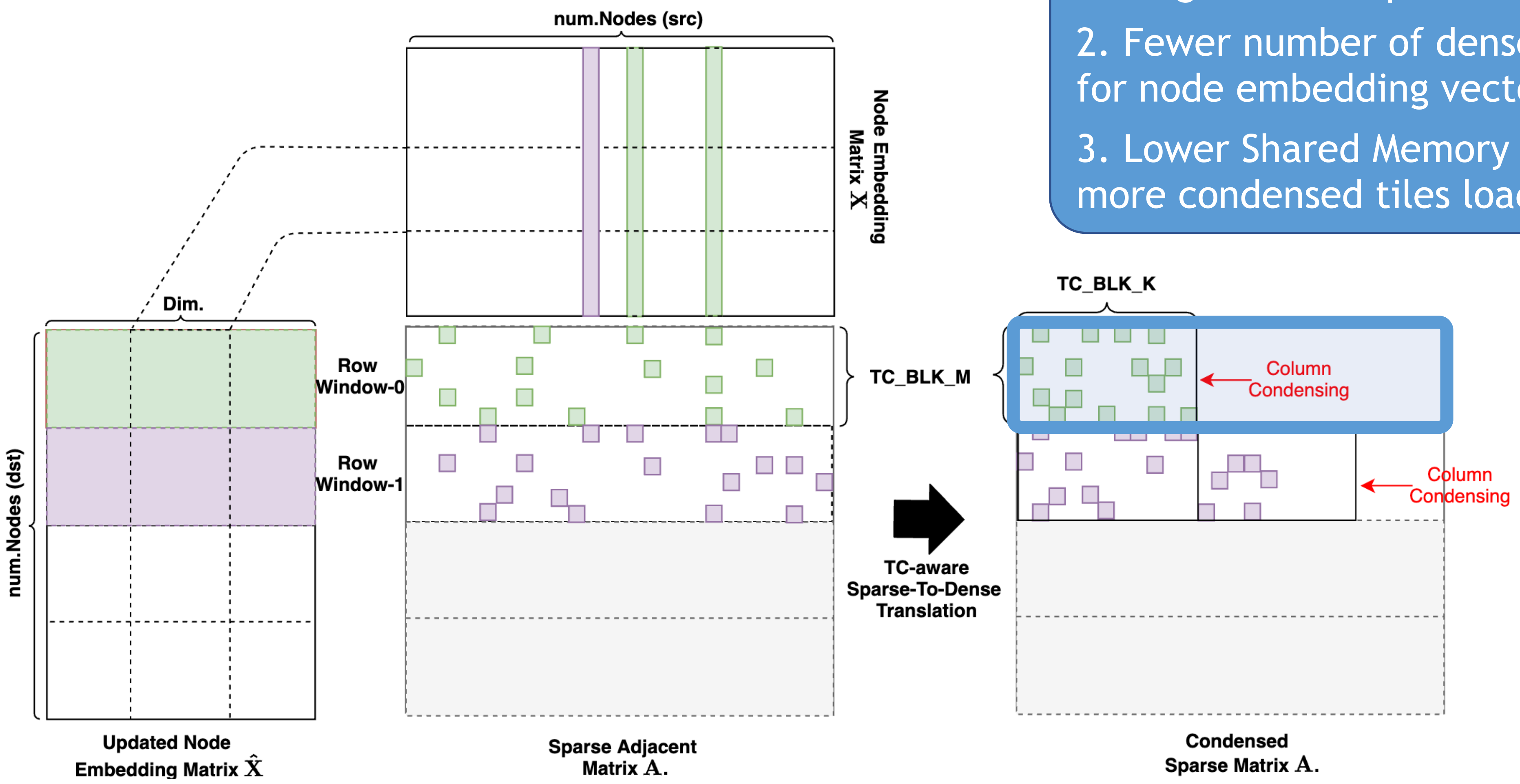
Preprocess graphs by TC-GNN

Sparse Graph Translation



Sparse Graph Translation

1. Fewer number of iterations for Calling TC WMMA primitives.
2. Fewer number of dense row access for node embedding vector.
3. Lower Shared Memory Usage due to more condensed tiles loading.

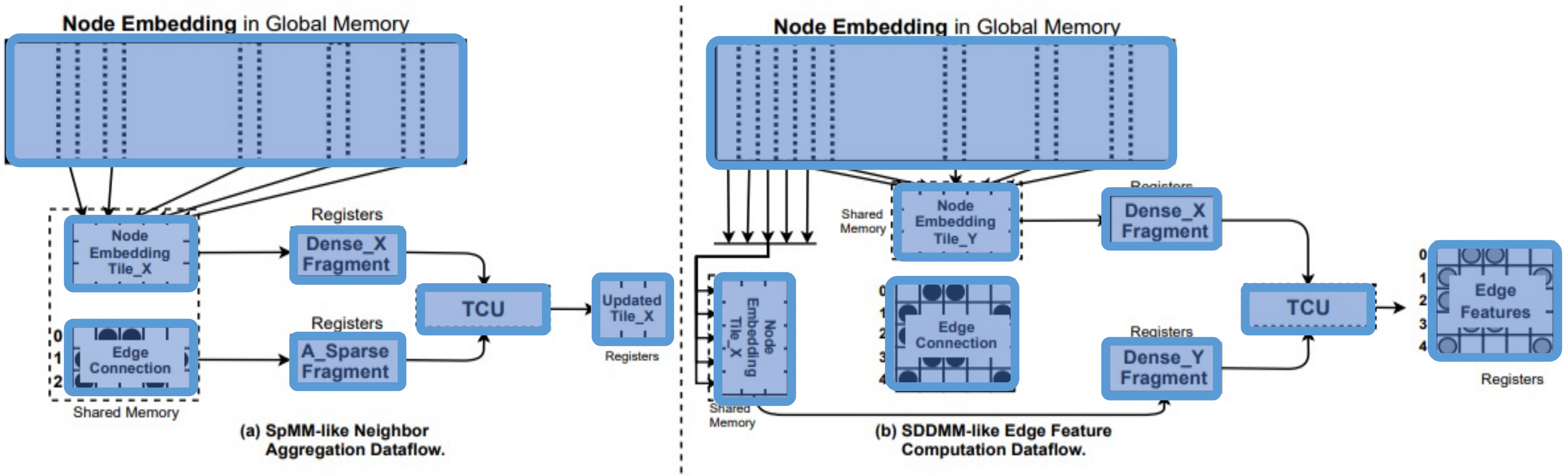


TC-aware Sparse Graph Translation

Algorithm 1: TCU-aware Sparse Graph Translation.

```
input   :Graph adjacent matrix A (nodePointer, edgeList).
output  :Result of winPartition and edgeToCol.
1
/* 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);
2
    /* Deduplicate edges of the current rowWindow.     */
    eArrClean = Deduplication(nIdArray);
    /* #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];
3
        edgeToCol[eIndex] = eArrClean[eid];
    end
end
```

TC-optimized Dataflow



TC-Optimized Dataflow Design for (a) Neighbor Aggregation and (b) Edge Feature Computing in GNNs

TC-tailored SpMM

Algorithm 2: TC-GNN Neighbor Aggregation.

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

Block

Warp

```
/* 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
        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
```

TC-tailored SDDMM

Algorithm 3: TC-GNN Edge Feature Computation.

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

Block

Warp

```
/* 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 blo
        edgeChunk = GetChunk(edgeList, edgeRan, T
        /* 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
```

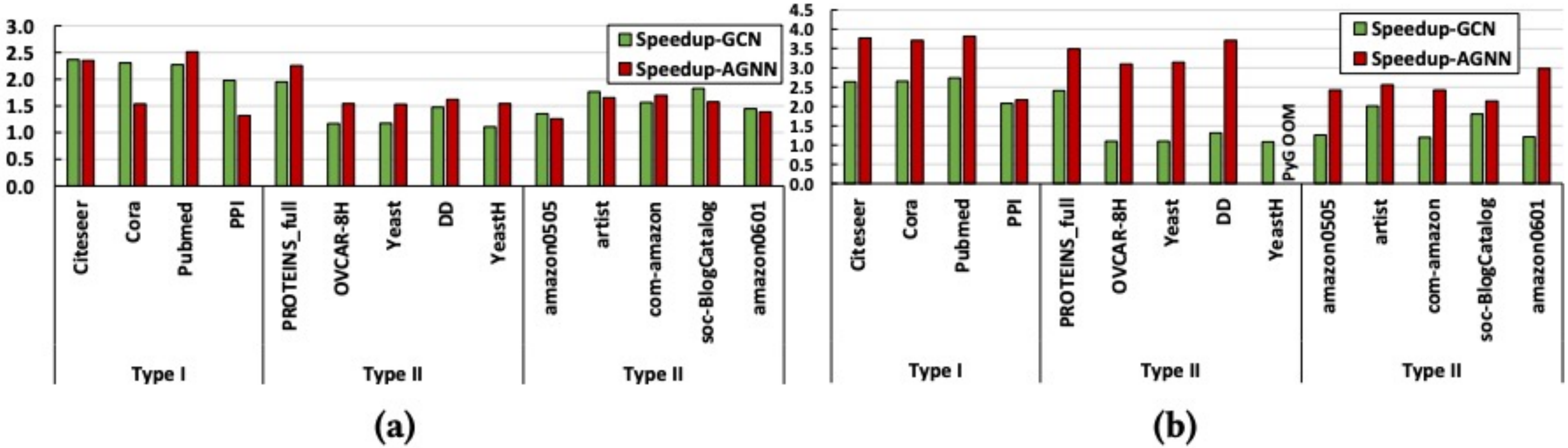
Evaluation

- Baseline:
  - Deep Graph Library (DGL)
  - PyTorch Geometric (PyG)
- GNN model:
  - GCN (Graph Convolutional Network)
  - AGNN (Attention-based GNN)
- Platform:
  - A desktop server with 8-core 16-thread Intel Xeon Silver 4110 CPU (64GB host memory) and NVIDIA RTX3090 GPU (24GB device memory)

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

End-to-end Performance: DGL & PyG



Speedup over (a) DGL and (b) PyG on GCN and AGNN.

Avg: 1.70X

Operator Performance (dgl.op)

- SpMM (**dgl.op.copy_u_sum**)

	dgl.op (ms)	TC-GNN (ms)
PROTEINS_full	0.088	0.044
OVCAR-8H	1.295	1.018
Yeast	1.183	0.862
DD	0.454	0.287
SW-620H	1.291	1.018

Avg: 1.50X

- SDDMM (**dgl.op.u_dot_v**)

	dgl.op (ms)	TC-GNN (ms)
PROTEINS_full	0.062	0.019
OVCAR-8H	0.466	0.054
Yeast	0.401	0.051
DD	0.170	0.026
SW-620H	0.476	0.055

Avg: 6.98X

Kernel Performance (cuSPARSE)

- SpMM w.r.t cuSPARSE with different embedding dimension. (GFLOPS)

	D (16)		D (32)		D (64)	
	cuSPARSE	TC-GNN	cuSPARSE	TC-GNN	cuSPARSE	TC-GNN
PROTEINS_full	90.13	130.89	170.55	226.63	276.46	348.73
OVCAR-8H	135.26	143.54	237.81	239.05	237.96	340.02
Yeast	135.42	157.97	238.12	261.76	230.25	366.61
DD	156.17	207.04	309.67	350.57	467.94	498.02
SW-620H	135.22	143.56	237.72	239.17	238.13	340.04

Avg: 1.23X

Future Works

- GPU-accelerated Preprocessing.
  - Current version is based on CPU + OpenMP parallel.
  - Intra-warp/block sorting for variable length edge list is needed (may use CUB library for fixed-length array sorting + padding).
- Support/optimization for multiple precision TC.
  - Current version is using TF32 on Ampere with WMMA shape of 16x8x16.
  - Adaptive optimization for different inputs settings (graph/dimension) when multiple WMMA shape available (e.g., FP16 with 16x8x8 and 16x8x16).
- Kernel Fusion with other layers.
  - Current version focuses on training.
  - More fusion operation in inference, such as Graphconv+BatchNorm.

$$\mathbf{BN}(x_{i,j}) = \left(\frac{x_{i,j} - \mathbb{E}[x_{*,j}]}{\sqrt{Var[x_{*,j}] + \epsilon}}\right) \cdot \gamma_j + \beta_j$$

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Thank You

yuke_wang@cs.ucsb.edu

GitHub https://github.com/YukeWang96/TC-GNN_ATC23.git