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
Graphs are Everywhere, GNNs are Useful Hammer!

Social Networks

Financial Services

Power Grid

Molecular Biology
Background

- **Graph Neural Network Basics.**
  
  \[
  a_v^{(k+1)} = \text{Aggregate}^{(k+1)}(h_u^{(k)}) | u \in N(v) \cup h_v^{(k)} \\
  h_v^{(k+1)} = \text{Update}^{(k+1)}(a_v^{(k+1)})
  \]

- **Basic computation in GNNs.**
  - Neighbor aggregation (SpMM-like).
    \[
    \hat{X} = (F_{N \times N} \odot A_{N \times N}) \cdot X_{N \times D}
    \]
  - Edge feature computation (SDDMM-like).
    \[
    F = (X_{N \times D} \cdot X_{N \times D}^T) \odot A_{N \times N}
    \]
Background

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

• 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.

Figure 3: A Subcore of GPU SM with TCUs.

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

```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.
- Existing major sparse computation kernels (e.g., cuSPARSE) leverage CUDA cores.
- Existing Tensor-Core based kernels (e.g., Block-SpMM) rely on rigid input sparsity pattern (e.g., block sparsity).

Lack of efficient support for sparse graph neural network computation.

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

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
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.
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.
Sparse Graph Translation

Updated Embedding X

Adjacency Matrix A

Condensed Row Window

TC-aware Sparse-To-Dense Translation

Column Condensing

Condensed Sparse Matrix A.
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.
Algorithm 1: TCU-aware Sparse Graph Translation.

```plaintext
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);
     /* 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];
       edgeToCol[eIndex] = eArrClean[eid];
   end
```

TC-aware Sparse Graph Translation
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.

```c
/* 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 = InitSparse(edgeChunk, winId);
      /* Compute XnewTile via TCU GEMM. */
      XnewTile = TCompute(ATile, XTile);
      /* Store XnewTile of X. */
      X̂ = StoreDense(XnewTile, winId, colId);
   end
end
```

TC-tailored SDDMM

Algorithm 3: TC-GNN Edge Feature Computation.

```c
/* Traverse through all row windows. */
for winId in numRowWIndows 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 (XTileA). */
      XTileA = FetchDenseRow(winId, TCblkId, X);
      /* Fetch a dense tile (XTileB). */
      XTileB = FetchDenseCol(colToNId, edgeToCol, X);
      /* Compute edgeValTile via TCU GEMM. */
      edgeValTile = TCompute(XTileA, XTileB);
      /* 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.

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

<table>
<thead>
<tr>
<th></th>
<th>dgl.op (ms)</th>
<th>TC-GNN (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROTEINS_full</td>
<td>0.088</td>
<td>0.044</td>
</tr>
<tr>
<td>OVCAR-8H</td>
<td>1.295</td>
<td>1.018</td>
</tr>
<tr>
<td>Yeast</td>
<td>1.183</td>
<td>0.862</td>
</tr>
<tr>
<td>DD</td>
<td>0.454</td>
<td>0.287</td>
</tr>
<tr>
<td>SW-620H</td>
<td>1.291</td>
<td>1.018</td>
</tr>
</tbody>
</table>

**Avg: 1.50X**

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

<table>
<thead>
<tr>
<th></th>
<th>dgl.op (ms)</th>
<th>TC-GNN (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROTEINS_full</td>
<td>0.062</td>
<td>0.019</td>
</tr>
<tr>
<td>OVCAR-8H</td>
<td>0.466</td>
<td>0.054</td>
</tr>
<tr>
<td>Yeast</td>
<td>0.401</td>
<td>0.051</td>
</tr>
<tr>
<td>DD</td>
<td>0.170</td>
<td>0.026</td>
</tr>
<tr>
<td>SW-620H</td>
<td>0.476</td>
<td>0.055</td>
</tr>
</tbody>
</table>

**Avg: 6.98X**
Kernel Performance (cuSPARSE)

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

<table>
<thead>
<tr>
<th></th>
<th>D (16)</th>
<th>D (32)</th>
<th>D (64)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cuSPARSE</td>
<td>TC-GNN</td>
<td>cuSPARSE</td>
</tr>
<tr>
<td>PROTEINS_full</td>
<td>90.13</td>
<td>130.89</td>
<td>170.55</td>
</tr>
<tr>
<td>OVCAR-8H</td>
<td>135.26</td>
<td>143.54</td>
<td>237.81</td>
</tr>
<tr>
<td>Yeast</td>
<td>135.42</td>
<td>157.97</td>
<td>238.12</td>
</tr>
<tr>
<td>DD</td>
<td>156.17</td>
<td>207.04</td>
<td>309.67</td>
</tr>
<tr>
<td>SW-620H</td>
<td>135.22</td>
<td>143.56</td>
<td>237.72</td>
</tr>
</tbody>
</table>

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.

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

yuke_wang@cs.ucsb.edu

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