QGTC: Accelerating Quantized Graph Neural Networks via GPU Tensor Core

Yuke Wang, Boyuan Feng, Yufei Ding
UC at Santa Barbara
Background

• Graph Neural Network Basics.

\[ a^{(k+1)}_v = \text{Aggregate}^{(k+1)}(h^{(k)}_u | u \in N(v) \cup h^{(k)}_v) \]

\[ h^{(k+1)}_v = \text{Update}^{(k+1)}(a^{(k+1)}_v) \]

• The adjacent matrix of GNNs is naturally well-suited for quantization.

• The quantization of weight and node embedding can also be beneficial.

Using 0/1 to indicate the existence of edge connections.

The precision loss in quantization can largely be offset through the iterative neighbor aggregation.
Background (cont’d)

• GPU Tensor Cores (TCs).
  • TC supports the compute primitive of \( D = A \times B + C \).
  • Matrix tile A and B are certain precision (e.g., 1-bit), while matrix tile C and D use uint32.

• 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.

```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

• The current TC can only support limited choices of bitwidth (e.g., 1-bit and 4-bit)

• TC initially tailored for dense GEMM computation may not directly fit the context of sparse GNN computation.

• The low-bit computation would cause the compatibility issue.

Unable to meet the demands of users for any-bitwidth (e.g., 2-bit) computation.

A huge waste of computation and memory access on those non-existed edges.

Unable to integrate with the existing deep-learning frameworks that operate on well-defined data type (e.g., FP32).
Contributions

- **Input level:** METIS for generating dense small subgraphs from sparse large graphs.

- **Algorithm level:**
  1-bit computation for anybitwidth QGNN computation.

- **GPU kernel level:**
  (i) 1-bit binarized representation for subgraph adjacent matrices;
  (ii) 3D-stacked bit compression;
  (iii) Zero-tile skipping and non-zero tile reuse.

- **Framework level:**
  Integration with PyTorch framework.
Quantized Computation in GNNs

**Any-bitwidth Scalar-Scalar Multiplication:** Assuming we have a 3-bit scalar value \((a)\) and multiply it with a 2-bit scalar value \((b)\), we can first represent these two values as

\[
\begin{align*}
    a &= at_2 \cdot 2^2 + at_1 \cdot 2^1 + at_0 \cdot 2^0 \\
    b &= bt_1 \cdot 2^1 + bt_0 \cdot 2^0
\end{align*}
\]

where \(at_n\) and \(bt_n\) indicate the bit value (0/1) at the certain bit position after bit decomposition. By following the general rule of multiplication, we can get \(a \cdot b\) as

\[
a \cdot b = (at_2 \cdot 2^2 + at_1 \cdot 2^1 + at_0 \cdot 2^0)(bt_1 \cdot 2^1 + bt_0 \cdot 2^0) \quad (4)
\]

through simplification we can get that

\[
a \cdot b = at_2 bt_1 \cdot 2^3 + (at_1 bt_1 + at_2 bt_0) \cdot 2^2 \\
+ (at_0 bt_1 + at_1 bt_0) \cdot 2^1 + at_0 bt_0 \cdot 2^0 \quad (5)
\]

**Any-bitwidth Vector-Vector Multiplication:** We extend the any-bitwidth scalar-scalar computation towards any-bitwidth vector-vector computation between a 3-bit vector \(\overrightarrow{a}^y\) and 2-bit vector \(\overrightarrow{b}^y\), each of which has \(k\) elements. Therefore, the above scalar-scalar multiplication formula can be extended to \(k\)-dimension vector-vector multiplication

\[
\overrightarrow{a}^y \cdot \overrightarrow{b}^y = \sum_{y}^{k} a(y) \cdot b(y) = \sum_{y}^{k} at_2^{(y)} bt_1^{(y)} \cdot 2^3 \\
+ \sum_{y}^{k} (at_1^{(y)} bt_1^{(y)} + at_2^{(y)} bt_0^{(y)}) \cdot 2^2 \\
+ \sum_{y}^{k} (at_0^{(y)} bt_1^{(y)} + at_1^{(y)} bt_0^{(y)}) \cdot 2^1 + \sum_{y}^{k} at_0^{(y)} bt_0^{(y)} \cdot 2^0
\]

Such a 1-bit vector-vector multiplication can be effectively implemented as

\[
ans_{i,j} = popcnt(\overrightarrow{a}^i & \overrightarrow{b}^j)
\]

where \(popcnt()\) counts the total number of 1s of the result in its bit representation (e.g., \(popcnt\) will return 3 for a binary number 1011). A similar procedure can be applied to generate
Quantized Computation in GNNs (cont’d)

Algorithm 1: 1-layer Quantized GNN Computation.

```
input: Full-bit adjacent matrix A (N × N), node embedding matrix X (N × D), and weight matrix W (N × H),
output: Updated full-bit node embedding matrix X (N × H).

/* Bit decomposition of the input matrices. */
A_bin = bitDecompose(A, 1)[0];
X_list = bitDecompose(X, s);
W_list = bitDecompose(W, t);

X_new_list = list(); C_dict = dict(); X = zeros_as(X);

/* Neighbor aggregation by bit-GEMM (A × X). */
for xIdx in len(X_list) do
    X_new_list.append(BMM(A_bin, X_list[xIdx]));
end

/* Node update by bit-GEMM (X_new × W). */
for xIdx in len(X_new_list) do
    for wIdx in len(W_new_list) do
        /* Compute bit-matrix at target bit level. */
        bitIdx = xIdx + wIdx;
        tmp_C = BMM(X_new_list[xIdx], W_list[wIdx]);
        C_dict[bitIdx].append(tmp_C);
    end
end

/* Elementwise reduction of results. */
for bitIdx in len(C_dict) do
    for idx in len(C_dict[bitIdx]) do
        X[idx] += C_dict[bitIdx][idx] << bitIdx;
    end
end
```

- Input bit-Decomposition
- Neighbor Embedding Aggregation (X’=AX)
- Node Embedding Update (H=X’W)
3D-Stacked Bit Compression

Figure 4. 3D-Stacked Bit Compression. Note that every 32 bits are compressed and stored in little-endian.

Zero-tile Jumping

Figure 5. Zero-tile Jumping. Note that each small grey square box (on the left side) indicates an edge connection between two nodes within a graph. Each grey rectangular box (on the right side) indicates at least one of its 32 consecutive small square boxes is grey (the presence of an edge).
Non-zero Tile Reuse

Figure 6. Non-zero Tile Reuse. Note that the grey box indicates the zero-tile of the subgraph adjacent matrix, while the white box with a block solid dot inside represents the non-zero tiles of the subgraph adjacent matrix.

PyTorch Integration

Bit-Tensor Data Type:
- We use the 32-bit integer tensor in PyTorch as the “vehicle” for holding any-bitwidth quantized numbers.
- Tensor.to_bit(nbits) to encode a int32/fp32 regular tensor as a bit tensor.
- Tensor.to_val(nbits) to decode a bit Tensor as int32/fp32 regular tensor.

Bit-Tensor Computation:
- bitMM2Bit(C, A, B, bit_A, bit_B, Bit_C) for bit tensor input (A, B) to bit tensor output (C).
- bitMM2Int(C, A, B, bit_A, bit_B) for bit tensor input (A, B) to regular (int32/fp32) tensor output (C).
Evaluation

• **GNN models:**
  - Cluster GCN
  - Batched GIN

• **Baselines:**
  - Deep Graph Library (DGL)
  - cuBLAS
  - CUTLASS

• **Platform Configuration:**
  - 8-core 16-thread Intel Xeon Silver 4110 CPU 2.8GHz with 64GB host memory.
  - NVIDIA Ampere RTX3090 GPU with 24GB device memory.

| Type | Dataset       | #Vertex | #Edge     | Dim. | #Class |
|------|---------------|---------|-----------|------|--------|
| I    | Proteins     | 43,471  | 162,088   | 29   | 2      |
|      | artist       | 50,515  | 1,638,396 | 100  | 12     |
| II   | BlogCatalog  | 88,784  | 2,093,195 | 128  | 39     |
|      | PPI          | 56,944  | 818,716   | 50   | 121    |
| III  | ogbn-arxiv   | 169,343 | 1,166,243 | 128  | 40     |
|      | ogbn-products| 2,449,029 | 61,859,140 | 100  | 47     |
Figure 7. End-to-end performance comparison with (a) DGL on Cluster GCN and (b) DGL on Batched GIN. (c) Compared with TC-based cuBLASgemmEX (int8) on GNN aggregation kernel throughput performance (in TFLOPs). Note that “QGTC_3” stands for QGTC with 3-bit data representation for node embedding matrix.
## Additional Studies

### Table 3. Compared with CUTLASS-int4 (TFLOPs).

| N   | Dim | CUTLASS (int4) | QGTC (1-bit) | QGTC (2-bit) | QGTC (3-bit) | QGTC (4-bit) |
|-----|-----|----------------|--------------|--------------|--------------|--------------|
| 2048| 32  | 10.36          | 32.65        | 19.99        | 14.40        | 11.30        |
| 4096| 32  | 12.28          | 81.41        | 46.23        | 32.27        | 24.75        |
| 8192| 32  | 12.67          | 94.58        | 50.82        | 35.22        | 26.31        |
| 2048| 64  | 21.40          | 63.94        | 39.41        | 29.83        | 22.15        |
| 4096| 64  | 24.66          | 89.18        | 51.21        | 35.17        | 25.38        |
| 8192| 64  | 24.70          | 104.66       | 55.16        | 40.77        | 31.07        |

### Figure 8. Zero-tile jumping efficiency. The percentage (%) on each green bar indicates the ratio of the number of tiles processed w/ jumping versus w/o jumping solution.
Additional Studies (cont’d)

Figure 9. Adjacency matrix size impact. Note that we choose the common subgraph size $N=\{128, 256, \ldots, 32768\}$ and the hidden embedding dimension $D=\{16, 32, \ldots, 1024\}$.

Figure 10. Non-zero tile reuse effectiveness. Note that we choose subgraph size $N=\{1024, 2048, 4096, 8192\}$ for this study.
Thank You

QGTC is open-sourced at [https://github.com/YukeWang96/PPoPP22_QGTC.git](https://github.com/YukeWang96/PPoPP22_QGTC.git)