QGTC: Accelerating Quantized GNN via GPU Tensor Core

Anonymous Author(s)

ABSTRACT

Over the recent years, quantized graph neural network (QGNN) attracts lots of research and industry attention due to their high robustness and low computation and memory overhead. Unfortunately, the performance gains of QGNN have never been realized on modern GPU platforms. To this end, we propose the first Tensor Core (TC) based computing framework, **QGTC**, to support any-bitwidth computation for QGNNs on GPUs. We introduce a novel quantized low-bit arithmetic design based on the low-bit data representation and bit-decomposed computation. We craft a novel TC-tailored CUDA kernel design by incorporating 3D-stacked bit compression, zero-tile jumping, and non-zero tile reuse technique to improve the performance systematically. We incorporate an effective bandwidth-optimized subgraph packing strategy to maximize the transferring efficiency between CPU host and GPU device. We integrate QGTC with the Pytorch framework for better programmability and extensibility. Extensive experiments demonstrate an average 3.17× speedup compared with the state-of-the-art Deep Graph Library framework across diverse settings.

1 INTRODUCTION

With the popularity surge of the graph neural networks (GNNs) [12, 18, 31], research around the full-precision GNNs has been widely studied in terms of its algorithms [18, 33] and execution performance [10, 20, 32]. On the other side, quantized GNN [9, 29] (QGNN) recently attract lots of attention thanks to its negligible accuracy loss, resilience towards malicious attacks, and significantly lower computations and memory overhead. We next summarize several key features of GNN that make it intrinsically suitable for quantization. First, the adjacent matrix of GNNs is naturally well-suited for quantization, since we only need to use 0/1 to indicate the existence of edge connections. Thus, using low bits for such information can save both memory and computation. Second, the quantization on weight and node embedding can also be beneficial. Because the tiny precision loss in quantization can largely be offset by the node information fusion through the iterative neighbor aggregation process of GNNs. Besides, the quantization of the original continuous floating-point values can absorb the input perturbation from adversarial attacks.

Despite the great theoretical success of QGNN, the realization of such benefits on modern high-performance computing platforms (*e.g.*, GPUs) is still facing tremendous challenges. First, existing GPU-based GNN computing frameworks [10, 32] are designed and tailored for GPU CUDA cores, which is intrinsically bound by its peak throughput performance and can only handle the byte-based data types (*e.g.*, `int32`). Although quantized computation can be achieved via pure algorithmic emulation, the actual bit-level performance gains could hardly be harvested, since all underlying arithmetic operations still have to rely on well-defined data types from CUDA/C++ libraries.

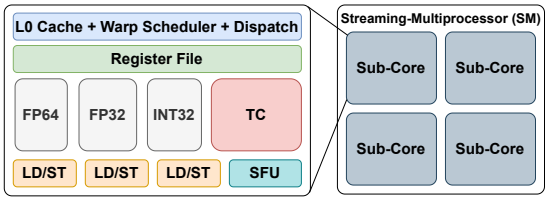

Figure 1: GPU SM with TC Design. Note that FP64, FP32, INT, LD/ST, and SFU are for double-precision, single-precision, integer, load/store, and special function unit, respectively.

To tackle these challenges, we decide to move forward with the recent GPU hardware feature – **Tensor Core** (TC). The modern NVIDIA GPU with TC design is illustrated in Figure 1. TC provides the native support of bit-level operations (`XOR`, `AND`), which could be the major ingredient for quantized computation. Besides, TC can easily beat CUDA core with a significantly higher throughput performance (more than 10×) on conventional NN operations (*e.g.*, linear transformation and convolution). This demonstrates the potential of using TC in accelerating QGNNs. However, directly using TC for QGNN computation is encountering several challenges. First, the current TC can only support limited choices of bitwidth (*e.g.*, 1-bit and 4-bit), which may not be able to meet the demands of users for any-bitwidth (*e.g.*, 2-bit) computation. Besides, TC initially tailored for GEMM computation may not directly fit the context of sparse GNN computation. A huge amount of computation and memory access efforts would be wasted on those non-existed edges. Because the hard constraint of TC input matrix tile-size (*e.g.*, 8×128 for 1-bit GEMM) has to be satisfied, which may require excessive zero paddings. In addition, the low-bit computation would cause the compatibility issue, since none of the existing deep-learning frameworks [1, 27] could directly operate on the low-bit data type, which has never been defined by any libraries. Therefore, we remark there are several aspects to be considered in order to use TC for QGNNs: 1) ***Hardware-level Support***. This inspires us to explore the high-performance GPU hardware features that can efficiently support the QGNN computation. Even though it is hard to find such a GPU hardware feature that can directly support any-bitwidth QGNN, some indirect hardware features would potentially be helpful. For example, NVIDIA introduced the 1-bit TC-based GEMM on Turing Architecture, which essentially can be used to composite any-bitwidth GEMM. 2) ***Software-level Optimizations***. This motivates us to optimize the kernel computation according to the characters of QGNN. GNN computation is featured with a highly sparse and irregular scheme. It is intrinsically not favorable for the dense GPU computation flow tailored for the traditional NN computation. Thus, how to handle such input-level irregularity from the computation and memory perspectives is essential to the

SC '21, November 14–19, 2021, St. Louis, MO

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

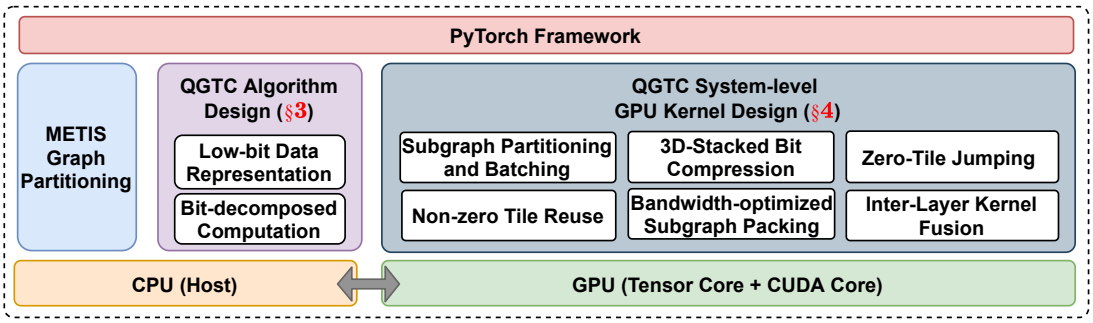

Figure 2: QGTC Overview.

performance of QGNN. For example, subgraph partitioning [16] based mini-batch GNN computation has been used to increase the computation efficiency without compromising model accuracy performance. 3) ***Framework-level Integration***. This encourages us to bridge the gap between quantized low-bit implementations and deep-learning frameworks built for full-precision computation. Therefore, our whole system-level design can be seamlessly integrated with the state-of-the-art mainstream NN frameworks to benefit the execution performance and the developing productivity.

To this end, we propose the first TC-based computation framework (Figure 2) to support any-bitwidth QGNN on GPUs.

At the input side, we incorporate the METIS [16] graph partitioning to generate a set of dense subgraphs from the highly irregular and sparse input graphs. The insight here is that nodes in real-world graphs are likely to form clusters, and such information can be used to benefit the efficiency of GNN computing and model algorithmic performance.

At the algorithm side, we leverage the insight that any-bitwidth QGNN computation can always be decomposed into the 1-bit computation. Each bit in the output can be generated by different combination of bits from the input. Thus, we propose to use quantized low-bit data representation and bit-decomposed computation base on the "atomic" 1-bit data type.

At the GPU kernel side, we craft a low-bit computation design tailored for QGNN computation on batched dense subgraphs. We addressing the key performance bottleneck of the low-bit GNN computing from the memory and computing perspective. Specifically, we use only 1-bit binarized representation for the subgraph adjacent matrix, which is memory efficient for representing the presence/absence of edge connections between nodes. Besides, we use a 3D-stacked bit-compression technique for maintaining quantized low-bit node embedding features and weights. In addition, we fully exploit the intra-subgraph sparsity through zero-tile skipping and non-zero tile reuse, which can further avoid unnecessary computations and improve the data locality.

At the framework side, we seamlessly integrate QGTC with the state-of-the-art Tensor based Pytorch [27] frameworks. We introduce the notion of bit-Tensor data type and bit-Tensor Computation and warp them up as a new set of Pytorch API extensions. End-users can directly interact with the QGTC Pytorch APIs to access all functionalities. This can largely improve the programmability and extensibility of QGTC.

To conclude, our key contribution can be summarized as follows

- We propose a novel 1-bit composition technique for any-bitwidth arithmetic design (§3), which can support the computation of QGNN with board range of precision demands.
- We introduce a highly-efficient implementation of QGNN kernel (§4) built on top of GPU Tensor Core with a series of computation (*e.g.*, zero-tile jumping) and memory (*e.g.*, non-zero tile reuse) optimizations. We further design a 3D-stacked bit-compression technique for maintaining subgraph adjacency matrices, node feature embeddings, and weight matrices with any-bitwidth precision.
- We incorporate an effective bandwidth-optimized subgraph packing strategy to maximize the transferring efficiency.
- We integrate QGTC with the Pytorch framework (§5) by introducing the notion of bit-Tensor data type and bit-Tensor computation for better programmability and extensibility.
- Intensive experiments demonstrate the significance of QGTC in terms of high performance (average 3.17× speedup) compared with the state-of-the-art Deep Graph Library framework on mainstream GNN models across various datasets.

2 BACKGROUND AND RELATED WORK

2.1 Graph Neural Networks

Graph neural network (GNN) is an effective tool for graph-based machine learning. The detailed computing flow of GNNs is illustrated in Figure 3. 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,

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

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); $\mathrm{N}(v)$ is the neighbor set of node v. The aggregation method and the order of aggregation and update could vary across different GNNs. Some methods [12, 18] just rely on the neighboring nodes while others [31] 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.

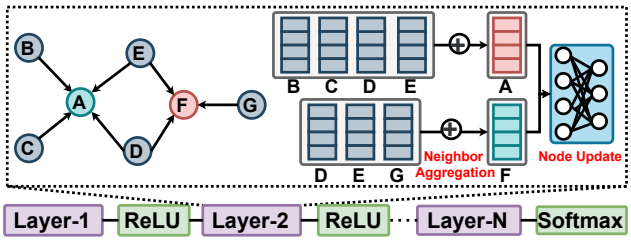

Figure 3: GNN General Computation Flow.

The most recent advancement of GNN is its batched computation [5], which has also been adopted by many state-of-the-art GNN computing frameworks [10, 32] for large graphs that cannot be easily fit into the GPU/CPU memory for computation directly. Batched GNN computation has been highlighted with good accuracy and runtime performance [5] in comparison with full-graph computation. Batched GNN computation takes several steps. First, it decomposed the input graphs by employing the state-of-the-art graph partitioning toolset, such as METIS [16], which can minimize the graph structural information loss meanwhile maximizing the number of edge connections within each subgraph (*i.e.*, improving the subgraph modularity). Second, it feeds the small subgraphs into the GNN models for computation, which will generate the node feature vector for each subgraph. Third, the generated node embeddings can be used in multiple downstream tasks, such as node/graph classification and link prediction.

2.2 Quantization on GNNs

Besides the research efforts on full-precision GNNs, recent focus also shifts towards the quantized GNNs. For example, Boyuan et al. [9] propose the first framework for running quantized GNNs and several types of quantization schemes can be applied on GNNs, such as the quantization based on the GNN layer, quantization based on node degrees, and quantization based on the edge weights. And their experimental results also demonstrate the effectiveness of the GNN quantization in terms of memory saving and model accuracy. Shyam et al. [29] introduce an architecturally-agnostic and stable method, Degree-Quant, to improve performance over existing quantization-aware training baselines commonly used on other architectures, such as CNNs. They achieve up to 4.7× speedups on CPU when using `int8` in comparison with the full-precision `float`.

Compared with the full-precision GNNs, low-bit GNNs bring the benefit of model robustness towards the adversarial attacks and the low computation and memory overheads. However, work from [9] only showcases the theoretical memory and computation benefits via software-level quantization simulation, where its underlying computation is still carried out in 32-bit full-precision `float`. Work from [29] only demonstrates such gains on CPUs, which has limited applicability in the real-world GNN computation settings. This encourages us to harvest its real performance benefits on the modern widely used GPU platforms.

2.3 Tensor Core on GPUs

The recent advancement of GPU hardware technology has pushed the computing power to a new level. Among those innovations, the most significant one is the Tensor Core (TC) on NVIDIA GPU. Different from scalar-scalar computation on CUDA Cores, TC provide a matrix-matrix compute primitive, which can deliver more than 10× higher computation throughput. The initial version of TC is designed for handling the GEMM with half precision input and full-precision output. More variants (*e.g.*, `int8`, `int4`, and `int1` inputs with 32-bit unsigned integer (`uint32`) output) have been introduced since the recent CUDA release (11.0) and new GPU micro architectures (*e.g.*, Turing and Ampere).

In particular, TC supports the compute primitive of $\mathbf{D} = \mathbf{A}\times\mathbf{B}+\mathbf{C}$, where matrix tile $\mathbf{A}$ and $\mathbf{B}$ are required to be a certain type of precision (*e.g.*, 1-bit), while matrix tile $\mathbf{C}$ and $\mathbf{D}$ use `uint32`. Depending on the input data precision and the version of GPU micro architecture, the matrix tile size of $\mathbf{A}(M\times K)$, $\mathbf{B}(K\times N)$, and $\mathbf{C}(M\times N)$ may have different choices. For example, 1-bit Tensor Core computing requires $M = N = 8$ and $K = 128$. Different from the CUDA Cores which requires users to define the execution flow of each thread (*i.e.*, work of individual threads). TC requires the collaboration of a warp of threads (32 threads) (*i.e.*, work of individual warps). This can be reflected in two ways. First, before calling TC for computation, all registers of a warp of threads need to collaboratively store the matrix tile into a new memory hierarchy (called *Fragment* [25]), which allows data sharing across registers. This intra-warp sharing provides opportunities for fragment-based memory optimizations. Second, during the computation, these loaded matrix fragments will be taken as the TC input to generate the output fragment, which also consists of the registers from each thread in a warp. Data movements among these registers are also managed by a warp of threads collaboratively.

Since the appearance of the TC, research efforts have been devoted to accelerating high-performance computing workloads with TC. Ahmad et al. [2] process the batched small-size GEMM on TC for acceleration. Ang Li and Simon Su [19] leverage 1-bit GEMM capability on Turing GPU TC for accelerating binary neural network inference. Dakkak [7] accelerates the half-precision scan on TC by transforming scan to a GEMM workload. Boyuan et al. [8] introduce GEMM-based scientific computing on TC with extended-precision and high performance. QGTC enlarges the application range of TC by accelerating GNNs for any-bitwidth quantized GNN computation, which is not directly covered by any existing research, any release of CUDA [26]/cuBLAS [23]/CUTLASS [22] library, and the low-level TC hardware design.

Listing 1: Basic CUDA WMMA APIs for TC.

```
// 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 (32-bit).
wmma::store_matrix_sync(C, c_frag, N, mem_row_major);
```

TC can be used in several ways. The most simple way is to call cuBLAS [23] by using the `cublasSgemmEX` API. However, cuBLAS API only supports computation on the most common fixed bit-width

on TC, such as 8-bit, half-precision (16-bit), thus, it cannot support any bitwidth precision directly. The second way is to call the Warp Matrix Multiply-Accumulate (WMMA) (`nvcuda::wmma`) API [24] in CUDA C++ to operate TC directly. There are basically four types of operations (Listing 1). In this project, we follow the second way for more low-level implementation customization for batched GNN computation. Because it can offer more design/implementation flexibility for compositing arbitrary-bit computation and ease the optimization (*e.g.*, data loading and reuse) for batched GNN-specific workloads at the GPU kernel-level design.

3 QGTC ALGORITHM DESIGN

In this section, we will first introduce the low-bit computation basics, then discuss our algorithm design for quantized GNN.

3.1 1-bit Composition for Quantized Ops.

Over the last few years, quantized deep neural networks (QDNNs) [9, 29] have been extensively studied, largely due to their memory saving and high computation performance. In GNN, however, similar work is largely lagging behind. Work from [9] demonstrates that GNN is actually insensitive to quantization, even very low-bit quantization would not lead to evident accuracy loss because of the graph-like aggregation operations that can amortize such quantization influence. Another work from [3] also demonstrates that even the binarized GNN would be beneficial in some application scenarios. In this work, we foresee that the support for any-bitwidth precision computation on GNN is vital to satisfy various users demands (*e.g.*, execution time).

Given a quantization bit q and the 32-bit floating-point value $\alpha \in \mathcal{R}$, we quantize it as a q-bit value by using the Equation 2

$$\alpha^{(q)} = \left\lfloor \frac{\alpha - \alpha_{min}}{scale} \right\rfloor. \tag{2}$$

where α_{min} is an empirical lower bound that can be determined by users or application settings; *scale* is the ratio between the range ($|\alpha_{max} - \alpha_{min}|$, where α_{max} is an empirical upper bound) and the q-bit representation range (2^q); $\lfloor \cdot \rfloor$ is the floor function.

For any-bitwidth computation on quantized values, we propose a new type of arithmetics based on the "atomic" 1-bit computation widely used in the binarized neural network (BNN) domain [14].

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

$$\begin{aligned} a &= at_2 \cdot 2^2 + at_1 \cdot 2^1 + at_0 \cdot 2^0 \\ b &= bt_1 \cdot 2^1 + bt_0 \cdot 2^0 \end{aligned} \tag{3}$$

where at_* and bt_* 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

$$a \cdot b = (at_2 \cdot 2^2 + at_1 \cdot 2^1 + at_0 \cdot 2^0)(bt_1 \cdot 2^1 + bt_0 \cdot 2^0) \tag{4}$$

through simplification we can get that

$$\begin{aligned} a \cdot b = at_2bt_1 \cdot 2^3 + (at_1bt_1 + at_2bt_0) \cdot 2^2 \\ +(at_0bt_1 + at_1bt_0) \cdot 2^1 + at_0bt_0 \cdot 2^0 \end{aligned} \tag{5}$$

Any-bitwidth Vector-Vector Multiplication: We extend the any-bitwidth scalar-scalar computation towards vector-vector any-bitwidth computation between a 3-bit vector $\overrightarrow{v_i}$ and 2-bit vector $\overrightarrow{v_j}$, each of which has k elements. Therefore, the above scalar-scalar multiplication formula can be extended to k-dimension vector-vector multiplication as follows.

$$\begin{aligned} \overrightarrow{v_i} \cdot \overrightarrow{v_j} = \sum_y^k a^{(y)} \cdot b^{(y)} = \sum_y^k at_2^{(y)} bt_1^{(y)} \cdot 2^3 \\ + \sum_y^k (at_1^{(y)} bt_1^{(y)} + at_2^{(y)} bt_0^{(y)}) \cdot 2^2 \\ + \sum_y^k (at_0^{(y)} bt_1^{(y)} + at_1^{(y)} bt_0^{(y)}) \cdot 2^1 + \sum_y^k at_0^{(y)} bt_0^{(y)} \cdot 2^0 \end{aligned} \tag{6}$$

From the above formula, we can see that in order to compute the result of any-bitwidth vector-vector multiplication, we first do bit decomposition on all elements in each vector then do bit-bit multiplication between elements from each vector, and finally do bit shifting and reduction to get the final result. For example, after bit-decomposition of $\overrightarrow{v_i}$ and $\overrightarrow{v_j}$, we can get $\overrightarrow{v_i}$ at bit position 2 as $at_2^{(y)}$ and $\overrightarrow{v_j}$ at bit position 1 as $bt_1^{(y)}$, where $y \in [0, k)$. From the multiplication and addition, we can get the multiplication result of $\overrightarrow{v_i} \cdot \overrightarrow{v_j}$ at bit position 3. Such a 1-bit vector-vector multiplication can be effectively implemented as

$$ans_{i,j} = popcnt(\overrightarrow{v_i} \& \overrightarrow{v_j}) \tag{7}$$

where $popcnt()$ counts the total number of 1*s* 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 the result at bit position 0, 1, and 2. After all these individual bit in temporary results are ready, we can do bit shifting and reduction to get the final result. Based on such any-bitwidth vector-vector results, we can easily derive the any-bit matrix-matrix multiplication scheme, where each element in the output matrix can be seen as the results of any-bitwidth vector-vector multiplication.

3.2 Quantized Computation in GNNs

Applying any-bitwidth precision computation in GNNs would find two major specialties. First, the adjacent matrix (**A**) of GNNs only need to use binary (1-bit) number to represent the presence/absence of edges. Second, the node embedding matrix (**X**) and the weight matrix (**W**) can be represented with any-bitwidth to meet the precision demands of different users.

As described in Algorithm 1, each layer of any-bitwidth GNN consists of a *node aggregation* and a *node update*) phase. Specifically, neighbor aggregation conducts $\mathbf{X\_new} = \mathbf{A} \cdot \mathbf{X}$ through a *1-bit-and-s-bit* matrix multiplication and the node update conducts $\hat{\mathbf{X}} = \mathbf{X\_new} \cdot \mathbf{W}$ through a *s-bit-and-t-bit* matrix multiplication. To avoid any data overflow during the reduction (Line 15 to 19), $\hat{\mathbf{X}}$ should also use full-bit data type (*e.g.*, `int32`). For large graphs, their adjacent matrices cannot be easily fit into the GPU device memory directly. In this scenario, we employ METIS [16] for graph partitioning and run GNN as batched subgraph computation, which is used by the most popular cluster-GCN [5] design. Considering that the number of subgraphs generated by METIS [16] is usually within the reasonable size (2000 to 20000), such a batched GNN computation can be accommodated on a single modern GPU without violating the its memory constraints. Note that to reduce the runtime overhead, the bit-decomposition of matrix **W** and **A** can be pre-computed and cached before the GNN computation at each

Algorithm 1: 1-layer Quantized GNN Computation.

input : Full-bit adjacent matrix $\mathbf{A}$ ($N \times N$), node embedding matrix $\mathbf{X}$ ($N \times D$), weight matrix $\mathbf{W}$ ($N \times H$).
output: Updated full-bit node embedding matrix $\hat{\mathbf{X}}$ ($N \times H$).

```
/* Bit decomposition of the input matrices.                    */
A_bin = bitDecompse(A, 1)[0];
X_list = bitDecompse(X, s);
W_list = bitDecompse(W, t);
X_new_list = list(); C_dict = dict(); X̂ = zeros_as(X);
/* Neighbor aggregation by bit-GEMM between A and X.           */
for xIdx in len(X_list) do
    X_new_list.append(BMM(A_bin, X_list[xIdx]));
end
/* Node update by bit-GEMM between X_new and W.                */
for xIdx in len(X_new_list) do
    for wIdx in len(W_new_list) do
        /* Compute the 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 from the last step.        */
for bitIdx in len(C_dict) do
    for Idx in len(C_dict[bitIdx]) do
        X̂[Idx] += C_dict[bitIdx][Idx] ≪ bitIdx;
    end
end
```

layer. Because across different GNN layers of the same subgraphs, the matrix $\mathbf{A}$ can be reused. On the other side, across different subgraphs at the same GNN layers, the matrix $\mathbf{W}$ can be reused.

4 IMPLEMENTATION

In this section, we will discuss the implementation details and the optimizations used in QGTC.

4.1 Subgraph Partitioning and Batching

Real-world graphs usually come with large number of nodes and highly-irregular graph structure (edge connections). This bring two levels of difficulties for GNN computing. The first one is the memory consumption, since GPU device memory cannot accommodate all nodes, edges, and node embedding features at the same time. The second one is the inefficient execution, since the irregular and sparse edge connections leads to low memory access efficiency and poor computation performance. To this end, in QGTC, we combine the state-of-the-art graph partitioning technique METIS [16] and subgraph batch processing [5] to handle different sizes of input graphs effectively. Compared with other solutions, such as graph clustering approaches [4, 15, 28] and BFS-based methods [6], METIS achieves better quality of its captured subgraph partitions (more edges in each subgraph) and the significantly higher runtime performance owing to its partial parallelization. Note that the number of subgraphs/partitions is determined by users and is passed as a runtime parameter to METIS.

After the subgraph partitioning, we will conduct a batching step for GNN computation on GPUs. This step basically gathers a set of subgraph partitions based on a user-defined batch size. Later, during the GNN computing, subgraphs are loaded to GPU memory by batch. Using the partitioning and batching strategy for GNN computing gives users control of workloads at two-levels of granularity. First, the workload granularity is defined by the number of subgraphs/partitions. This would manage the size of each subgraph partition and the edge connection density of each subgraph. In general, the more number of the subgraphs/partitions would lead to denser edges connections within each subgraph, which may bring better computation and memory locality. Second, the processing granularity is controlled by the batching size. This would determine the size of graphs that will be fit into the GPU at each round of execution. The selection of batch size would maximize the utilization of the GPU for better performance while respecting the GPU computation and memory resource constraints.

4.2 3D-Stacked Bit Compression

As we discussed in the previous section, existing NN frameworks are developed for full-precision computation. This leads to challenges from two aspects. First, the low-bit quantized data type cannot directly borrow the full-precision data type as the "vehicle" for computation. The major reason is that the full-precision data type such as `float` and `int32` cannot bring any benefits of the memory or computation saving. Second, low-bit quantization would not fit any type of bit alignment, since its bit-level boundary mostly cannot be divisible by the size of a byte (8-bit), making it hard to retrieve its actual value.

To this end, we propose a novel 3D-stacked bit-compression technique to handle any-bitwidth data type effectively. The major idea is to compress any-bitwidth input with 32-bit alignment for ease of value retrieval and memory alignment. As exemplified in Figure 4(a), we have a input matrix with the shape of 3-bit$\times M \times K$. For each bit of the element in the matrix, we store it in a bit matrix (1-bit$\times M \times K$) stacked along the vertical z axis. During the computation of any-bitwidth matrix multiplication $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, two types of 3D-stacked bit-compression are employed. For matrix $\mathbf{A}$, we use the *column-wise compression* with 32-bit alignment, as illustrated in Figure 4(b). The main reason of choosing column-wise compression is that the matrix multiplication would benefit from coalesced across-column memory access along each row of matrix $\mathbf{A}$. 32-bit alignment can benefit the read performance by coalesced loading from the global memory to fragment. After the compression on matrix $\mathbf{A}$ (1-bit$\times M \times K$), we will get a 32-bit compressed 3-bit $\mathbf{A_c}$ with the shape of 3-bit$\times(\mathbf{PAD8}(M) \times \lfloor \mathbf{PAD128}(K)/32 \rfloor)$, where **PAD8** and **PAD128** are for padding rows/columns that cannot be divisible by the basic TC computing size ($M(8) \times N(8) \times K(128)$).

For matrix $\mathbf{B}$, we use the *row-wise compression* with 32-bit alignment, as shown in Figure 4(c) which can benefit the across-row access along each column of matrix $\mathbf{B}$. After the compression on matrix $\mathbf{B}$ (1-bit$\times M \times$K), we will get a 32-bit compressed 2-bit $\mathbf{B_c}$ with the shape of 2-bit$\times \lfloor \mathbf{PAD128}(K)/32 \rfloor \times \mathbf{PAD8}(N)$ for the output layer. Note that if the $\mathbf{A} \times \mathbf{B}$ is the hidden layer of a GNN model,

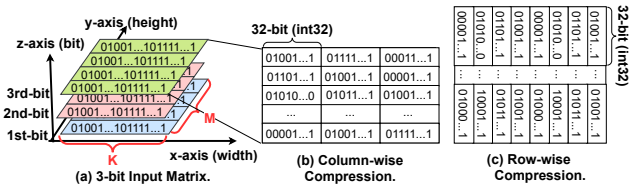

Figure 4: 3D-stacked Bit Compression. Note that every 32-bit is compressed and stored in little-endian.

the padding strategy on matrix **B** would be slightly different considering that the result matrix **C** will become a new matrix **A** in the next layer which demands 128-bit padding. In this case, avoid additional padding overhead, we will get the 2-bit $\mathbf{B_c}$ with the shape of 2-bit×$\lfloor \mathbf{PAD128}(K)/32 \rfloor \times \mathbf{PAD128}(N)$ in the current layer.

4.3 Zero-tile Jumping

Even though the subgraph partitioning such as METIS [16] make the subgraph denser (more number of edge connections within each subgraph), there are still some TC tiles (*i.e.*, the input matrix tile for a single TC computation) are filled with all-zero elements. Therefore, directly iterating through these zero tiles would introduce the cost of unnecessary memory (loading data from the global memory to thread-local registers) and computation (processing 1-bit TC-GEMM on input adjacent matrix tile that contains all-zero elements). Based on this observation, we introduce a novel zero-tile jumping technique to reduce the unnecessary computations by leveraging the bitwise `OR` operation and warp-level synchronization primitives.

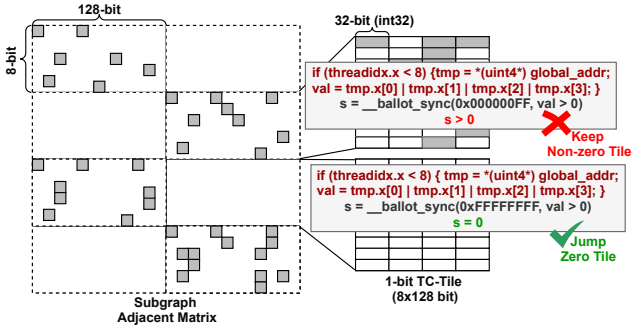

Figure 5: Zero-tile Jumping. Note that each small grey square box (on the left side) indicates an edge connection between two nodes within a graph. Each grey rectangular box (on the right side) indicates at least one of its 32 consecutive small square boxes is grey (the presence of an edge).

As illustrated in Figure 5, each 1-bit TC-GEMM would work on the tile size of 8×128 register fragment. This can be well partitioned into 8×4 `int32` elements. To check whether the 8×128 tile contains all-zero elements, we first employ only 8 threads from a warp of threads to fetch an `uint4_v` vector data (each `uint4_v` element in CUDA consists of 4 `int32` elements placed in continuous memory address). The reason of using `uint4_v` is to maximize the memory access efficiency by issuing fewer global memory requests. Once all `uint4_v` elements have been loaded. Each thread will apply bitwise `OR` across all 4 `int32` elements, which will check whether each row of a TC-tile is all-zero. The next step is to tell whether across-different rows, the whole tile is all-zero, we will use the warp-level primitive to sync the information across these 8 active threads in the warp. This step will generate an `int32` number. If this number is zero, it will indicate all elements in this input TC-tile are zero. Otherwise, we still need to conduct the 1-bit TC-GEMM on the currently tile. We will give more quantitatively analysis of such zero-tile jumping at the Section 6.4.

4.4 Non-Zero Tile Reuse

In addition to jumping over the zero tiles, we further consider reusing the non-zero tiles to improve data locality. In the aggregation step of the GNN computation, we generate the output feature map at different bit level separately. For example, when we choose 1-bit adjacent subgraph matrix and a 4-bit feature embedding matrix, we will execute the iteration 4 times to generate the output. One straightforward solution, called *cross-bit reduction*, is to generate the complete output matrix tile at each bit level first. This requires loading the matrix tile imperatively, as shown in Figure 6(a). However, this would cause one problem that each non-zero tile from the adjacent matrix will be repetitively loaded when computing with each bit matrix from the embedding matrix.

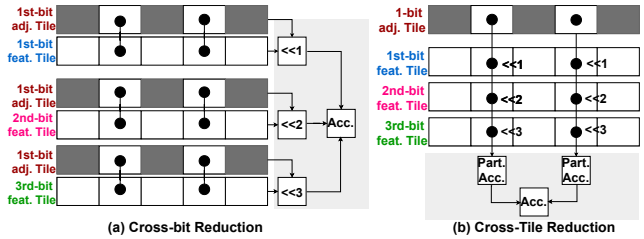

Figure 6: Non-zero Tile Reuse. Note that the grey box indicates the zero-tile of the subgraph adjacent matrix, while the white box with a block solid dot inside represents the non-zero tiles of the subgraph adjacent matrix.

In fact, we can consider reordering the steps in a way that we can maximize the benefit of each non-zero tile of subgraph adjacency matrix. As shown in Figure 6(b), we introduce a *cross-tile reduction* strategy. Specifically, for each loaded non-zero fragment, we will use it to generate an output tile at all bit levels and do a localized reduction (only on the current tile) to generate a partial aggregation result. Once this part has been done, we will move forward to the next non-zero tile and repeat the same process until all non-zero tiles have been processed. Depending on the number of bits we choose the node feature embedding matrix, the complexity of loading the nonzero tiles can be reduced from $O(n)$ to $O(1)$, where n is the number of bits for node feature embedding matrix.

4.5 Bandwidth-Optimized Subgraph Packing

During the GNN computation of the subgraphs, data communication between the CPU host and GPU device is also non-avoidable. It will swap the subgraph data (such as edge lists and node embedding) in/out of the GPU device. One basic approach is to transfer the dense adjacent matrix considering that a single subgraph is

generally within the range of the modern GPU memory. However, this could easily lead to a huge amount of data traffic between the CPU and GPU host. The transferring performance is heavily bounded by PCIe bandwidth (32 GB/s for PCIe 4.0x16) between the CPU host and the GPU device. For node embedding matrix, the current practice is to transfer the node embedding matrix by initializing another standalone PCIe transferring, which incurs additional overheads and is unable to maximize the bandwidth usage.

To overcome these issues, we employ a new strategy, called bandwidth-optimized subgraph packing. Instead of directly migrating the large dense adjacent matrix transferring we just transfer the sparse edge list then covert it to dense matrix on the GPU device. This can trade the low-cost on-device computation for the high-cost PCIe data transferring. Besides, to facilitate the node embedding matrix transferring at the meantime, we compress the sparse edge list and the node feature embedding into a compound memory object (occupying consecutive memory space) on the host first and then initiate the transferring of this compound memory object from the host CPU to GPU device.

4.6 Inter-layer Kernel Fusion

Across the GNN layers, we incorporate the low-bit data transferring. Specifically, the output of the one hidden layer will directly be handed over to the next layer as the input. There are several strategies we use. First, we apply data quantization and bit-decomposition at the end of the computation kernel such as the neighbor aggregation and node update. This can help to avoid outputting the result to the slow global memory and apply the data manipulation again. Second, standalone activation function kernels such as `ReLU` and `tanh`, can be effectively fused into our computation kernel as a `device` function, which can directly operates the shared memory results to achieve high performance. For the batch normalization (BN) layers that follow the graph convolution layers, we can also do layer fusion based on the following equation.

$$\mathbf{BN}(x_{i,j}) = \left(\frac{x_{i,j} - \mathbb{E}[x_{*,j}]}{\sqrt{Var[x_{*,j}] + \epsilon}} \right) \cdot \gamma_j + \beta_j \tag{8}$$

where β_j, γ_j, and ϵ are the BN parameters that can be incorporated into the low-bit convolutional kernel to avoid launching a BN kernel. One thing worth noting is that computation at the hidden layer and the output layer is slightly different. For hidden layers, each kernel has the quantization + bit-decomposition on the output activation, since the next layer relies on the low-bit data as the input for computation. While for the last layer, once the full-precision accumulation is complete, it will directly output the full-precision result for the `softmax` layer (considering the node classification task) to generate logits that demand high precision.

5 INTEGRATION WITH PYTORCH

Besides the highly efficient kernel design and data transferring optimization, for better usability and programmability, we integrate QGTC with the popular Pytorch framework. However, there are two key technical challenges. The first one is how to represent the quantized low-bit number in those Tensor-based frameworks that are built on byte-based data types (*e.g.*, `int32`). The second one is how to apply valid computation between the quantized low-bit number and those well-defined byte-based number. For example, how could we get the correct results when we do arithmetic multiplication between a 2-bit number and a 32-bit integer number. To this end, we introduce two new techniques.

Bit-Tensor Data Type: We use the 32-bit `IntTensor` in Pytorch as the "vehicle" for holding any-bitwidth quantized data. And we leverage our 3D-stacked bit compression technique (Section 4.2) to package the quantized data. We offer a Pytorch API `Tensor.to_bit(nbits)` for such data type conversion functionality. Note that existing Pytorch APIs, such as `print`, are only defined for those complete data types, such as Int. Therefore, to access the element value of a bit-Tensor, we provide `Tensor.to_val(nbits)` to decode a bit-Tensor as `int32` Tensor (converting each element from a low-bit number to a `int32` number). This can make our design compatible with existing Pytorch functionalities.

Bit-Tensor Computation: We handle two different types of computation: 1) the operations that only involve bit-Tensor and 2) the operations that involve both bit-Tensor and `float`/`int32` Tensor. For the first type of operations, we built two APIs based on whether we want to get the `int32` output or still get the quantized low-bit output as a bit-Tensor. For any-bitwidth MM with low-bit output, the API is `bitMM2Int(C, A, B, bit_A, bit_B, bit_C)`, where `A` and `B` are bit-Tensors, `bit_A/B/C` are bitwidth parameters. For any-bitwidth MM with `int32` output, the API is `bitMM2Bit(C, A, B, bit_A, bit_B, bit_C)`. For the second type of operations, we will first decode a bit-Tensor as a `float`/`int32` Tensor by using `Tensor.to_val(nbits)`. Then we call the official APIs in Pytorch for the regular full-precision computation.

6 EVALUATION

In this section, we comprehensively evaluate QGTC in terms of its performance and adaptability on various GNN models and graphs.

6.1 Experiment Setup

Benchmarks: We choose two most representative GNN models widely used by previous work [10, 20, 32] on the node classification task to cover different types of aggregation. ***1) Cluster GCN*** [18] is one of the most popular GNN model architectures. It is also the key backbone network for many other GNNs, such as GraphSAGE [12], and differentiable pooling (Diffpool) [34]. For Cluster GCN evaluation, we use the setting: *3 layers with 16 hidden* ***2) Batched GIN*** [33] differs from cluster GCN in its order of aggregation and node update. Batched GIN aggregates neighbor embedding before the node feature update (via linear transformation). GIN demonstrates its strength by capturing the graph properties that cannot be collected by GCN according to [33]. Therefore, improving the performance of GIN will benefit more advanced GNNs, such as GAT [31] and AGNN [30]. For batched GIN evaluation, we use the setting: *3 layers with 64 hidden dimensions of each layer*. For quantization bitwidth, we cover the bitwidth settings from the existing quantized GNN studies [9, 29] and also conduct comprehensive experimental analysis on a wide range of bitwidth settings (from 2-bit to 32-bit). Our goal is to facilitate future quantization research, which will explore better quantized GNN designs that can well balance the model accuracy and runtime performance.

Baselines: we choose several baseline implementations for comparison. **1)** ***Deep Graph Library (DGL)*** [32] is the state-of-the-art

GNN framework on GPUs, which is built with the high-performance cuSPARSE [21] library as the backend and uses Pytorch [27] as its front-end programming interface. DGL significantly outperforms the other existing GNN frameworks [10] over various datasets on many mainstream GNN model architectures. Therefore, we make an in-depth comparison with DGL. **2) *Pytorch-Geometric (PyG)*** [10] is another popular GNN framework in which users can define edge convolutions when building their customized GNN aggregation layers. PyG leverages torch-scatter [11] library (a highly-engineered CUDA kernel) as the backend, which highlights its performance on batched small graphs.

Table 1: Datasets for Evaluation.

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

Datasets: We cover all three types of datasets, which have been used in many previous GNN-related work [10, 20, 32]. Details of these datasets are listed in Table 1. Specifically, ***Type I*** graphs are the popular GNN datasets evaluated by many GNN algorithmic papers [12, 18, 33]. ***Type II*** graphs [17] are the popular benchmark datasets for graph kernels and are selected as the built-in datasets for PyG [10]. ***Type III*** graphs [13] are challenging GNN benchmark datasets in terms of the large number of nodes and edges. These graphs demonstrate high irregularity in its structures. Note that we do graph partitioning by using METIS [16] and set the number of total subgraphs as 1500 as adopted by prior work [5, 35].

Platforms & Metrics: QGTC backend is implemented with C++ and CUDA C, while QGTC front-end is implemented in Python. Our major evaluation platform is a Ubuntu16.04 server with an 8-core 16-thread Intel Xeon Silver 4110 CPU @ 2.8GHz with 64GB host memory and an NVIDIA Ampere RTX3090 GPU with 24GB device memory. The GPU device kernel is compiled with CUDA 11.0 Toolkit and the CPU host code is compiled with GCC 7.5.0 with the compilation option of "-std=c++14 -O3" for integration with the Pytorch framework. To measure the performance speedup, we calculate the averaged latency of 200 rounds of end-to-end results.

6.2 Compared with DGL

In this section, we conduct a detailed experimental analysis and comparison with DGL under the different choices of bitwidth. As shown in Figure 7, QGTC achieves 2.20× and 2.13× speedup on average compared to DGL over three types of datasets for cluster GCN and batched GIN, respectively. We also notice that the performance benefit is closely related to the bitwidth we choose, as we can see that from 16-bit to 32-bit the performance shows a large difference compared with 1-bit to 8-bit setting. We next provide detailed analysis and give insights for each type of dataset.

Type I Graphs: Our QGTC performance shows the advantage (an average 3.03× speedup) over the DGL cluster GCN across various settings under different choices of bitwidth. The overall trend of the performance is clear that with a fewer number of bits for both the weights and the node embedding features, QGTC is more likely to reach higher performance. The major reason behind this trend is that a smaller size of bitwidth would lead to less memory access and fewer computations. In the previous design, such as SGQuant [9], such a saving in memory and computation cannot be effectively converted into performance benefits. Because their underlying computation is still carried out in floating point. In contrast, QGTC can successfully harvest such bit-level benefits. DGL reaches an inferior performance due to 1) full-precision (32-bit) computation comes with the high computation complexity compared with our QGTC low-bit design; 2) DGL can only rely on CUDA core for computation which is naturally bounded by the peak computation performance compared with our QGTC design on TC with much higher throughput performance. In comparison with batched GIN, cluster GCN shows higher benefits since it applies the node update first before the neighbor aggregation, this can effectively reduce the data movements and computation overhead.

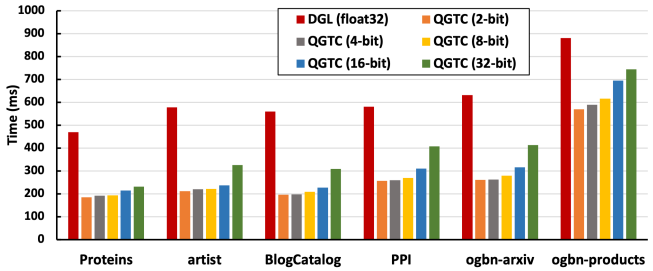

Figure 7: Comparison with DGL on Cluster GCN.

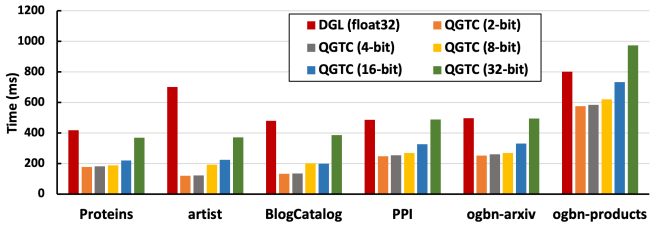

Figure 8: Comparison with DGL on Batched GIN.

Type II Graphs: On the Type II graphs, QGTC achieves relatively higher performance (an average 5.25× speedup) compared with the Type I graphs. There are several reasons. First, compared with the Type I graphs, the second type of graph maintains a higher node average degrees (more neighbors per node). This will lead to a higher non-zero element density in their graph adjacency matrix. It improves the effective computation for each TC tile in the shape of 8×128 bits. Second, from our QGTC system-level optimization perspective, our non-zero tile reuse strategy can be more beneficial, since more edges can be processed in less number of TC tiles and each TC tile can be fully reused once it is loaded from the global memory to thread-local registers. In contrast, DGL working on full-precision data type could not enjoy such benefit.

Type III Graphs: The speedup is also evident (an average 1.75× speedup for cluster GCN and average 1.42× speedup for batched GIN) on graphs with a large number of nodes and edges, such as *ogbn-products*. Compared with the above two types of datasets, Type III datasets achieve relatively lower performance improvements because of two reasons. First, under the same number of partitions, the size of each partition (subgraph) will increase due to the large number of nodes/edges in the original graph. This also increase

the data transfer time between the CPU-GPU for edge list and node embedding features. This, to some degree, would amortize the performance benefits from low-bit GNN computation. Second, the extra cost of input bit-compression for adjacent matrix and node feature embeddings can also increase. Whereas DGL working on the floating-point data type does not need to worry about this part of overhead. On the other side, the above two types of overhead can be well managed by selecting a reasonable number of partitions and batch size during the GNN computation for balancing computation and accuracy performance, as discussed in prior research [5, 35].

6.3 Compared with PyG

In this section, we further compare QGTC with PyG [10] on cluster GCN, another popular GNN computing framework equipped with the highly engineered torch-scatter [11] CUDA library for GNN computation. As shown in Figure 9, QGTC can outperform

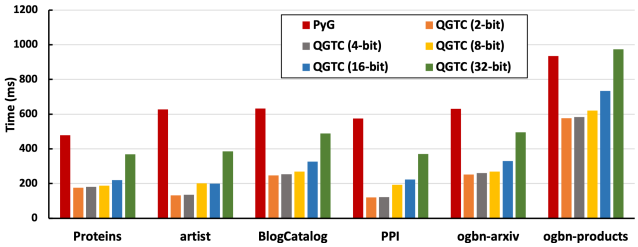

Figure 9: Comparison with PyG on cluster GCN.

PyG with 2.88× speedup on average for cluster GCN. Especially, QGTC achieves more evident speedups on datasets with higher node degree and embedding dimension, such as PPI, since 1) QGTC applies an effective TC acceleration for low-bit execution of GNN which breaks the throughput limits of CUDA core. 2) PyG kernel design uses floating-point arithmetic operations which requires more data movements, therefore, achieving inferior performance compared with our QGTC.

6.4 Additional Studies

Adjacency Matrix Size Impact. In this study, we will demonstrate the subgraph adjacency matrix size impact on the performance of QGTC. Specifically, adjacency matrix size can be controlled by specifying the *number of subgraphs* (in METIS) and *batching size* (in data loader). The size of the adjacency matrix will impact the performance of aggregation in terms of computations and data movements. We use 1-bit for both adjacency matrix and node embedding matrix in this study.

As shown in Figure 10, we can observe that under the same size of D, with the increase of the number of nodes, our major 1-bit GEMM computation kernel would scale up its performance. Note that different colored lines represent different embedding sizes, and we mainly focus on the computation of $\mathbf{AX}$ (*i.e.*, $N \times N \times D$, where N is the number of nodes and D is the node embedding dimension) for neighbor aggregation phase. One specialty of those batched GNN computations *w.r.t.* the traditional NN computation is that batch GNN have more skewed-sized matrices in terms of the ratio between N and D. This, to some degree, limits the achievable

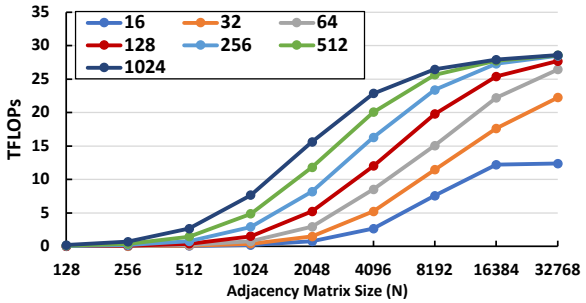

Figure 10: Adjacency matrix size impact on performance.

peak performance on TC. What also worth noticing is that among different lines (different choices of D), the larger D usually leads to better utilization of the GPU, since more computation and memory resources of GPU will become active to achieve higher overall computation throughput.

Zero-Tile Jumping Efficiency. In this experimental study, we evaluate the efficiency of zero-tile jumping in QGTC. Specifically, we would compute the ratio of the non-zero TC tiles (8×128) that are actually involved in our computation versus the total number of TC tiles in the adjacent matrix. As shown in Table 2, we can observe

Table 2: Zero-tile Jumping Efficiency.

Dataset	Base (w/o ZTS)	QGTC (w/ ZTS)	Reduction (%)
PROTEINS_full	5760	1920	66.67%
PPI	10880	3776	65.29%
artist	26880	11584	56.90%
soc-BlogCatalog	94720	34304	63.78%
ogbn-arxiv	97280	6144	93.68%
ogbn-products	360192	59424	83.50%

that our zero-tile jumping technology can largely save the efforts for processing all-zero tiles. Based on our observation, the source of such all-zero TC tiles comes from two levels. The first type of all-zero TC tiles are coming from batching subgraphs. Because there is no edge connection among nodes across different subgraphs. This type of all-zero TC tiles dominates the overall collected number of all-zero tiles. The second type of all-zero tiles come from the missing edge connections between the nodes within each subgraph. While this type of all-zero tiles is minor in its quantity compared with the first type. It still provides us the opportunity to save efforts for memory access and computation.

Layerwise Runtime Decomposition. In this experiment, we decompose a single inference execution time on the 3-layer cluster GCN. As shown in Figure 11, we can notice that the input-

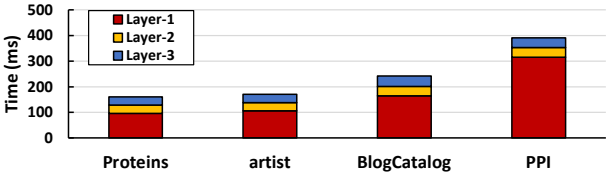

Figure 11: Layerwise runtime decomposition.

layer usually takes the most portion (more than 60%) of the overall execution time. This is mainly because of the quantization and bit-decomposition overhead at the input layer that can only be

executed on the CUDA core. While the actual low-bit execution of the hidden and output layer on TC take less overall execution time.

Non-zero Tile Reuse. In this experiment, we will demonstrate the effectiveness of our non-zero tile reuse by a control-variable study. We eliminate the number of non-zero tiles impact on performance by setting all tiles to non-zero tiles (*i.e.*, filling the initial matrix with all ones). Then we choose the neighbor aggregation process ($\hat{\mathbf{X}} = \mathbf{AX}$) for the study and fix the D to 1024. We change N from 1024 to 8192. Three bit combinations is used in our evaluation, where $\mathbf{A}$ is consistently using 1-bit while $\mathbf{X}$ is using 4, 8, and 16 bit.

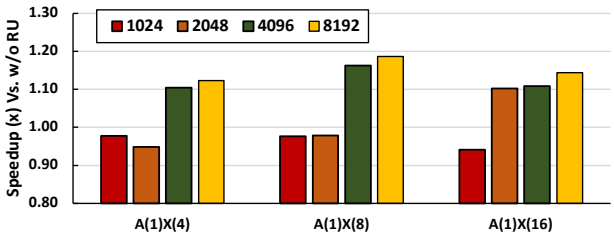

Figure 12: Non-zero Tile Reuse.

As described in Figure 12, our non-zero tile ruse can improve the throughput performance on those large matrix size with the higher number of bits. The major reason behind this is that reuse the non-zero tile can largely reduce the global memory access for fetching the same 1-bit adjacency matrix tile repetitively, which is the key performance bottleneck for those large metrics. The setting (w/o nonzero-tile) reuse shows more advantage on the smaller size matrix because the overhead of recurrent loading the same adjacency matrix tile is not pronounced compared with GEMM operations on TC.

Bitwidth Impact on Performance In this study, we will show the choices of different bitwidth for data representation impact on performance. We focus on the performance impact on the node update phase, which follows the formula ($\hat{\mathbf{X}} = \mathbf{XW}$), where $\mathbf{X}(N \times D)$ and $\hat{\mathbf{X}}(D \times D1)$ are the input node feature embedding matrix and the updated node feature embedding matrix, respectively. $\mathbf{W}(D \times D1)$ is the weight matrix for node feature embedding update. We fix the dimension (D) of $\mathbf{X}$ to 1024 meanwhile changing the bitwidth for both $\mathbf{X}$ and $\mathbf{W}$ under the different numbers of nodes (N). As

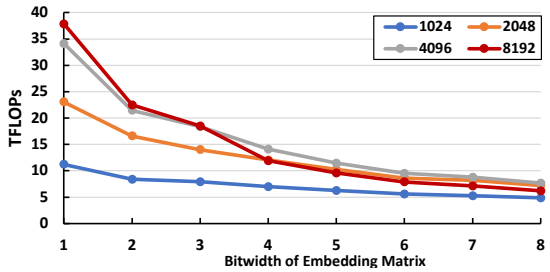

Figure 13: Bitwidth impact on node embedding matrix.

shown in Figure 13, the increase of bitwidth on $\mathbf{X}$ and $\mathbf{W}$ is decreasing the overall performance due to more computation and data movement. This can be more clearly justified at the kernel-level design, where the larger bitwidth will increase the number of iterations in Algorithm 1 from Line 8 to Line 14. Across different N sizes (from 1024 to 8192), we can observe that large-size GEMM computation is more sensitive to the bitwidth increase. The major reason behind this is that the bit-level computation and bit-level data movements of large size matrix would increase more drastically compared with the small size matrix. Meanwhile, when we compute the throughput performance in TFLOPs, we only consider the numbers (represented by several bits) but not the total bits involved. Therefore, the decrease of the TFLOPs would be more significant on large matrices compared with small matrices.

Comparison of CPU-GPU Data Transferring. During the computation of the batched GNNs, the communication between CPU and GPU would take non-trivial costs. Here, we compare two different methods of CPU-GPU communication. The first implementation (**I**) is to separately transfer the dense adjacent matrix and the dense node embedding matrix. The second implementation (**II**) is to separately transfer the sparse adjacent matrix and dense node embedding. It then calls `sparse_to_dense` on GPU for adjacent matrix initialization. The third implementation (**III**) used by QGTC is to pack the sparse adjacent matrix and dense node embedding matrix first and then transfer the packed data. Once it finishes transferring, it will call `sparse_to_dense` on GPUs for adjacent matrix initialization. We select two *ogb* datasets for study in this experiment. As shown in Figure 14, the third implementation used in QGTC consistently outperforms Implementation I with an average 15.51× and implementation II with an average 1.54× speedup. There are two major reasons. First, separate transferring of the edge list and embedding matrix would lead to the overhead of initiating one additional PCIe data communication, which goes through a slow warm-up process. Second, separate transferring hard to maximize the PCIe bandwidth utilization.

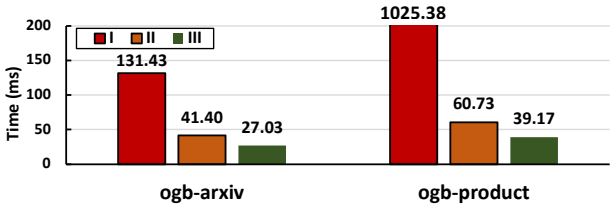

Figure 14: CPU-GPU data transferring comparison.

7 CONCLUSION

In this paper, we propose the first QGNN computing framework, QGTC, which supports any-bitwidth computation via GPU Tensor Core. Specifically, we introduce the first GNN-tailored any-bitwidth arithmetic design that can emulate different bitwidth computations to meet the end-users demands. Second, we craft a TC-tailored CUDA kernel design by incorporating 3D-stacked bit compression, zero-tile jumping, and non-zero tile reuse technique to maximize the performance gains from GPU Tensor Core. Third, we incorporate an effective bandwidth-optimized subgraph packing strategy to maximize the data transferring efficiency. Finally, we integrate QGTC with the Pytorch framework for better programmability and extensibility. Extensive experiments demonstrate significant performance gains over the state-of-the-art Deep Graph Library framework across various settings.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. [n.d.]. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*.

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

[3] Mehdi Bahri, Gaétan Bahl, and Stefanos Zafeiriou. 2020. Binary Graph Neural Networks. *arXiv preprint arXiv:2012.15823* (2020).

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web (WWW)*.

[5] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

[6] E. Cuthill and J. McKee. 1969. Reducing the Bandwidth of Sparse Symmetric Matrices. In *Proceedings of the 1969 24th National Conference (ACM)*.

[7] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. [n.d.]. Accelerating Reduction and Scan Using Tensor Core Units. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*.

[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] Boyuan Feng, Yuke Wang, Xu Li, Shu Yang, Xueqiao Peng, and Yufei Ding. [n.d.]. SGQuant: Squeezing the Last Bit on Graph Neural Networks with Specialized Quantization. In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*.

[10] 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)*.

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

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

[13] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[14] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Proceedings of the 30th international conference on neural information processing systems*. Citeseer.

[15] Konstantinos I Karantasis, Andrew Lenharth, Donald Nguyen, Mara J Garzaran, and Keshav Pingali. 2014. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE.

[16] George Karypis and Vipin Kumar. 2009. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0. http://www.cs.umn.edu/~metis.

[17] 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

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

[19] 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).

[20] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. [n.d.]. Neugraph: parallel deep neural network computation on large graphs. In *2019 {USENIX} Annual Technical Conference (USENIX ATC 19)*.

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

[22] NVIDIA. [n.d.]. CUDA Template Library for Dense Linear Algebra at All Levels and Scales (CUTLASS).

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

[24] 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

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

[26] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit

[27] 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) 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alch'e-Buc, E. Fox, and R. Garnett (Eds.).

[28] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E* (2007).

[29] Shyam A Tailor, Javier Fernandez-Marques, and Nicholas D Lane. 2021. Degree-Quant: Quantization-Aware Training for Graph Neural Networks. *International Conference on Learning Representations (ICLR)* (2021).

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

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

[32] 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).

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

[34] 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)*.

[35] Hanqing Zeng and Viktor Prasanna. [n.d.]. Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.