TANGO: rethinking quantization for graph neural network training on GPUs

Shiyang Chen
Rutgers, The State University of New Jersey

Da Zheng*
Amazon

Caiwen Ding
University of Connecticut

Chengying Huan
Institution of Software, Chinese Academy of Sciences

Yuede Ji
University of North Texas

Hang Liu
Rutgers, The State University of New Jersey

ABSTRACT

Graph learning is becoming increasingly popular due to its superior performance in tackling many grand challenges. While quantization is widely used to accelerate Graph Neural Network (GNN) computation, quantized training faces remarkable roadblocks. Current quantized GNN training systems often experience longer training time than their full-precision counterparts for two reasons: (i) addressing the quantization accuracy challenge leads to excessive overhead, and (ii) the optimization potential exposed by quantization is not adequately leveraged. This paper introduces TANGO which re-thinks quantization *challenges* and *opportunities* for graph neural network training on GPUs with three contributions: Firstly, we introduce efficient rules to maintain accuracy during quantized GNN training. Secondly, we design and implement quantization-aware primitives and inter-primitive optimizations to speed up GNN training. Finally, we integrate TANGO with the popular Deep Graph Library (DGL) system and demonstrate its superior performance over the state-of-the-art approaches on various GNN models and datasets.

ACM Reference Format:
Shiyang Chen, Da Zheng, Caiwen Ding, Chengying Huan, Yuede Ji, and Hang Liu. 2023. TANGO: rethinking quantization for graph neural network training on GPUs . In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3607037

1 INTRODUCTION

Graph analytics can claim a large share of the credit for tackling many grand challenges of our time – such as understanding the spread of pandemics [1], designing extremely large-scale integrated circuits [2], and uncovering software vulnerabilities [3], among many others [4–11]. In particular, since the introduction of the Graph Convolution Network (GCN) by Kipf and Welling in 2016 [12], GNNs have gained widespread popularity as a vibrant field in graph analytics. This is primarily because various GNN models have shown promising results in addressing these challenges through node and edge embedding-based designs. [13–21].

A typical GNN model often performs *linear transformation*, and *graph structure related operations* on node feature (**H**), edge feature (**E**), and graph structure (**G**). Using Graph Attention Network (GAT) [22] (refer to Figure 1, next page) as an example, this model (i) applies a multilayer perception on node feature matrix, (ii) uses graph topology to derive the edge features, and (iii) calculates the destination feature by considering both the source node and edge features. One could further extend the aforementioned steps (i) - (iii) to multi-hop neighbors to derive multi-layer GATs. Of note, step (i) is a GEneral Matrix Multiply (GEMM) primitive, while steps (ii) and (iii) are sparse primitives, i.e., SParse-dense Matrix Multiplication (SPMM) and Sampled Dense-Dense Matrix product (SDDMM), whose sparse nature is usually defined by the graph.

Quantization is a primary approach to accelerating Deep Neural Network (DNN) and GNN models for two major optimizations *opportunities*, that is, quantization could lead to both computation and data access reductions. On the one hand, for computation-intensive primitives, e.g., GEMM, computing on quantized data is faster than on a floating-point counterpart. For instance, computing with 8-bit integers on tensor core offers 2× the throughput of 16-bit floating-point and 32× that of 32-bit floating-point, respectively [23]. On the other hand, for sparse primitives, i.e., SPMM and SDDMM, which are data-intensive, quantization reduces the size of tensors, thus reducing memory traffic and time consumption.

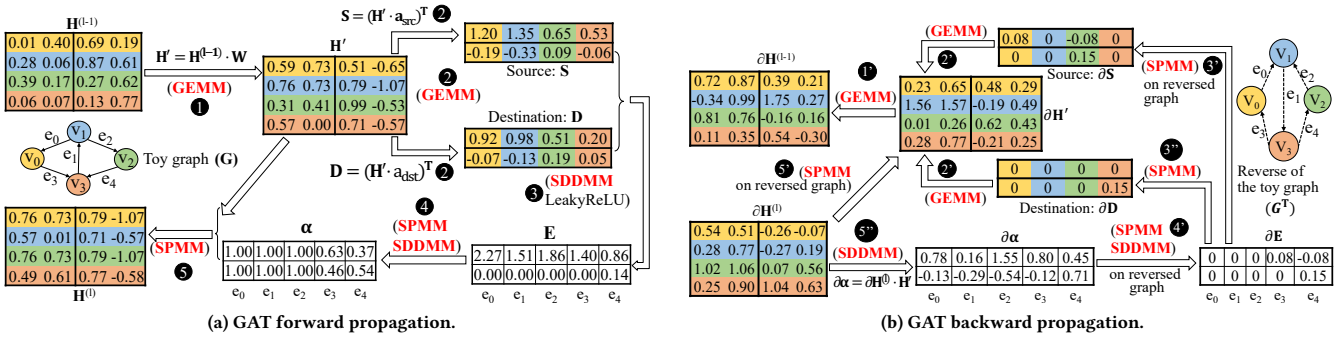
Whereas the *challenge* is that quantization errors (i.e., caused by fewer bits) could prevent the model from achieving the desired accuracy. Correspondingly, there mainly exist three tracks of research efforts: (i) For multiply-and-accumulate operations in matrix multiplication, limited precision can cause values of different magnitudes to accumulate inaccurately. The proposed solutions are chunk-based accumulation [24] and dynamically adjustable data formats [25–27]. (ii) For weight updates in backpropagation, quantization errors could negate the gradient update to the weights. SWALP [28] proposes accumulating and updating the weight after multiple training epochs. (iii) To mitigate divergence from quantization errors, Zhu et al. [29] propose heuristics for gradient clipping and learning rate adjustments.

Contemporary quantized GNN training systems often experience longer training time than their full-precision counterparts for two reasons: *(i) addressing the accuracy challenge results in significant overhead.* The computation of the proposed novel data formats is inefficient because commodity GPUs support neither fixed-point

*The work is not related to the author's position at Amazon.

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607037



(a) GAT forward propagation. (b) GAT backward propagation.

Figure 1: GAT training on a toy graph, i.e., middle left of (a), with two heads. This example is used throughout this paper.

data format nor floating-point format with dynamically adjusted exponent and mantissa. The model trained using SWALP or clipped gradients needs more epochs to converge because the weights are updated less frequently. *(ii) The optimization potential offered by quantization is not well-utilized.* ActNN [30], TinyKG [31], and EXACT [32] quantize the tensors to save memory and dequantize them back to full-precision for computation, increasing the overall training time [33, 34]. For example, TinyKG with 8-bit quantization is 54.1% slower than using FP32. Of note, Degree-Quant [35] performs Quantization-Aware Training (QAT) [36–42] that uses full-precision to "simulate quantization" in training to reduce the error for quantized inference, which, again, experiences longer training time.

This paper introduces Tango, the first GPU-based quantized GNN training system that both *maintains the model accuracy* and *reduces turnaround time* when compared to the full precision counterpart. Particularly, Tango encompasses three contributions:

- We introduce several lightweight rules to maintain accuracy for quantized GNN training. The rules include GPU-accelerated stochastic rounding, derivation of proper quantization bit count, novel quantization-aware GEMM, SPMM, and SDDMM, and full precision weight update and softmax.
- We design and implement a quantization-aware system to reduce the GNN training time on GPUs. Our techniques include GEMM with on-the-fly quantization, incidence-matrix-based adaptive SPMM, SDDMM with on-the-fly dequantization, and inter-primitive optimizations.
- For ease of use, we integrate Tango with DGL, which uses PyTorch as the backend. Therefore, all existing DGL-based models can enjoy the performance benefits from Tango without any changes. We demonstrate that Tango constantly outperforms state of the art for all evaluated GNN models while maintaining the training accuracy.

The remainder of this paper is organized as follows: Section 2 presents the background. Section 3 discusses the design and techniques in Tango. Specifically, Section 3.1 describes the challenges and opportunities of Tango, Section 3.2 illustrates the lightweight rules for maintaining training accuracy during quantization, and Section 3.3 presents the systematic effort on quantization accelerated training. We evaluate Tango in Section 4, describe related work in Section 5, and conclude in Section 6.

2 BACKGROUND

2.1 A running example for GAT training

This section uses a running example to illustrate how to express the forward and backward computations of GAT with three key primitives, i.e., GEMM, SPMM, and SDDMM.

Primitives for forward computation. Figure 1a presents the forward workflow of GAT on a toy graph, i.e., node projection (❶ - ❷), attention computation (❸ - ❹), and message aggregation (❺).

In step ❶, GAT resorts to *GEMM primitive* to perform a linear transformation for node features, i.e., $\mathbf{H'} = \mathbf{H}^{(l-1)} \cdot \mathbf{W}$. In $\mathbf{H}^{(l-1)}$, where each row of $\mathbf{H}^{(l-1)}$, in the same color, represents the features of one node. Each node feature contains two heads in $\mathbf{H}^{(l-1)}$. Using the first row of $\mathbf{H}^{(l-1)}$ as an example, [0.01, 0.40] is the first head, and the remaining two values belong to the second head. $\mathbf{W}$ is the learnable weight matrix for linear transformation.

In step ❷, GAT consolidates each head of the feature vector into one scalar by GEMM, i.e., $\mathbf{S} = (\mathbf{H'} \cdot \mathbf{a}_{src})^T$ and $\mathbf{D} = (\mathbf{H'} \cdot \mathbf{a}_{dst})^T$. Using node $\mathbf{v}_0$ of ❷ as an example, for the source feature, [0.59, 0.73]×[0.91, 0.90]^T = 1.20, and [0.51, -0.65]×[0.42, 0.62] = -0.19. Similarly, we can derive the entire $\mathbf{S}$ and $\mathbf{D}$.

In step ❸, the source ($\mathbf{S}$) and destination ($\mathbf{D}$) node feature matrices are combined by an *SDDMM primitive* to arrive at the edge feature $\mathbf{E}$. Formally, the SDDMM is defined as $\mathbf{E} = \mathbf{G} \odot (\mathbf{S} \oplus \mathbf{D}^T)$, where every row of $\mathbf{S}$ computes against every row of $\mathbf{D}$ with the customized operation $\oplus$, and $\odot$ is a Hadamard product operator. The resultant matrix $\mathbf{E}$ is masked out by the sparse adjacency matrix $\mathbf{G}$ of the graph so $\mathbf{E}[i][j] = 0$ when there is no edge between v_i and v_j. In Figure 1a, $\oplus$ denotes addition. Using edge e_3 as an example, since it connects source v_0 and destination v_3, we arrive at [1.20, -0.19] + [0.20, 0.05] = [1.40, -0.14] for e_3. Then an element-wise LeakyReLU is applied to the edge features. Particularly, each non-negative entry in $\mathbf{E}$ is unchanged while negative ones become close to 0, which we use 0 to represent. Hence [1.40, -0.14] becomes [1.40, 0.00] in Figure 1a.

In step ❹, edges of the same destination come together to compute the head-wise attention scores through softmax operation.

Using node v_3 as an example, it has e_3 and e_4 as the incoming edges. Therefore, the attention scores of e_3 and e_4 are computed as 0.63 = $\frac{e^{1.40}}{e^{1.40}+e^{0.86}}$, and 0.46 = $\frac{e^{0}}{e^{0}+e^{0.14}}$ for e_3, and 0.37 = $\frac{e^{0.86}}{e^{1.41}+e^{0.86}}$ and 0.54 = $\frac{e^{0.14}}{e^{0}+e^{0.14}}$ for e_4. Putting this example into a general formula, we use SPMM and SDDMM operations together to compute the denominator as follows: First, we use *SPMM* to aggregate the in-edges for every node such as $\mathbf{M'} = (\mathbf{G} \odot exp(\mathbf{E})) \cdot \mathbf{1}$. Of note, $\mathbf{1}$ is an all '1' dense matrix. Second, since the first step computed the denominator for each destination vertex, we use *SDDMM* to assign this denominator back to each incoming edge via $\mathbf{E'} = \mathbf{G} \odot (\mathbf{1} \cdot \mathbf{M'}^T)$. Subsequently, $\boldsymbol{\alpha} = \frac{exp(\mathbf{E})}{\mathbf{E'}}$.

In step ❺, GAT performs an *SPMM* to derive the new node embedding via $\mathbf{H}^{(l)} = (\mathbf{G} \odot \boldsymbol{\alpha}) \cdot \mathbf{H'}$. Intuitively, this step derives the new embedding by computing the weighted sum of all the incoming neighbors to a destination vertex. Using v_3 as an example, its incoming neighbors are $\{v_0, v_2\}$. We arrive at $\mathbf{H}^{(l)}[v_3] = \boldsymbol{\alpha}[e_3] \cdot \mathbf{H'}[v_0] + \boldsymbol{\alpha}[e_4] \cdot \mathbf{H'}[v_2]$, resulting in [0.49, 0.61, 0.77, -0.58].

Primitives for backward computation. Figure 1b is the backward pass of Figure 1a. Steps ❺' (*SPMM*) and ❺'' (*SDDMM*) of Figure 1b are the corresponding backward operations for step ❺ in Figure 1a. In the forward SPMM operation (❺), $\mathbf{H}^{(l)} = (\mathbf{G} \odot \boldsymbol{\alpha}) \cdot \mathbf{H'}$, we hence disperse the gradients of $\mathbf{H}^{(l)}$ to both $\mathbf{H'}$ and $\boldsymbol{\alpha}$. First, we arrive at $\partial\mathbf{H'} = (\mathbf{G}^T \odot \boldsymbol{\alpha}) \cdot \partial\mathbf{H}^{(l)}$, which is an *SPMM* (❺') on the reversed graph since the updated node feature $\mathbf{H}^{(l)}$ is aggregated from the source node feature. Using $\partial\mathbf{H}[v_1]$ as an example, it receives gradients from both e_0 and e_2. Therefore, we arrive at $\partial\mathbf{H}^{(l-1)}[v_1] = \boldsymbol{\alpha}[e_0] \cdot \partial\mathbf{H}^{(l)}[v_0] + \boldsymbol{\alpha}[e_2] \cdot \partial\mathbf{H}^{(l)}[v_2]$. That is, [1.56,1.57,-0.19,0.49]= 1.0×[0.54, 0.51] + 1.0×[1.02,1.06] $\|_{concat}$ 1.0×[-0.26, -0.07] + 1.0×[0.07,0.56]. Second, we have $\partial\boldsymbol{\alpha} = \mathbf{G} \odot (\partial\mathbf{H}^{(l)} \cdot \mathbf{H'}^T)$, which is an SDDMM operator on the original graph, where it performs row-wise dot-product (❺''). Using $\partial\boldsymbol{\alpha}[e_0]$ as an example, it connects nodes v_1 and v_0, we arrive at $\partial\boldsymbol{\alpha}[e_0] = \partial\mathbf{H}^{(l)}[v_0] \cdot \mathbf{H'}[v_1]$. In the example, we get [0.78,-0.13] =[0.54,0.51] × [0.76,0.73] $^T\|_{concat}$[-0.26,-0.07]×[0.79,-1.07]^T.

Step ❹' computes the gradient of edge features using attention scores. Using e_3 as an example, we compute $\partial\mathbf{E}[e_3] = \boldsymbol{\alpha}[e_3](\partial\boldsymbol{\alpha}[e_3] - (\partial\boldsymbol{\alpha}[e_3]\boldsymbol{\alpha}[e_3] + \partial\boldsymbol{\alpha}[e_4]\boldsymbol{\alpha}[e_4]))$ based on the derivative of softmax operation. We first use *SPMM* to aggregate the incoming edge features for every node $\mathbf{P} = (\mathbf{G} \odot \partial\boldsymbol{\alpha} \odot \boldsymbol{\alpha}) \cdot \mathbf{1}$. For example, $\partial\boldsymbol{\alpha}[e_3]\boldsymbol{\alpha}[e_3] + \partial\boldsymbol{\alpha}[e_4]\boldsymbol{\alpha}[e_4]$ is the aggregation on v_3 as 0.80×0.63 + 0.45×0.37 = 0.67 for the first head. Then we compute the final gradient for every edge with *SDDMM*, $\partial\mathbf{E} = \boldsymbol{\alpha} \odot (\partial\boldsymbol{\alpha} - (\mathbf{G}^T \odot (\mathbf{P} \cdot \mathbf{1}^T)))$. That is, every node feature $\mathbf{P}$ is assigned to their out-edges in the reversed graph, and then computed with $\partial\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}$. The gradient of the first head of e_3 is 0.63 × (0.80 - 0.67) = 0.08.

In step ❸' and ❸'', the gradient of edge attention score is used to compute the source feature and destination feature with two *SPMM* operations, $\partial\mathbf{S} = (\mathbf{G}^T \odot \partial\mathbf{E}) \cdot \mathbf{1}$ and $\partial\mathbf{D} = (\mathbf{G} \odot \partial\mathbf{E}) \cdot \mathbf{1}$, where nodes aggregate their out-edge and in-edge attention scores, respectively. Still use v_3 as an example, its gradient $\partial\mathbf{S}[v_3] = \partial\mathbf{E}[e_1] = [0, 0]$ and $\partial\mathbf{D}[v_3] = \partial\mathbf{E}[e_3] + \partial\mathbf{E}[e_4] = [0, 0.15]$. The gradients from multiple out-edges are accumulated.

Note steps ❷' and ❶', which do not depend on the graph structure, will follow the traditional DNN back propagation method for gradient computation. We skip the details.

2.2 GNN models

There exist a variety of GNN models. Graph Convolutional Network (GCN) [12] derives a graph convolutional operator through spectral graph theory. It can be expressed by GEMM and SPMM operations. Later, GraphSAGE [13] uses sampling to encode the graph topology for inductive learning. The model is applicable for unseen nodes because it learns the feature from sampled sub-graph. *GraphSAGE can be implemented with GEMM and SPMM.* GAT [22] further introduces graph attention mechanisms that can attend to various neighbors with weights. This model contains GEMM, SPMM, and SDDMM primitives. Later, Relational GCN (RGCN) extends GCN via assigning different parameters to edges with different types [43, 44]. *Here, RGCN consists of GEMM and SPMM primitives.* HGT proposes a transformer-based GNN model for heterogeneous graphs [45]. It contains different parameters for distinct edge and node types. *This model includes GEMM, SDDMM, and SPMM primitives.*

We study two GNN models, i.e., GCN and GAT, for two reasons: (i) These two models are the most popular and cover all the required primitives for most GNN models. (ii) These two models contain relatively large and complete training and testing datasets.

2.3 Quantization

For a collection of values $\mathbf{X} = \{\mathbf{X}_i \mid \mathbf{X}_i \in [\mathbf{X}_{min}, \mathbf{X}_{max}]\}$ which are represented in full precision, quantization uses fewer number of bits (i.e., B) to represent each $\mathbf{X}_i$. Quantization scatters $\mathbf{X}$ into $2^B - 1$ buckets. Subsequently, all the $\mathbf{X}_i$'s in the same bucket are represented as the same value, i.e., the bucket value.

For *uniform* quantization, we assign each bucket the same value range, that is, $s = \frac{\alpha-\beta}{2^B-1}$, where $[\alpha, \beta]$ is the clipping range of $\mathbf{X}$. There are also *nonuniform quantization* whose quantized values are not necessarily uniformly spaced. If one wants to include the entire value range of $\mathbf{X}$, one needs $\alpha = \mathbf{X}_{min}$, and $\beta = \mathbf{X}_{max}$. Formally,

$$\mathbf{X}_{i,Quant} = round(\frac{\mathbf{X}_i}{s}) - Z, \tag{1}$$

where $Z = \frac{\alpha+\beta}{2}$ is the zero point after quantization. One can recover the original value $\mathbf{X}_i$ by dequantizing $\mathbf{X}_{i,Quant}$:

$$\mathbf{X}_i \approx s \cdot (\mathbf{X}_{i,Quant} + Z). \tag{2}$$

Quantization further includes the following three configurations: (i) *Asymmetric* vs. *symmetric* quantization. Particularly, for s, one can let $-\alpha$=β=max($|\mathbf{X}_{max}|$, $|\mathbf{X}_{min}|$). While asymmetric quantization will likely enjoy a more precise clipping range when compared to symmetric quantization, the latter design, however, simplifies the quantization function in Equation 1 as $Z = \frac{\alpha+\beta}{2} = 0$. (ii) *Quantization granularity* concerns about the size of $\mathbf{X}$. Using a matrix as an example, we can extract the same s for the entire matrix or one s per row/column of a matrix. The latter has a finer granularity than the former. (iii) *Static vs dynamic* quantization determines whether we change s for the same tensor $\mathbf{X}$ from iteration to iteration. The dynamic version does so, while the static one does not. In Tango, we adopt *symmetric*, *tensor-level granularity*, *dynamic* quantization to maintain training accuracy and enhance training speed.

3 TANGO: AN ACCURACY AND SPEED CO-DESIGNED QUANTIZATION SYSTEM

3.1 Tango overview

Our key observation is that quantization presents both *challenges* and *opportunities* for GNN training. Tango aims to tackle the challenges efficiently while extracting quantization benefits as follows:

Challenge: Maintaining training accuracy poses three issues: (i) quantization could introduce additional computation tasks in addition to the steps in Figure 1. We need to reduce the overhead brought by those additional computations. (ii) For various operations in GNN training (in Figure 1), we need to decide what operations should be quantized and how we should quantize the tensors in each operator to meet the training accuracy requirements. In addition, (iii) those rules should expose optimization opportunities for Tango to accelerate the most time-consuming operations in GNN training with quantization.

In this paper, (i) we introduce GPU-friendly stochastic rounding and a lightweight operation to determine the required # of quantization bits, reducing the cost of meeting accuracy requirements. (ii) We determine that weight update and softmax operations should be performed in full precision, while GEMM, SPMM, and SDDMM can be performed in our novel quantization-aware manner. This minimizes the impact on training accuracy while providing critical optimization opportunities for reducing turnaround time. Notably, (iii) GEMM, SPMM, and SDDMM are the most time-consuming phases in GNN, and our quantization-aware design offers optimization opportunities (see below) to reduce computation costs in GEMM and memory costs for SPMM and SDDMM.

Opportunity: Accelerating training by quantization. Quantization offers two avenues to improve training speed, that is, higher computation throughput and less memory traffic with values in lower precision. We use quantized computing to accelerate the most computation-intensive primitives and operations, i.e., GEMM, SPMM and SDDMM. *However, the problem is that these primitives are highly optimized and fine-tuned by commercial libraries.* And CUBLAS GEMM and cuSPARSE SPMM, and SDDMM are closed-source. Integrating our proposed optimizations into these kernels and achieving the desired speedup is extremely challenging.

In this paper, (i) we utilize our novel quantization-aware GEMM to reduce computation time. Moreover, we identify an optimal tiling strategy to overlap the on-the-fly quantization of the matrix with the subsequent quantized computations. (ii) To address the memory-intensive nature of SPMM and SDDMM, we sequentially quantize the input tensor and write the quantized value in memory. The computation then randomly accesses the smaller quantized tensor, which provides better cache behavior than direct random access to full-precision tensors.

3.2 Lightweight rules for maintaining training accuracy during quantized training

GPU-accelerated stochastic rounding. We adopt stochastic rounding to reduce the quantization error [46], with which the expectation of the quantization error should be 0 statistically. In particular, given a scaled floating-point number x between $[-2^{B-1}-1, +2^{B-1}-1]$ as the range of B-bit integers, we round it to integer based on:

$$x_{Quant} = \begin{cases} floor(x)+1, & w/probability \quad x - floor(x); \\ floor(x), & w/probability \quad 1-(x-floor(x)). \end{cases} \tag{3}$$

We design and implement a GPU-accelerated pseudo-random number generator to facilitate fast stochastic rounding, which is ~20× faster than the native cuRAND random number generator on GPU [47]. Our key optimization is storing random generator states in GPU registers as opposed to in global memory, which is the case in the existing cuRAND library [47]. Since the random number generator is a memory-bound operation, this optimization helps significantly improves the throughput. Of note, because cuRAND is closed-source, we cannot directly integrate this optimization into cuRAND. We thus implement our generator based upon xoshiro256++[48] with our memory optimizations.

Lightweight rule for deriving # of desired quantization bits. We develop a metric to measure the quantization error, which subsequently helps derive the # of desired quantization bits. During quantization, a value $\mathbf{X}_i$ will be rounded to one of the quantization grid points $\mathbf{X}_{i,Quant}$. We introduce the following metric to estimate the quantization error of a tensor $\mathbf{X}$:

$$Error_{\mathbf{X}} = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{\mathbf{X}_i - \mathbf{X}_{i,Quant}}{\mathbf{X}_i + \mathbf{X}_{i,Quant} + \epsilon}\right|, \tag{4}$$

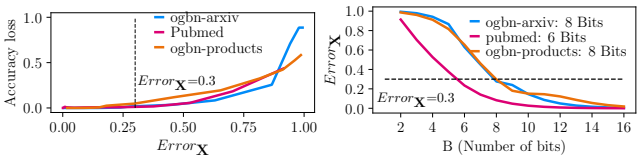
where N is the number of elements in the tensor.

Intuitively, $Error_{\mathbf{X}}$ derives the relative quantization error of a tensor $\mathbf{X}$, where the numerator, i.e., $\left|\mathbf{X}_i - \mathbf{X}_{i,Quant}\right|$ is the absolute quantization error while the denominator is the sum of $\mathbf{X}_i$, $\mathbf{X}_{i,Quant}$, and ϵ. The denominator needs the sum of the three values for two reasons: (i) a small ϵ to avoid dividing by zero, i.e., when $\mathbf{X}_i = \mathbf{X}_{i,Quant} = 0$. Tango chooses $\epsilon = 0.0005$. Of note, Tango does not experience $\mathbf{X}_i + \mathbf{X}_{i,Quant} = 0$ when $\mathbf{X}_i \neq 0$ and $\mathbf{X}_{i,Quant} \neq 0$ because our quantization is symmetric. (ii) If we use ϵ with only either $\mathbf{X}_i$ or $\mathbf{X}_{i,Quant}$ as the denominator, we could suffer from quantization error divided by ϵ. This would lead to an extremely large relative error for a particular $\mathbf{X}_i$, overshadowing the relative quantization error of other $\mathbf{X}_i$'s.

Our proposed quantization error metric in Equation 4 is a relative error thus inductive. That is, this parameter could be used to compare the quantization error across tensors. Therefore, we can tune a desired $Error_{\mathbf{X}}$ that is generally applicable for various tensors. The value range of $Error_{\mathbf{X}}$ is [0, 1]. Particularly, if $\mathbf{X}_i$ has no rounding error, the corresponding error is 0. When the rounding error of $\mathbf{X}_i$ is significant, the term approaches 1.

We leverage Equation 4 to select the desired number of quantization bits as follows: we compute $Error_{\mathbf{X}}$ of the output tensor of the first GNN layer with quantization. Note that we do not apply this metric to the input tensor of the first layer because its quantization error can be recovered by learning from the graph structure [49]. We also want to mention that the training process could potentially amend the quantization error when the bit count is even lower. Our bit count derivation metric derives a lower bound bit count that could maintain the training accuracy.

As shown in Figure 2a, our heuristic demonstrates that when $Error_{\mathbf{X}} < 0.3$, Tango can maintain the accuracy requirement across various datasets. Therefore, we let $Error_{\mathbf{X}} = 0.3$ across all datasets.



(a) Training accuracy vs $Error_{\mathbf{X}}$. **(b) $Error_{\mathbf{X}}$ vs B.**

Figure 2: The accuracy for different $Error_{\mathbf{X}}$ and the required number of bits to retain the desired $Error_{\mathbf{X}}$ for ogbn-arxiv, Pubmed, and ogbn-products datasets.

Figure 2b shows that the desired number of bits for ogbn-arxiv, Pubmed, and ogbn-products are 8, 6, and 8, respectively.

The benefit of our metric is as follows: because our lightweight rule calculates the $Error_{\mathbf{X}}$ solely for the first layer during the initial epoch. In contrast, determining the accuracy loss typically necessitates training the model until convergence (i.e., all epochs). The effectiveness of our approach is demonstrated by our empirical findings presented in Fig 2(a), which shows that $Error_{\mathbf{X}} <= 0.3$ is a general threshold to maintain the accuracy across datasets.

Novel quantization-aware matrix multiplication with scaling factor computation. Since the majority of the tensor primitives in GNN are either dense matrix multiplication or a variant of it, the accuracy analysis would be similar across these primitives. We hence restrict our accuracy analysis to dense matrix multiplication (i.e., GEMM) for brevity.

For two reasons, the resultant matrix of a quantized matrix multiplication has to be of higher precision. First, the result of a multiplication operation between two 8-bit integers could go beyond the value range of an 8-bit integer. Second, the subsequent accumulation of the multiplied values can again push the value beyond the range of an 8-bit integer.

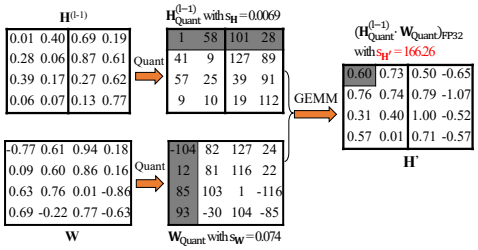


Figure 3: Quantization for GEMM of step ❶ in Figure 1a.

Figure 3 presents this problem when we perform $\mathbf{H}^{(l-1)} \cdot \mathbf{W}$ in quantized mode. After quantization, the first row of $\mathbf{H}^{(l-1)}_{Quant}$, i.e., [1 58 101 28] multiplies with the first column of $\mathbf{W}_{Quant}$, i.e., [-104 12 85 93]^T experiences both issues mentioned above. In fact, all the entries in the resultant matrix $(\mathbf{H}^{(l-1)}_{Quant} \cdot \mathbf{W}_{Quant})_{int32}$ exceed the 8-bit range of $[-127, 127]$. Therefore, we opt for a 32-bit data format to store the result to avoid this overflow problem. The good news is that storing the results in 32-bit integers introduces negligible overheads on commodity GPUs. Also, note that recent tensor core units on NVIDIA GPUs force the resultant matrix to be a 32-bit integer matrix for the input of two 8-bit integer matrices.

To reduce the quantization overheads, TANGO directly dequantizes the GEMM results, i.e., $\mathbf{H}'$ into FP32 after computing the resultant matrix with our optimizations. In the meantime, TANGO also derives the scaling factor $s_{\mathbf{H}'}$=166.26 during the quantized GEMM operation, as shown in Figure 3. This design avoids a dedicated dequantization kernel, a scaling factor computation kernel, and the associated expensive global memory accesses.

Full precision weight update. To combat the round-off error, we update the model weights with dequantized FP32 gradients. The reason is that the magnitude of the model weight is often significantly larger than the gradients. In addition, the small learning rate further amplifies the difference. Previously, existing projects use shared exponent [27], Flexpoint [25], or delayed updates [50] to tackle the round-off error. Unfortunately, these designs could suffer from shortcomings of delayed convergence, unavailability on commodity GPUs, being slow to implement on GPUs, or multiple of these shortcomings [51, 52]. Of note, although the updated FP32 weights are quantized into 8-bit integers in the next iteration, quantizing the updated weights into 8-bit integers is often better than directly updating the quantized weights with quantized gradients as elaborated below.

Assuming $W_{full} = W_{quant} + W_{round\_off}$ and $\Delta W_{full} = \Delta W_{quant} + \Delta W_{round\_off}$, where W_{full} and ΔW_{full} are weights and update values (e.g. gradients) of full precision, respectively. W_{quant} and ΔW_{quant} are the output from the quantization function Q. W_{round_off} and ΔW_{round_off} are the correspondingly round-off errors. Below is our analysis:

$$Q(W_{full}) + Q(\Delta W_{full}) = W_{quant} + \Delta W_{quant}. \tag{5}$$

If we add the full precision before quantization, we arrive at:

$$\begin{aligned} Q(W_{full} + \Delta W_{full}) \sim\ & W_{quant} + \Delta W_{quant} \\ & + Q(W_{round\_off} + \Delta W_{round\_off}). \end{aligned} \tag{6}$$

One can observe that Equation 6 offers higher accuracy than Equation 5 as $Q(W_{round\_off} + \Delta W_{round\_off})$ curbs the round off error.

Full precision for the layer before Softmax. The Softmax layer amplifies the quantization error due to its exponential operations. For simplicity, we consider a layer before Softmax with two outputs z_0 and z_1. The difference in Softmax score (D) between the two outputs is:

$$D = \frac{\frac{exp(z_0)}{exp(z_0)+exp(z_1)}}{\frac{exp(z_1)}{exp(z_0)+exp(z_1)}} = \frac{exp(z_0)}{exp(z_1)}. \tag{7}$$

Once the quantization error e_i is introduced to z_i, the perturbation of output difference follows:

$$D' = \frac{exp(z_0+e_0)}{exp(z_1+e_1)} = D \cdot \frac{exp(e_0)}{exp(e_1)} = D \cdot \underbrace{exp(e_0 - e_1)}_{\text{Amplified error}}. \tag{8}$$

This analysis suggests that the exponential function applied to $(e_0 - e_1)$ will rapidly make the difference D either bigger or smaller, departing from the desired faithful D. Therefore, we propose to use full precision to compute the layer before the Softmax.

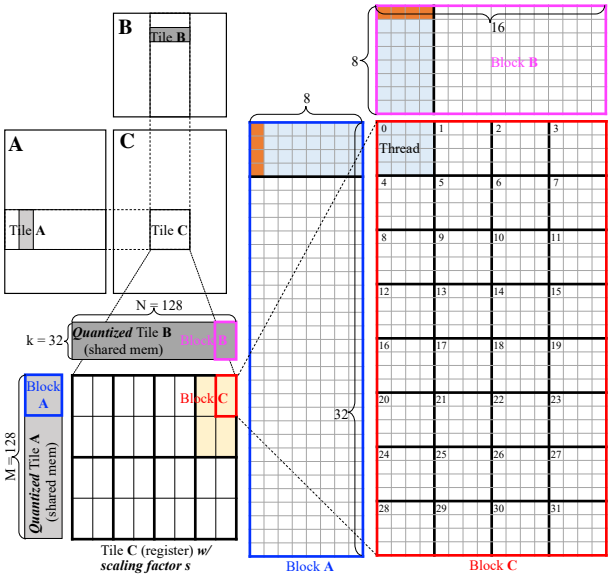


Figure 4: TANGO GEMM with on-the-fly quantization and scaling factor s computation.

3.3 Quantization accelerated training

GEMM with on-the-fly quantization. Figure 4 illustrates our GEMM with on-the-fly quantization and scaling factor (i.e., s) computation. Our quantized GEMM includes four steps: First, we quantize while loading the tiles of input matrices from global memory to shared memory (i.e., Tiles **A** and **B**). Note that the input matrices are usually needed for backward computation. Therefore, we store the quantized tiles back in global memory while computing, eliminating the round-trip memory latency in the naive design. Second, we store the resultant block in registers to minimize the write latency. Third, during computation, we pack four 8-bit integers into a 32-bit register and use one **DP4A** instruction for four multiply-accumulate operations between two packed registers. Fourth, we dequantize the resultant 32-bit integers in registers to floating-point. We also fuse the computation of parameter s in the kernel for the following primitives.

We develop a data tiling strategy to hide the data access latency behind the computations. First, when loading from global memory, we choose an appropriate tile size, i.e., 128×32 which is the ideal tile size to balance the computation capability and memory throughput on V100 GPUs. Below is our analysis: assuming the sizes of Tiles **A** and **B** are $M \cdot k$ and $N \cdot k$, to pipeline the loading of Tiles **A** and **B** with the computation, we hide the loading latency by arithmetic operations, $loadLatency = arithmeticLatency$. We denote that the latency of loading one FP32 value from global memory is $Latency_{global}$ and the latency of performing one multiply-accumulate operation is $Latency_{compute}$. We arrive at $loadLatency = (M \cdot k + N \cdot k) \cdot Latency_{global}$, and $arithmeticLatency = (M \cdot N \cdot k) \cdot Latency_{compute}$. This leads to $\frac{M+N}{M \cdot N} = \frac{Latency_{compute}}{Latency_{global}}$. On V100 GPU, we find $Latency_{global} \approx 400$, $Latency_{compute} \approx 4$. Without loss of generality, we let $M = N$. We hence arrive at $M = N \approx 200$. Since the size of M and N should be the power of two, we find $M = N = 128$ offer the best performance. Second, at the warp level, we let each warp load two blocks from Tiles **A** and **B** to compute four adjacent blocks in Tile **C** as shown in Figure 4. We derive the optimal block size that can hide the latency of accessing shared memory. Particularly, in each iteration, a thread loads 32 packed INT8 as the eight blocks colored on the right side of Figure 4. Then the 16 **DP4A** instructions cover the 18 cycles latency of loading for the next iteration.

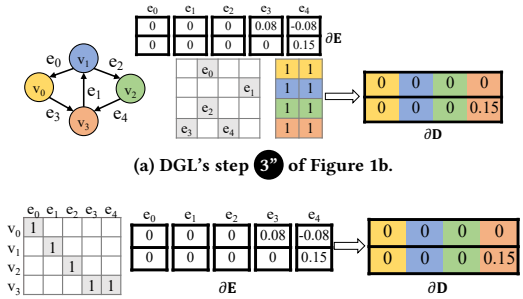
For computation, we carefully schedule the threads to improve the computation intensity. First, when loading and quantizing Tile **A** from global memory to shared memory, we transpose the Tile because the access is in column while Tile **A** is row-major in global memory. Second, to avoid bank conflict, a warp stores Block **A** in shared memory with a 16-byte offset in each column. Each warp works on 2×2 **C** blocks to reuse Block **A** and **B**. Third, we schedule the threads as shown in the left side of Figure 4, mapping 32 threads within a warp to block **C** to increase shared memory throughput. For example, threads 0,1,2,3 access the same address in block **A** so the loaded data can be broadcast to 4 threads.

Of note, there exist frameworks that can generate GEMM kernels with efficient tiling and scheduling, but none of them can be used to implement GEMM with *on-the-fly quantization*. On the one hand, template-based frameworks, such as AutoTVM [53] and Ansor [54], optimize kernels by enumerating the combinatorial choices of optimizations (e.g., tile layout, tile size, and parallelization). Searching the design space is time-consuming, and the generated kernels are not guaranteed to be optimal. On the other hand, on-the-fly quantization is not supported by the existing templates. Moreover, the kernel generated by these frameworks, e.g., triton compiler [55], uses too many registers, resulting in unsatisfied performance.

Incidence matrix-based adaptive SPMM. TANGO performs quantization in a separate kernel for SPMM. We use quantization to reduce the memory traffic for SPMM since quantization leads the node and edge feature matrices to a smaller size. Unlike GEMM, which performs sequential memory access for the input matrices, SPMM experiences random memory accesses. In this case, performing on-the-fly quantization would lead to random memory access for input matrices of 32-bit floating-point data type. Further, because of unpredictable access patterns, on-the-fly quantization could potentially lead to repeated quantization of the same data. Instead, a dedicated quantization kernel would read 32-bit input floating-point matrices sequentially once and write the 8-bit quantized matrices out, again, sequentially and once. Therefore, during SPMM, we perform random memory access to input matrices of 8-bit as opposed to 32-bit.

TANGO further introduces two SPMM variant optimizations, that is, incidence matrix-based SPMM and adaptive SPMM to improve the quantized training performance.

Incidence matrix-based SPMM accelerates the SPMM variant, i.e., step ❸ of Figure 1b. Particularly, this SPMM computes the gradients of node features by aggregating the incoming edge features for each node. Using node v_3 as an example, as shown in Figure 5a, because v_3 contributes to the edge features for e_3 and e_4, its gradient is the partial derivative of e_3 and e_4, that is, $\partial v_3 = \partial e_3 + \partial e_4$. However, because DGL uses the adjacency matrix format for the graph, shown in Figure 5a, we need three matrices for the SPMM, that is, this graph, ∂e, and the node features with all "1"s.



(a) DGL's step 3" of Figure 1b.

(b) TANGO's incidence matrix-based step 3" of Figure 1b.

Figure 5: Incidence matrix-based SPMM.

The drawback of this design is two-fold: First, one needs to allocate and access the all "1" node feature matrix which is redundant and expensive. Second, although the operation in Figure 5a can be formulated as an SPMM operation, it includes three inputs, which is not supported by the state-of-the-art cuSPARSE library.

In Figure 5b, TANGO formulates this 3" computation as an incidence matrix-based SPMM. Particularly, the incidence matrix is a $V \times E$ matrix where V and E are, respectively, the number of nodes and edges in the graph. Each row of the incidence matrix contains the incoming edges of each node by marking the associated entry as 1. This design allows us to compute step 3" by multiplying two matrices, i.e., the incidence matrix and edge feature. Because we only need two input matrices, TANGO can now adopt high-performance cuSPARSE SPMM kernels for step 3", which is significantly faster than DGL's three matrices-based SPMM.

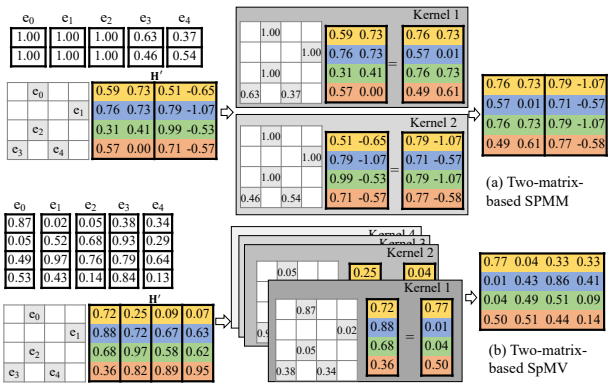


Figure 6: Transforming three-matrix-based SPMM, e.g., step 5 in Figure 1a into a collection of: (a) two-matrix-based SPMM and (b) two-matrix-based SpMV.

Kernel count-based adaptation. Chances are certain SPMM computations still involve three matrices, such as step 5 in Figure 1a. In Figure 6, we demonstrate how one could transform a three-matrix-based SPMM kernel into a collection of two-matrix-based SPMM kernels or, to an extreme, Sparse Matrix-Vector multiplication (SpMV) kernels. Once that transformation is completed, one could directly rely on cuSPARSE to perform each two-matrix-based SPMM or SpMV. Note, we prefer cuSPARSE over DGL primitives because our evaluation shows that a single cuSPARSE SPMM kernel is significantly faster than DGL's native two-matrix-based SPMM across various configurations.

However, the benefits brought by cuSPARSE kernels diminish along with the increment of the number of kernels, as kernel invocation cost soars when too many kernels are launched. In summary, neither DGL nor transformed cuSPARSE bests the other across all configurations. We hence adaptively leverage these two solutions to achieve the best performance of both worlds.

Figure 6a depicts the SPMM with both edge and node features as matrices. In this design, the first head of the node feature is scaled by the first element of the edge feature and similarly for the second head. We can use multiple optimized cuSPARSE SPMM kernels to replace the native kernel used in DGL. In this case, the two heads can be finished by two SPMM kernels. Figure 6b assumes we have four heads in step 5 of Figure 6. In this case, we arrive at four SpMV kernels for the original three-matrix-based SPMM.

SDDMM with on-the-fly dequantization. Similar to our SPMM design, we first perform sequential memory access to quantize the input matrices. During SDDMM, we perform random memory access on those quantized matrices of smaller sizes. Briefly, the existing SDDMM performs one round of random access on full precision matrices. In contrast, TANGO performs one round of sequential access on full precision matrices and one round of random access to the low precision matrices. This will lead TANGO to have a shorter turnaround time. Since SDDMM might perform addition or subtraction operations, one cannot directly compute the quantized values. This leads to our SDDMM with on-the-fly dequantization.

We use step 3 of Figure 1a to explain the reason. This SDDMM computes the edge feature by $\mathbf{E}[i][j] = \mathbf{S}[v_i] + \mathbf{D}[v_j]$. Assuming the scaling factor s for $\mathbf{S}$ and $\mathbf{D}$ are, respectively, $s_{\mathbf{S}}$ and $s_{\mathbf{D}}$. The addition in quantized format should be $\mathbf{S}[v_i] + \mathbf{D}[v_j] \approx s_{\mathbf{S}} \cdot \mathbf{S}_{Quant}[v_i] + s_{\mathbf{D}} \cdot \mathbf{D}_{Quant}[v_j]$. Because $s_{\mathbf{S}}$ and $s_{\mathbf{D}}$ are often not equal, one cannot directly add the quantized values $\mathbf{S}_{Quant}[v_i]$ and $\mathbf{D}_{Quant}[v_j]$. Therefore, TANGO loads the quantized data to enjoy reduced memory traffic, subsequently on-the-fly dequantize the loaded values for addition/subtraction computation.

If SDDMM performs multiplication and division, we can conduct SDDMM directly on the quantized value. Using step 5" of Figure 1b as an example, one needs to compute $\partial\boldsymbol{\alpha}[e_0] = \partial\mathbf{H}^{(l)}[v_0] \cdot \mathbf{H}'[v_1]$. In this case, assuming the scaling factor of $\partial\mathbf{H}^{(l)}[v_0]$ and $\mathbf{H}'[v_1]$ are s_0 and s_1. The computation can be approximated as $\partial\boldsymbol{\alpha}[e_0] \approx (s_0 \cdot \partial\mathbf{H}^{(l)}[v_0]) \cdot (s_1 \cdot \mathbf{H}'[v_1])$. We can further arrive at $\partial\boldsymbol{\alpha}[e_0] \approx (s_0 \cdot s_1) \cdot \partial\mathbf{H}^{(l)}[v_0] \cdot \mathbf{H}'[v_1]$. This allows TANGO to perform quantized multiplication directly. Division can also directly work on the quantized values.

Inter-primitive optimization. Noticing that the follow-up operators can reuse some quantized tensors, TANGO caches these quantized tensors to reduce the quantization overhead. In general, there exist two caching opportunities: (1) caching forward pass for backward, (2) caching prior operators for subsequent operators. TANGO develops a detection algorithm that runs on the computation graphs to automatically derive these reuse cases.

First, the backward computation can reuse the quantized tensors from the forward pass. For example, the GEMM of step 1 in Figure 1

has the forward computation $\mathbf{H}' = \mathbf{H}^{(l-1)} \cdot \mathbf{W}$. The corresponding backward step contains the gradient computation for weight, that is, $\partial\mathbf{W} = \mathbf{H}^{(l-1)} \cdot \partial\mathbf{H}'^{T}$, and the gradient computation of node features, i.e., $\partial\mathbf{H}^{(l-1)} = \partial\mathbf{H}' \cdot \mathbf{W}^{T}$, as shown in step 1' in Figure 1b. Clearly, the quantized matrices $\mathbf{H}^{(l-1)}$ and $\mathbf{W}$ are used in both forward and backward computations. We thus save the quantized input $\mathbf{H}^{(l-1)}$ and $\mathbf{W}$ during the forward pass for the backward pass to avoid repeated quantization. Second, when two operators share the same tensor as input, we can cache the quantized tensor from the former operator and use it for the latter. For example, in Figure 1b, the SPMM in step 5' and SDDMM in 5" both need the quantized $\partial\mathbf{H}^{(l)}$. This way, we cache the quantized input tensor. We also intentionally schedule the computation orders such that the cached tensors can be reused.

We derive the caching opportunity on the computation graph, i.e., Figure 1a as follows. The computation graph consists of tensors as nodes and operators as edges. For nodes with more than one out edge, we can quantize once for multiple operators. For example, the tensor $\partial\mathbf{H}^{(l)}_{\text{Quant}}$ in Figure 1b has two operators as out-edges, so we cache this tensor. Then we reverse the edges in the computation graph for the backward pass. In this backpropagation graph, we will check if the to-be-quantized tensors are already quantized in the forward graph in order to facilitate quantization sharing.

Quantization overhead vs. benefit analysis. While quantization helps reduce computation and data movement, it also brings overheads. Mainly, quantization introduces two types of overheads: parameter computing and data type casting. First, the parameter s in Equation 1 is derived by reducing the elements with the maximum absolute value. For a $N \times N$ matrix, the reduction needs N^2 operations to derive the absolute maximum values. Second, quantizing or dequantizing an element requires two operations: multiply and data type casting. Therefore, the quantization before the primitive performs $4N^2$ floating-point operations.

For GEMM with the input matrices at sizes of $M \times K$ and $K \times N$, we perform $4K(M+N)$ and $2MN$ operations for quantization and dequantization, respectively. For GEMM, we reduce the number of multiply-accumulate instructions from MNK to $\frac{MNK}{4}$, which is often significantly higher than the overheads. Sparse primitives quantize the node and edge feature matrices. We assume D as the size of features. Given a graph with N nodes and E edges, in SPMM, the node and edge features require $4D(N+E)$ operations for quantization. Later, only the resultant node features are dequantized with $2ND$ operations. Quantization in SDDMM performs $4ND$ operations for node features. Dequantizing resultant edge features needs $2ED$ operations. Regarding the benefits of sparse primitives, sparse primitives enjoy a better cache access pattern brought by quantization which reduces the sizes of the input matrices.

4 EXPERIMENTS

4.1 Experimental setup

Datasets. We conduct experiments on five graph datasets as shown in Table 1. The *obgn-arxiv* [56] and *Pubmed* [57] are citation graphs whose nodes represent papers and edges are the citations. The task is to predict the categories of papers. The *ogbn-products* [58] dataset depicts an product co-purchasing network, where nodes represent products sold in Amazon, and edges between two products indicate that they are purchased together. The task is to predict the category of a product. The *DBLP* dataset is a co-authorship network where nodes are authors and edges are co-authorship [59]. The *Amazon* dataset contains products as the nodes and edges represent co-purchase [60]. The tasks of *DBLP* and *Amazon* are to predict if a link exists between two nodes. We add the reverse edges for the directed graphs and self-connects edges to ensure the SPMM operation works for every node.

Dataset	ogbn-arxiv	ogbn-products	Pubmed	DBLP	Amazon
Nodes	169,343	2,449,029	19,717	317,080	410,236
Edges	1,166,243	61,859,140	88,651	1,049,866	3,356,824
Task	NC	NC	NC	LP	LP

Table 1: Graph datasets. NC denotes "node classification", and LP denotes "link prediction".

Models. We evaluate GCN [12] and GAT [22] from the example implementations of DGL and the models are trained with the same number of epochs and hyperparameter settings. Both models use the hidden size of 128 and two GNN layers; GAT has four attention heads. For node classification, the model generates the node embedding as the set probabilities of each category. For link prediction, we perform dot-product between two node embeddings as the score of edge existence. The training epochs for Pubmed, ogbn-arxiv, and ogbn-products are 30, 500, and 150.

Implementation details. *For ease of use, we integrate Tango in DGL. Therefore, all the models on DGL can enjoy the performance benefits brought from Tango without any changes.* We provide our optimized quantized CUDA kernels to replace the corresponding primitives in DGL. DGL employs the GEMM function from the cuBLAS library and sources its sparse primitives either directly from cuSPARSE [61] or through its own implementations. DGL's interface is designed in Python and its primitives are integrated as PyTorch functions. To ensure a fair comparison, the kernels of Tango are also invoked via PyTorch's auto-differential engine during training [62]. Furthermore, DGL supports multiple graph data structure formats, and Tango leverages DGL's heuristics to determine the most efficient format for its primitives.

Evaluation platforms. We use Python 3.6.10 and CUDA 11.7 on six V100S GPUs and Intel(R) Xeon(R) Gold 6244 @ 3.60GHz CPU. The model is trained with PyTorch 1.13.0 and DGL 0.8. We also have access to a single A100 GPU for a limited time. We use that to compare GEMM on INT8 tensor core vs FP16 tensor core.

4.2 Tango vs. state-of-the-art

We compare Tango with DGL [63] and EXACT [32]. DGL trains the model in full precision, and EXACT trains the model with quantized tensors. EXACT aims to save memory by quantizing the saved tensors, and it has no optimization in computation. We set EXACT to use 8-bit quantization. Of note, the GNN models are implemented through PyTorch, which experiences significant high-level language overheads when calling Tango primitives. As a result, we observe significantly smaller model-level speedups than the primitive-level comparisons (detailed in Section 4.3).

Training speed. We evaluate the training speedup of Tango on GCN and GAT models. We train each model 5 times and report the average elapsed time achieving the same accuracy as the baseline,

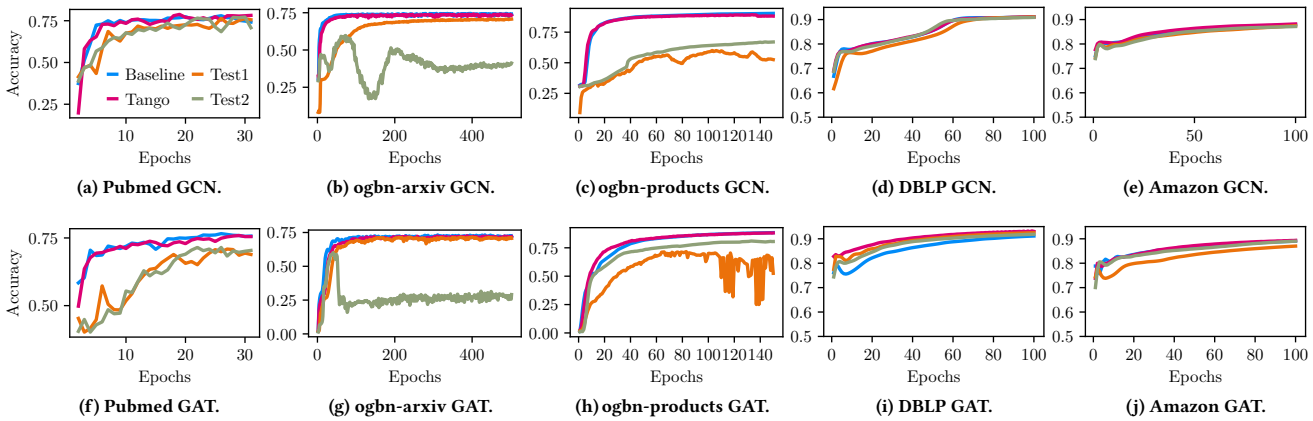


Figure 7: The convergence analysis of GCN and GAT with TANGO. Test1 denotes TANGO with quantized layer before Softmax. Test2 denotes TANGO using nearest rounding instead of stochastic rounding.

including the forward and backward computations. As shown in Figure 8, TANGO has 1.2× and 1.5× speedup on average on GCN and GAT models compared with DGL, respectively. The GAT model enjoys more benefits because it contains more quantized primitives than GCN. Further, larger graphs have more speedup for the GCN model, except the DBLP dataset, which has the smallest average degree among the five graphs. Overall, TANGO achieves an average speedup of 2.9× on GCN and 4.1× on GAT compared with EXACT. The key takeaway is that applying quantization without appropriate optimizations will lead to a significant slowdown (e.g., EXACT).

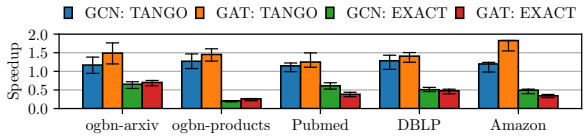


Figure 8: The speedup of training the GNN models with TANGO and EXACT compared with DGL.

Accuracy study. Figure 7 studies the accuracy impact of the techniques in Section 3.2 for GCN and GAT. In particular, we evaluate TANGO, TANGO with quantized layer before Softmax (**Test1**), and TANGO without stochastic rounding (**Test2**). For clarification, the training crashes without quantization-aware matrix multiplication and full precision weight update. The baseline models are trained in FP32 with the same number of epochs as the quantized training.

Overall, TANGO could achieve >99% accuracy of the full precision training with the same number of epochs. When quantizing the layer before Softmax (**Test1**), the models show noticeable accuracy loss except for the DBLP dataset, GCN model on Pubmed, and GAT model on ogbn-arxiv. The average relative accuracy drop is 9.7%. Despite the similar final accuracy, GAT on Pubmed and GCN on ogbn-arxiv converge slower than the baseline by 18 and 35 epochs, respectively. For quantization without stochastic rounding (**Test2**), we observe for GCN on Pubmed and both models on DBLP and Amazon, the quantization error changes the optimization direction in some epochs. Thus the models take more epochs to recover. Moreover, the models on ogbn-arxiv and ogbn-products suffer from significant accuracy drops. As shown in Figure 7a, although the model can achieve convergence, the training process shows more instability than that with stochastic rounding.

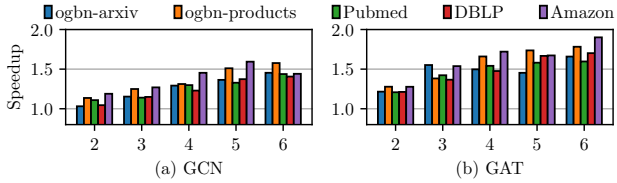


Figure 9: TANGO's impact on multi-GPU training. The X-axis is the number of GPUs.

Multi-GPU training. Figure 9 studies TANGO's impact on multi-GPU training. We directly adopt DGL's mini-batch multi-GPU training. That is, each GPU trains the model on a batch of sampled subgraphs per epoch. Then, the gradients of all GPUs are updated by an all-reduce operation. *We compare the training speed between full precision baseline and TANGO using the same number of GPUs.*

TANGO achieves speedup over the full precision baseline via transferring the quantized node features and gradients. Since we perform stochastic rounding-based quantization, this process will introduce nontrivial turnaround time. In TANGO, we overlap the feature quantization with the subgraph sampling. The overall trend is that more GPUs would enjoy higher speedup as the Peripheral Component Interconnect Express (PCI-E) congestion is better alleviated by our quantization. Particularly, the speedup increases from 1.1× to 1.5×, and 1.2× to 1.7× from two to six GPUs on GCN and GAT, respectively.

4.3 Turnaround time analysis

Caching the quantized tensors. Figure 10 shows the performance of caching the quantized tensor in forward for backward reuse. We test the GEMM primitive on different datasets. We test with two hidden sizes, $D = 128$ and $D = 256$. The result shows 1.7× and 1.6×

on average when $D = 128$ and $D = 256$, respectively. The saving is related to the data size; smaller graphs, such as Pubmed, enjoy more time savings.

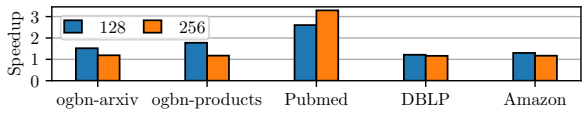


Figure 10: The speedup of caching the quantized tensors.

GEMM. Figure 11a shows the speedup of our quantized GEMM over cuBLAS GEMM with the hidden size $D = 256$ and $D = 512$. Of note, we include the quantization cost in Tango GEMM time. Our quantized GEMM primitive has 2.2× and 2.5× speedup on average when $D = 256$ and $D = 512$, respectively. The trend also suggests that quantization offers more speedup on the GEMM operator when the hidden size increases. In addition, we also compare our quantized INT8 GEMM with FP16 GEMM on A100 Tensor Core GPUs. Figure 11b shows that our quantized GEMM primitives have 1.9× for $D = 256$ and 1.8× for $D = 512$. Since both baseline and Tango use Tensor Core, the speedup over baseline is smaller than using CUDA core because the performance difference of computing in INT8 and FP16 is 2× on A100 tensor cores. Further, we observe a speedup drop for a bigger D because our quantization needs to scan through a bigger tensor to extract the scaling factor s.

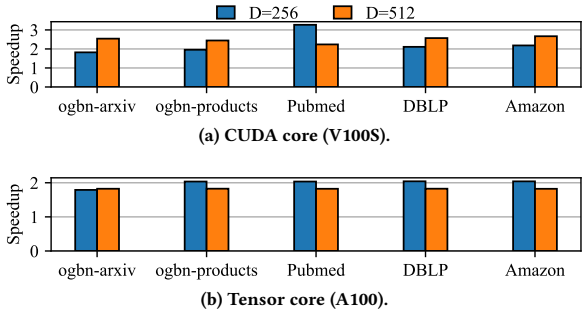


Figure 11: Tango GEMM vs cuBLAS GEMM.

Figure 12 shows the profiling results of quantized GEMM. We profile the ratio of achieved computation throughput (operation/s), memory throughput (GB/s), Instruction Per Cycle (IPC), and the number of instructions compared with cuBLAS FP32 GEMM [64]. The average computation and memory throughput ratios are 2.1× and 2.2×, respectively. Memory throughput is higher because our quantized GEMM writes the quantized matrix out. However, since GEMM is computation-intensive, our increased computation throughput dominates the performance impacts. Our further investigation into IPC and # of instructions in Figure 12b explains how Tango doubles the computation throughput. Our average IPC is ~70% of the baseline, with the instruction number reduced to ~31%. Together, we can roughly double the throughput of the baseline.

SPMM. Figure 13a shows the performance of using incidence matrix SPMM for edge aggregation compared with DGL SPMM kernels. We set the edge features size ranging from 4 to 20. All

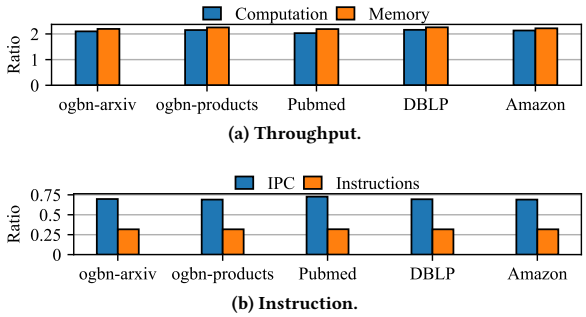


Figure 12: The hardware profiling of quantized GEMM.

Dataset	ogbn-arxiv	ogbn-products	Pubmed	DBLP	Amazon
Ours (GB/s)	344.06	491.72	353.38	331.57	342.93
Baseline (GB/s)	41.26	244.22	131.89	297.67	105.88

Table 2: The achieved memory throughput using incidence-based SPMM and DGL baseline.

dataset has an average 2.1× speedup. The ogbn-arxiv dataset has the best speedup of 5.5× on average because of the poor performance of its baseline kernel. That is, the randomness of the incidence matrix is much lower than that of the adjacency matrix from the baseline. Table 2 shows the achieved memory throughput using our incidence-based SPMM and baseline when the feature size is 16. The irregular access for the baseline on the ogbn-arxiv dataset leads to low memory throughput. Using the incidence matrix alleviates the irregularity because the edges incidents to a node are stored adjacent in memory.

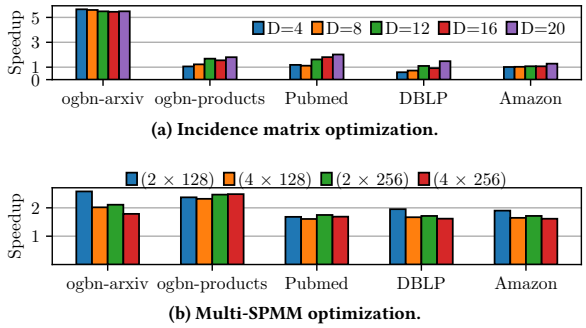


Figure 13: Tango SPMM optimizations.

Figure 13b shows the performance of using multiple SPMM's with a small edge feature dimension in a multi-head graph attention operation compared with DGL SPMM kernels. We set the node feature dimension as $(H \times D)$, and the edge features dimension as $(H \times 1)$, where H represents the number of heads and D represents the hidden size of each head. Ours has 2.1×, 1.9×, 2.0×, and 1.8× speedup over DGL's primitive on average respectively for (2×128), (4×128), (2×256), and (4×256). Increasing the number of heads leads to a smaller speedup when the hidden size is fixed because of the increased overhead of more kernel launches. The hidden size has little impact on speedup with the same head number.

Figure 14 shows the performance of using multiple cuSPARSE SpMV with a large edge feature dimension on ogbn-arxiv graph.

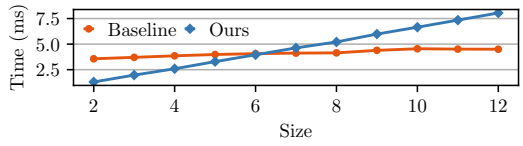


Figure 14: The performance of using multiple cuSPARSE SPMV with high edge feature dimension.

We test the feature size ranging from 2 to 12. When the size is smaller than 6, ours has a 1.6× speedup on average over DGL's implementation. The result shows the increased turnaround time as the number of kernels increases.

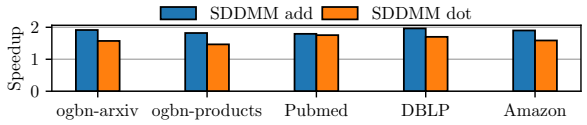
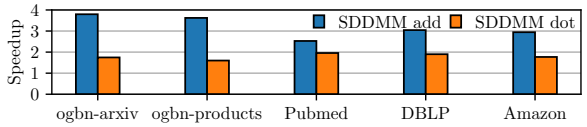
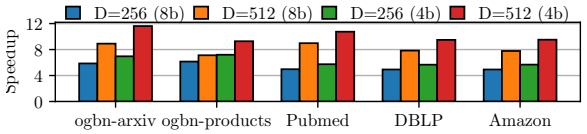


Figure 15: The performance of SDDMM operators.

SDDMM. Figure 15 shows the performance of quantized SDDMM compared with DGL SDDMM kernels. We evaluate two SDDMM variants, including the row-wise dot-product (step 5" in Figure 1b) and element-wise addition (step 3 in Figure 1a), denoted as *SDDMM dot* and *SDDMM add*. The node features are matrices with the size of (4, 64). Our SDDMM add and SDDMM dot achieves 1.9× and 1.6× speedups over DGL, respectively.



(a) INT4 SDDMM.



(b) INT4 GEMM and INT8 GEMM.

Figure 16: The turnaround time impacts by varying # of bits.

4.4 Speed impact for # of quantization bits

Because neither cuSPARSE nor DGL provides SPMM kernels of INT4, this section only studies INT4 GEMM and SDDMM which are implemented by TANGO. Figure 16a shows the SDDMM performance using INT4 compared with full precision DGL primitives. The addition and dot-product kernels achieve, on average, 3.3× and 1.8×, respectively. Dense graphs like *ogbn-arxiv* and *ogbn-products* enjoy more benefits from reduced memory traffic because the node embeddings are more likely to be reused by cache hit.

Figure 16b shows the GEMM performance using INT8, and INT4 compared with cuBLAS. Note that we run the tests on an A100 GPU with INT4 hardware support. Using INT8 and INT4 leads to 5.4× and 6.2× average speedup when hidden size $D = 256$. For $D = 512$, the average speedup is 8.1× and 10.1×, respectively. Using fewer bits shows marginal improvement because the sub-byte access under-utilizes the shared memory bandwidth.

5 RELATED WORK

Recent years have seen a surge of efforts on GNN [63, 65–78]. For a comprehensive study about the history and advancements in quantization for DNNs and GNNs, we refer the readers to two surveys [79, 80]. In addition to the related work in Section 1, this section further discusses GNN primitives and inference.

GNN operator optimization projects often focus on improving the SPMM and SDDMM kernels in GNN. GE-SpMM [81] and DA-SpMM [82] propose optimizations for implementing SPMM on GPU for GNN workloads. QGTC [83] accelerates quantized GNN operations using Tensor Cores on GPU by representing the adjacency matrix as a 1-bit sparse matrix and quantizing node features in any bits, which can be computed using the 1-bit computation function on Ampere Tensor Cores. GE-SpMM, QGTC, and DA-SpMM do not support models with multi-edge features like GAT, while TANGO revamps SPMM to support such models. In addition, TANGO also supports quantized GEMM and SDDMM. FeatGraph [84] uses tensor compilers, providing a flexible programming interface to generate SPMM and SDDMM for various GNN operations. FeatGraph aims to exploit parallelism for customized operations between features. However, FeatGraph does not support quantization.

GNN inference quantization has received significant attentions recently [35, 83, 85–87]. Unfortunately, none of these can achieve a shorter turnaround time for *training* than not quantized GNN models. Particularly, [87] quantizes the GNN into binary with knowledge distillation to reduce accuracy loss. Since knowledge distillation needs to train a teacher model and a student model, the training time increases. SGQuant [86] is a quantization scheme aiming to reduce memory consumption. It assigns different bits to embeddings and attention tensors on different levels, but the mismatch of datatype incurs extra conversion overhead. In contrast, TANGO introduces a variety of framework and primitive-level system optimizations, leading to a shorter turnaround time during quantized GNN training.

6 CONCLUSION

TANGO identifies both the challenges and opportunities brought by quantization to GNN training. Particularly, TANGO makes the following three major contributions. First, TANGO introduces various lightweight rules to maintain the accuracy for quantized GNN training. Second, we design and implement quantization-aware primitives and inter-primitive optimizations to reduce the turnaround time for quantized GNN training. Third, we integrate TANGO into DGL and evaluate it across a variety of GNN models and datasets to demonstrate the superior performance of TANGO.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their helpful suggestions. This work was in part supported by the NSF CRII Award No. 2331536, CAREER Award No. 2326141, and NSF 2212370, 2319880, 2328948, 2319975, 2331301 and Semiconductor Research Corporation (SRC) Artificial Intelligence Hardware program. Any opinions, findings conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Madhav Marathe and Anil Kumar S Vullikanti. Computational Epidemiology. *Communications of the ACM*, 56(7):88–96, 2013.

[2] Guo Zhang, Hao He, and Dina Katabi. Circuit-GNN: Graph neural networks for distributed circuit design. In *International Conference on Machine Learning*, pages 7364–7373. PMLR, 2019.

[3] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[4] Anil Gaihre, Da Zheng, Scott Weitze, Lingda Li, Shuaiwen Leon Song, Caiwen Ding, Xiaoye S Li, and Hang Liu. Dr. Top-k: Delegate-Centric Top-k on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[5] Kaustav Banerjee, Shukri J Souri, Pawan Kapur, and Krishna C Saraswat. 3-D ICs: A Novel Chip Design for Improving Deep-submicrometer Interconnect Performance and Systems-on-Chip Integration. *Proceedings of the IEEE*, 89(5):602–633, 2001.

[6] Qihang Chen, Boyu Tian, and Mingyu Gao. FINGERS: Exploiting Fine-Grained Parallelism in Graph Mining Accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 43–55, New York, NY, USA, 2022.

[7] Shiyang Chen, Shaoyi Huang, Santosh Pandey, Bingbing Li, Guang R Gao, Long Zheng, Caiwen Ding, and Hang Liu. ET: Re-thinking Self-Attention for Transformer Models on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–18, 2021.

[8] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. XBFS: eXploring runtime optimizations for breadth-first search on GPUs. In *Proceedings of the 28th International symposium on high-performance parallel and distributed computing*, pages 121–131, 2019.

[9] Hang Liu and H Howie Huang. {SIMD-X}: Programming and processing of graph algorithms on {GPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 411–428, 2019.

[10] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 544–557, 2018.

[11] Remi Lam, Alvaro Sanchez-Gonzalez, Matthew Willson, Peter Wirnsberger, Meire Fortunato, Alexander Pritzel, Suman Ravuri, Timo Ewalds, Ferran Alet, Zach Eaton-Rosen, et al. GraphCast: Learning skillful medium-range global weather forecasting. *arXiv preprint arXiv:2212.12794*, 2022.

[12] Thomas N Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*, 2016.

[13] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in Neural Information Processing System*, 30, 2017.

[14] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[15] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. Dgl-ke: Training knowledge graph embeddings at scale. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 739–748, 2020.

[16] Nidhi Rastogi, Sharmishtha Dutta, Christian Ryan, Mohammad Zaki, Alex Gittens, and Charu C. Aggarwal. Information Prediction using Knowledge Graphs for Contextual Malware Threat Intelligence. *CoRR*, abs/2102.05571, 2021.

[17] Qingyu Xu, Feng Zhang, Mingde Zhang, Jidong Zhai, Bingsheng He, Cheng Yang, Shuhao Zhang, Jiazao Lin, Haidi Liu, and Xiaoyong Du. Payment behavior prediction on shared parking lots with TR-GCN. *The VLDB Journal*, pages 1–24, 2022.

[18] Seung-Hwan Lim, Junghoon Chae, Guojing Cong, Drahomira Herrmannova, Robert M Patton, Ramakrishnan Kannan, and Thomas E Potok. Visual Understanding of COVID-19 Knowledge Graph for Predictive Analysis. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 4381–4386. IEEE, 2021.

[19] Yifei Wang, Shiyang Chen, Guobin Chen, Ethan Shurberg, Hang Liu, and Pengyu Hong. Motif-Based Graph Representation Learning with Application to Chemical Molecules. *Informatics*, 10(1), 2023.

[20] Jaeyeon Won, Jeyeon Si, Sam Son, Tae Jun Ham, and Jae W Lee. ULPPACK: Fast Sub-8-bit Matrix Multiply on Commodity SIMD Hardware. *Proceedings of Machine Learning and Systems*, 4:52–63, 2022.

[21] Minjia Zhang, Wenhan Wang, and Yuxiong He. GraSP: Optimizing Graph-Based Nearest Neighbor Search with Subgraph Sampling and Pruning. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, WSDM '22, page 1395–1405, New York, NY, USA, 2022. ACM.

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

[23] Nvidia. Tensor Cores. Retrieved from https://developer.nvidia.com/tensor-cores. Accessed: 2022, Nov 26.

[24] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training Deep Neural Networks with 8-bit Floating Point Numbers. *Advances in Neural Information Processing Systems*, 31, 2018.

[25] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. *Advances in Neural Information Processing Systems*, 30, 2017.

[26] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, et al. Mixed Precision Training of Convolutional Neural Networks using Integer Operations. In *International Conference on Learning Representations*, 2018.

[27] Charbel Sakr and Naresh Shanbhag. Per-Tensor Fixed-Point Quantization of the Back-Propagation Algorithm. In *International Conference on Learning Representations*, 2018.

[28] Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Chris De Sa. SWALP: Stochastic Weight Averaging in Low Precision Training. In *International Conference on Machine Learning*, pages 7015–7024. PMLR, 2019.

[29] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. Towards Unified INT8 Training for Convolutional Neural Network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1969–1979, 2020.

[30] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael Mahoney, and Joseph Gonzalez. ActNN: Reducing Training Memory Footprint via 2-Bit Activation Compressed Training. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 1803–1813. PMLR, 18–24 Jul 2021.

[31] Huiyuan Chen, Xiaoting Li, Kaixiong Zhou, Xia Hu, Chin-Chia Michael Yeh, Yan Zheng, and Hao Yang. TinyKG: Memory-Efficient Training Framework for Knowledge Graph Neural Recommender Systems. In *Proceedings of the 16th ACM Conference on Recommender Systems*, pages 257–267, 2022.

[32] Zirui Liu, Kaixiong Zhou, Fan Yang, Li Li, Rui Chen, and Xia Hu. EXACT: Scalable graph neural networks training via extreme activation compression. In *International Conference on Learning Representations*, 2022.

[33] Xiaoxuan Liu, Lianmin Zheng, Dequan Wang, Yukuo Cen, Weize Chen, Xu Han, Jianfei Chen, Zhiyuan Liu, Jie Tang, Joey Gonzalez, et al. GACT: Activation Compressed Training for Generic Network Architectures. *International Conference on Machine Learning*, 2022.

[34] R David Evans and Tor Aamodt. AC-GC: Lossy activation compression with guaranteed convergence. *Advances in Neural Information Processing Systems*, 34:27434–27448, 2021.

[35] Shyam Anil Tailor, Javier Fernandez-Marques, and Nicholas Donald Lane. Degree-Quant: Quantization-Aware Training for Graph Neural Networks. In *International Conference on Learning Representations*, 2021.

[36] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8BERT: Quantized 8bit BERT. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 36–39. IEEE, 2019.

[37] Aishwarya Bhandare, Vamsi Sripathi, Deepthi Karkada, Vivek Menon, Sun Choi, Kushal Datta, and Vikram Saletore. Efficient 8-Bit Quantization of Transformer Neural Machine Language Translation Model. *arXiv preprint arXiv:1906.00532*, 2019.

[38] Qing Jin, Linjie Yang, and Zhenyu Liao. Adabits: Neural Network Quantization with Adaptive Bit-Widths. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2146–2156, 2020.

[39] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. HAWQ: Hessian AWare Quantization of Neural Networks with Mixed-Precision. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 293–302, 2019.

[40] Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V Arthur, and Dharmendra S Modha. Backpropagation for Energy-Efficient Neuromorphic Computing. *Advances in Neural Information Processing Systems*, 28, 2015.

[41] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in Neural Information Processing Systems*, 28, 2015.

[42] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[43] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling Relational Data with Graph Convolutional Networks. In *The Semantic Web*, pages 593–607, Cham, 2018. Springer International

Publishing.
[44] Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and Philip S. Yu. A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering*, 29(1):17–37, 2017.
[45] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous Graph Transformer. In *Proceedings of The Web Conference 2020*, pages 2704–2710, 2020.
[46] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. In *International Conference on Machine Learning*, pages 1737–1746. PMLR, 2015.
[47] CUDA Nvidia. cuRAND library programming guide. *NVIDIA Corporation. edit*, 1, 2022.
[48] David Blackman and Sebastiano Vigna. Scrambled Linear Pseudorandom Number Generators. *ACM Transactions on Mathematical Software (TOMS)*, 47(4):1–32, 2021.
[49] Ralph Abboud, İsmail İlkan Ceylan, Martin Grohe, and Thomas Lukasiewicz. The Surprising Power of Graph Neural Networks with Random Node Initialization. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 2112–2118, 8 2021.
[50] Hyunsun Park, Jun Haeng Lee, Youngmin Oh, Sangwon Ha, and Seungwon Lee. Training Deep Neural Network in Limited Precision. *arXiv preprint arXiv:1810.05486*, 2018.
[51] Xi Chen, Xiaolin Hu, Hucheng Zhou, and Ningyi Xu. FxpNet: Training a deep convolutional neural network in fixed-point representation. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2494–2501. IEEE, 2017.
[52] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv preprint arXiv:1606.06160*, 2016.
[53] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.
[54] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating High-performance Tensor Programs for Deep Learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 863–879, 2020.
[55] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.
[56] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. Microsoft Academic Graph: When experts are not enough. *Quantitative Science Studies*, 1(1):396–413, 2020.
[57] Galileo Namata, Ben London, Lise Getoor, Bert Huang, and UMD EDU. Query-driven Active Surveying for Collective Classification. In *10th International Workshop on Mining and Learning with Graphs*, volume 8, page 1, 2012.
[58] K. Bhatia, K. Dahiya, H. Jain, P. Kar, A. Mittal, Y. Prabhu, and M. Varma. The Extreme Classification Repository: Multi-label Datasets and code, 2016.
[59] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities based on Ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.
[60] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. The Dynamics of Viral Marketing. *ACM Transactions on the Web (TWEB)*, 1(1):5–es, 2007.
[61] M Naumov, LS Chien, P Vandermersch, and U Kapasi. Cusparse library. In *GPU Technology Conference (GTC)*, 2010.
[62] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.
[63] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315*, 2019.
[64] CUDA Nvidia. cuBLAS library programming guide. *NVIDIA Corporation. edit*, 1, 2007.
[65] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. ACM.
[66] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. C-SAW: A framework for graph sampling and random walk on GPUs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.
[67] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. GNNLab: A Factored System for Sample-Based GNN Training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 417–434, New York, NY, USA, 2022. ACM.
[68] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. TileSpGEMM: A Tiled Algorithm for Parallel Sparse General Matrix-Matrix Multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 90–106, New York, NY, USA, 2022. ACM.
[69] Jesun Sahariar Firoz, Ang Li, Jiajia Li, and Kevin Barker. On the Feasibility of Using Reduced-Precision Tensor Core Operations for Graph Analytics. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2020.
[70] Jou-An Chen, Hsin-Hsuan Sung, Xipeng Shen, Nathan Tallent, Kevin Barker, and Ang Li. Bit-GraphBLAS: Bit-Level Optimizations of Matrix-Centric Graph Processing on GPU, 2022.
[71] Zhaoxia Deng, Jongsoo Park, Ping Tak Peter Tang, Haixin Liu, Jie Yang, Hector Yuen, Jianyu Huang, Daya Khudia, Xiaohan Wei, Ellie Wen, et al. Low-Precision Hardware Architectures Meet Recommendation Model Inference at Scale. *IEEE Micro*, 41(5):93–100, 2021.
[72] Kazem Cheshmi, Michelle Mills Strout, and Maryam Mehri Dehnavi. Optimizing Sparse Computations Jointly. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 459–460, 2022.
[73] Matthias Fey, Jan E Lenssen, Frank Weichert, and Jure Leskovec. GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings. In *International conference on machine learning*, pages 3294–3304. PMLR, 2021.
[74] Seher Acer, Ariful Azad, Erik G Boman, Aydın Buluç, Karen D Devine, SM Ferdous, Nitin Gawande, Sayan Ghosh, Mahantesh Halappanavar, Ananth Kalyanaraman, et al. EXAGRAPH: Graph and combinatorial methods for enabling exascale applications. *The International Journal of High Performance Computing Applications*, 35(6):553–571, 2021.
[75] Jiayu Li, Fugang Wang, Takuya Araki, and Judy Qiu. Generalized Sparse Matrix-Matrix Multiplication for Vector Engines and Graph Applications. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 33–42. IEEE, 2019.
[76] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference*, pages 443–458, Renton, WA, July 2019. USENIX Association.
[77] Youhui Bai, Cheng Li, Zhiqi Lin, Yufei Wu, Youshan Miao, Yunxin Liu, and Yinlong Xu. Efficient Data Loader for Fast Sampling-Based GNN Training on Large Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 32(10):2541–2556, 2021.
[78] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. DistDGL: Distributed Graph Neural Network Training for Billion-scale Graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44. IEEE, 2020.
[79] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A Survey of Quantization Methods for Efficient Neural Network Inference. *arXiv preprint arXiv:2103.13630*, 2021.
[80] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A White Paper on Neural Network Quantization. *arXiv preprint arXiv:2106.08295*, 2021.
[81] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2020.
[82] Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. Heuristic Adaptability to Input Dynamics for SpMM on GPUs . In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 595–600, 2022.
[83] Yuke Wang, Boyuan Feng, and Yufei Ding. QGTC: Accelerating Quantized Graph Neural Networks via GPU Tensor Core. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 107–119, New York, NY, USA, 2022. ACM.
[84] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC. IEEE, 2020.
[85] Mucong Ding, Kezhi Kong, Jingling Li, Chen Zhu, John Dickerson, Furong Huang, and Tom Goldstein. VQ-GNN: A universal framework to scale up graph neural networks using vector quantization. In *Advances in Neural Information Processing Systems*, volume 34, pages 6733–6746. Curran Associates, Inc., 2021.
[86] Boyuan Feng, Yuke Wang, Xu Li, Shu Yang, Xueqiao Peng, and Yufei Ding. SGQuant: Squeezing the Last Bit on Graph Neural Networks with Specialized Quantization. In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1044–1052, 2020.
[87] Mehdi Bahri, Gaétan Bahl, and Stefanos Zafeiriou. Binary Graph Neural Networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9492–9501, 2021.
[88] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. *arXiv preprint arXiv:1308.3432*, 2013.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT IDENTIFICATION

The main contributions of this paper focus on developing and implementing Tango, a novel approach that addresses the challenges and opportunities associated with quantized Graph Neural Network (GNN) training on GPUs. Tango offers three primary contributions: lightweight rules for maintaining accuracy in quantized GNN training, quantization-aware primitives and inter-primitive optimizations for accelerating the training process, and seamless integration with the Deep Graph Library (DGL) system. The computational artifacts associated with these contributions play a crucial role in demonstrating the effectiveness of Tango in improving GNN training performance.

In terms of software architecture, Tango is designed with optimized quantized CUDA kernels that replace the corresponding primitives in DGL. These kernels, as essential artifacts, enable Tango to accelerate GNN training while maintaining accuracy efficiently. By integrating Tango with DGL, users can easily take advantage of Tango's performance benefits without modifying their existing DGL models. The PyTorch auto-differential engine calls Tango's kernels during training, ensuring seamless interaction with the widely used deep learning framework.

The computational artifacts presented in this paper contribute significantly to the reproducibility of the experiments conducted in the article. By providing a detailed description of the experimental settings, including datasets, models, implementation details, and evaluation platforms, the authors enable other researchers to reproduce the experiments and verify the performance improvements demonstrated by Tango. The availability of Tango's implementation, integrated with the DGL system and using optimized quantized CUDA kernels, ensures that researchers can accurately replicate the experiments and validate the effectiveness of Tango in addressing quantization challenges and opportunities for GNN training on GPUs.

(i) *Optimized Quantized CUDA Kernels*: These kernels serve as a key computational artifact in Tango's design, allowing for accelerated GNN training while maintaining the required accuracy. They replace the corresponding primitives in DGL, thus offering seamless integration and improved performance in GNN training.

(ii) *Tango's Integration with DGL*: This artifact is crucial for user experience and accessibility. By integrating Tango with the DGL system, the performance benefits of Tango can be enjoyed by users without modifying their existing DGL models. This seamless integration allows researchers to reproduce the experiments and verify Tango's effectiveness.

(iii) *Datasets*: The datasets used in the experiments (obgn-arxiv, Pubmed, ogbn-products, DBLP, and Amazon) serve as essential artifacts, as they provide a comprehensive evaluation of Tango's performance across various graph-related tasks. By offering detailed information on these datasets, the authors ensure the reproducibility of the experiments and validation of Tango's performance improvements.

REPRODUCIBILITY OF EXPERIMENTS

(i) *Experiment Workflow*: The experiment workflow includes the end-to-end training evaluation and the evaluation of techniques. For the end-to-end training evaluation, the computational artifacts will be used to implement the training pipeline based on the example scripts in DGL, encompassing dataset preparation, model creation, training, and evaluation. The techniques are evaluated based on the exposed API of Tango artifacts, utilizing the datasets as input to report the performance.

(ii) *Execution Time*: Executing the end-to-end training evaluation takes approximately 2 hours for all datasets. The evaluation of techniques, which includes running time, hardware utilization, and accuracy of each training epoch, also takes an estimated 2 hours to complete.

(iii) *Expected Results*: The expected results consist of the execution time and the achieved model accuracy. In particular, the evaluation of the artifact will generate the profiling results of each experiment, which includes the running time, hardware utilization, and accuracy of each training epoch. The results will be in the form of plain text in the stdout.

(iv) *Relating Expected Results to Article Results*: The results from the experiment workflow are designed to align with the results presented in the article. The expected results will be converted to figures, as shown in the paper, using Matplotlib. The paper uses speedup to present the running time performance and the training progress of the accuracy curve to give convergence. Providing the expected results in the same format as the ones in the article facilitates the reproducibility of experiments, making it easier for other researchers to understand and verify the results.

ARTIFACT DEPENDENCIES REQUIREMENTS

(i) *Hardware*: V100S GPUs and Intel(R) Xeon(R) Gold 6244 @ 3.60GHz CPU are used in the paper. Besides, GPU later than Pascal are also supported by changing the compilation options.

(ii) *Operating system*: Ubuntu 20.04

(iii) *Software libraries*: Python 3.6.10, CUDA 11.7, DGL 0.8, PyTorch 1.13.0, and ogb 1.3.4

(iii) *Input datasets*: We choose the large graph datasets for GNN training. ogbn-arxiv, Pubmed, ogbn-products are chosen for node classification task. DBLP and Amazon are chosen for link prediction task. All datasets are publicly available. In particular, ogbn-arxiv and ogbn-products are avaliable in Open Graph Benchmark. DBLP and Amazon are downloaded from SNAP network database. We use DGL's built-in Pubmed datasets.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

Tango are integrated to PyTorch as extension with JIT compilation. Specifically, the first time running the Python script importing Tango will compile the code. The estimated compile time is 5 min for the first time. The compiled binary is cached in OS-specific PyTorch extension caching directory.

the datasets need to be preprocessed after downloading. Tango provides the preprocessing script. For convenience, the preprocessing is at runtime before the training. ogbn-arxiv, ogbn-products, and Pubmed are automatically downloaded when importing. DBLP and Amazon needs to be downloaded from SNAP website,