Eliminating Data Processing Bottlenecks in GNN Training over Large Graphs via Two-level Feature Compression

Yuxin Ma
Ping Gong
USTC
leeadama.gpzlx1@mail.ustc.edu.cn

Tianming Wu
Jiawei Yi
Chengru Yang
USTC
wutianming.jiaweiyi.hibiki@mail.ustc.edu.cn

Cheng Li
USTC
Institute of Artificial Intelligence, Hefei Comprehensive National Science Center
chengli7@ustc.edu.cn

Qirong Peng
Guiming Xie
OPPO
pengjirong.xieguiming@oppo.com

Yongcheng Bao
Haifeng Liu
OPPO
baoyongcheng.blade@oppo.com

Yinlong Xu
USTC
ylxu@ustc.edu.cn

ABSTRACT

Training GNNs over large graphs faces a severe data processing bottleneck, involving both sampling and feature loading. To tackle this issue, we introduce F^{2}CGT, a fast GNN training system incorporating feature compression. To avoid potential accuracy degradation, we propose a two-level, hybrid feature compression approach that applies different compression methods to various graph nodes. This differentiated choice strikes a balance between rounding errors, compression ratios, model accuracy loss, and preprocessing costs. Our theoretical analysis proves that this approach offers convergence and comparable model accuracy as the conventional training without feature compression. Additionally, we also co-design the on-GPU cache sub-system with compression-enabled training within F^{2}CGT. The new cache sub-system, driven by a cost model, runs new cache policies to carefully choose graph nodes with high access frequencies, and well partitions the spare GPU memory for various types of graph data, for improving cache hit rates. Finally, extensive evaluation of F^{2}CGT on two popular GNN models and four datasets, including three large public datasets, demonstrates that F^{2}CGT achieves a compression ratio of up to 128 and provides GNN training speedups of 1.23-2.56× and 3.58-71.46× for single-machine and distributed training, respectively, with up to 32 GPUs and marginal accuracy loss.

PVLDB Reference Format:
Yuxin Ma, Ping Gong, Tianming Wu, Jiawei Yi, Chengru Yang, Cheng Li, Qirong Peng, Guiming Xie, Yongcheng Bao, Haifeng Liu, and Yinlong Xu. Eliminating Data Processing Bottlenecks in GNN Training over Large Graphs via Two-level Feature Compression. PVLDB, 17(11): 2854 - 2866, 2024.
doi:10.14778/3681954.3681968

Yuxin Ma and Ping Gong equally contributed to this work.
Cheng Li is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681968

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/gpzlx1/F2CGT.

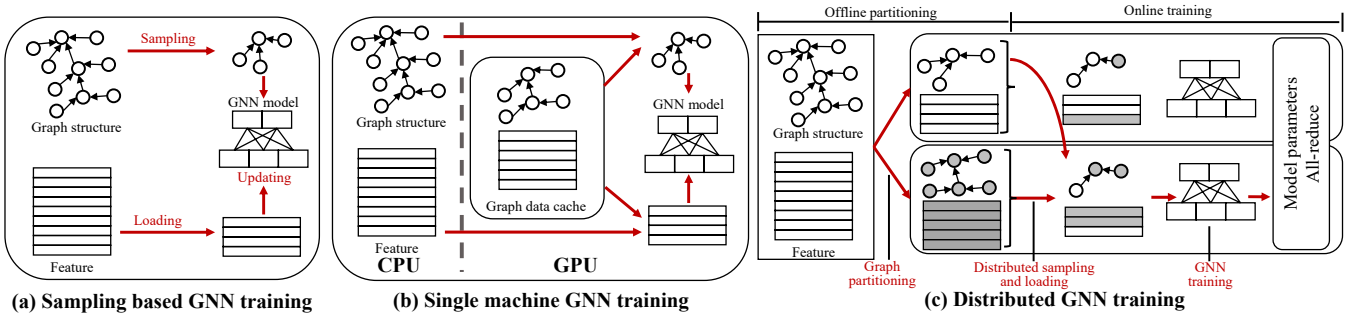
1 INTRODUCTION

Graph Neural Networks (GNNs) have extended machine learning methods for modeling graph-structured data. Popular GNN models such as GraphSAGE[15], GCN [21], and GAT [39] have recently achieved state-of-the-art (SOTA) performance in a broad range of fields, ranging from social networks [22], recommendation systems [11], to bioinformatics [31].

It is a norm to train GNN models by leveraging CPU-GPU cooperation, where GNN-related computations occur on the GPU, utilizing mini-batches of sampled subgraph data provided by the CPU. This collaboration between the CPU and GPU is essential because GNN computations can benefit significantly from the extensive parallelism offered by the GPU, while the limited GPU memory can hardly accommodate entire real-world graphs. However, this cooperation results in a performance issue as the expensive data processing cost between CPU and GPU limits the end-to-end training speed, which is mostly contributed by sampling subgraphs and loading feature vectors of sampled nodes to GPU per mini-batch basis, in single-machine or distributed training scenarios.

Some prior works [8, 24, 44, 46] employ on-GPU cache to speed up data loading, but this approach falls short for large graphs due to limited cache size and the randomness of data access. Some works [7, 10] apply model-specific optimizations to reduce data loading workload at the cost of accuracy degradation or model computational complexity.

In this paper, standing from a different perspective, we propose a general approach to minimize graph data volume as possible to eliminate the data processing bottleneck through feature compression. Through empirical insights and theoretical analysis, we demonstrate that feature compression can be applied to GNN training due to the mitigating effect of aggregation steps on compression errors. We propose two basic compression methods tailored to the characteristics of graph features and GNN training patterns and introduce a two-level compression approach to achieve higher compression ratios while maintaining model accuracy. Finally, we present F^{2}CGT,



(a) Sampling based GNN training (b) Single machine GNN training (c) Distributed GNN training

Figure 1: The sampled-based mini-batch GNN training workflow

a novel GNN training system that significantly enhances training efficiency by leveraging the advantages of feature compression. By introducing operator fusion between decompression and GNN aggregation steps, we substantially reduce model computation and GPU memory overhead. The saved memory can be allocated for caching to reduce PCIe and network communication. We also made significant modifications to the GPU cache design to improve cache efficiency. With a cost model, the new cache sub-system takes a few factors, such as spare GPU memory, compression ratios, and benefits of caching graph structure and feature vectors, into account for allocating space for both data. We also build a sampling-based parallel compression procedure to accelerate the preprocessing. With the support of feature compression, F^{2}CGT is able to reduce inter-machine communication by an order of magnitude or more. Additionally, we can now handle large graphs that previously required partitioning before distributed training. By storing a complete replica of the graph on each machine, we completely eliminate the need for remote feature retrieval.

Our extensive evaluation uses two popular GNN models and four datasets, where three are large graphs. Results demonstrate that F^{2}CGT achieves a compression ratio of up to 128, and provides GNN training speedup of 1.23-2.56× for single-machine training with 8 GPUs, and 3.58-71.46× for distributed training with 4 machines and 32 GPUs, with less than 1% accuracy loss, compared to DGL, the leading GNN system, as well as Legion [35] and DUCATI [45], the state-of-the-art approaches with the on-GPU cache.

In summary, we make the following main contributions:

- A comprehensive analysis about the tolerance of high graph feature compression ratios by GNNs, which reveals the main factors impacting training accuracy, and theoretically proves the convergence of F^{2}CGT by proposing a statistical bound on loss;
- A collection of three GNN-oriented graph feature compression methods that are tailored for sampling-based GNN training over large graphs, with necessary adaptions to align compressed features with data distributions of original graph features;
- A new sampling-based GNN training system F^{2}CGT that incorporates these compression methods and forms a feature-quantized data loading methodology for eliminates the feature loading bottleneck. F^{2}CGT improves on-GPU cache efficiency, enables fast preprocessing for offline feature compression, and leverages kernel fusion for accelerating online decoding and aggregation. Now it is open-sourced at [3];
- An in-depth evaluation of F^{2}CGT compared to compression enabled baselines with lossless or lossy compression, and non-compression baselines optionally with on-GPU cache, across 2 GNN models and 4 graph datasets, for single-machine or distributed training.

2 BACKGROUND AND MOTIVATION

2.1 Sampling-based Mini-batch GNN Training

GNNs training relies on both the graph structure and the features associated with nodes or edges, and commonly leverage GPUs for training acceleration [1]. However, when dealing with large-scale graphs, the size of the graph often exceeds the available GPU memory, making it impossible to store the entire graph in GPU memory. For instance, the MAG240M dataset, with a total size of 377 GB, exceeds the capacity of the device memory in the most advanced GPUs. To address this challenge, the current practice for training GNNs over large-scale graphs adopt a sampled-based mini-batch approach. As illustrated in Figure 1(a), this approach consists of three major steps, namely, **graph sampling**, **feature loading**, and **model updating**. These steps run sequentially but go through a large number of iterations towards model convergence. We will detail their explanation as follows.

Graph sampling. The GNN training procedure begins with sampling subgraphs from the original large graphs, where a neighbor sampling [15] method is often employed. It traverses the graph structural information from a set of frontier nodes, and sampling their neighboring nodes. It then continues to sample the next layer from the newly selected neighbors, and completes when the desired number of layers is reached. Despite the issue of over-smoothing [33] in GNNs, the sampling process typically involves 2 to 5 layers of sampling.

Feature loading. Once the sampled subgraph is formed, the next step is to gather the associated features of nodes or edges in that subgraph, which typically consist of high-dimensional vectors. Often, these feature vectors are placed on host memory due to their large volume. Therefore, the feature loading process collects the features, according to subgraphs, and then transfers them to the GPU for model training.

Model updating. Finally, taking the sampled subgraph and its associated feature vectors, the training procedure updates the target GNN model via the forward and backward steps. During these steps,

GNN training systems employ a message passing method [15], which iteratively aggregates node vectors from neighboring nodes to compute a node representation vector. This aggregation can be described by Equation 1 as below.

$$\mathbf{h}_i^{(t+1)} = \text{AGGRE}\left(\left\{\text{COMBINE}\left(\mathbf{h}_j^{(t)}, \mathbf{m}_{ji}^{(t)}\right) \middle| v_j \in \mathcal{N}(v_i)\right\}\right) \quad (1)$$

Here, $\mathbf{h}_i^{(t)}$ is the representation vector of node v_i in the t-th iteration, $\mathbf{m}_{ji}^{(t)}$ represents the message passed from node v_j to node v_i, and $\mathcal{N}(v_i)$ denotes the set of neighboring nodes of v_i. The COMBINE function merges the representation vector $\mathbf{h}_j^{(t)}$ with the passed message $\mathbf{m}_{ji}^{(t)}$. The AGGRE function is typically a reduction operation, such as the maximum or mean function.

2.2 Single-machine Training

When the graph data can fit into the memory or storage of a single GPU machine, the single-machine GNN training method is adopted. However, the limited per-GPU memory capacity poses a challenge, as it is typically much smaller than the size of the graph being processed. For example, the NVIDIA T4 GPU comes with 16 GB memory, while the Ogbn-Papers100M dataset is 79 GB. This tension will be amplified as in the real world, the graph size keeps increasing with its speed significantly higher than the increase in GPU memory capacity. To address this, as illustrated in Figure 1(b), the GNN training systems place graph structural information and feature vectors in host memory, and let GPU access these data through CUDA Unified Virtual Addressing (CUDA UVA) [5].

However, the previous studies [24], plus our experiments, have identified that the data processing steps of sampling subgraphs, and loading the associated features via PCIe are the major performance bottleneck. Furthermore, their time costs increase when the graph size expands. Table 1 illustrates this phenomenon, where transitioning from GNN training on the Products dataset (1.9 GB) to the Papers100M dataset (78 GB) results in an increase in the percentage of time spent on graph sampling and feature loading, from 36.9% to 42.0%. Furthermore, when increasing the training workload by expanding the set of frontier nodes in Papers100M dataset to 5% of the total graph nodes, the processing time cost ratio also increases from 42.0% to 65.7%.

Previous studies further point out that the vast majority of the loaded graph data are features. Unfortunately, this situation can be much worse since the state-of-the-art (SOTA) training systems such as AWS DGL [6] can further leverage GPU-based sampling [13], which improves the speed of model computation on graph data consumption and aggravates the bottleneck of feature data loading. Some approaches also incorporate an on-GPU cache, which keeps frequently visited graph data in GPU memory for reducing data transmission between the GPU and PCIe interfaces [24, 36, 45].

2.3 Distributed Training

When graphs are too large to be effectively processed on a single machine, the distributed GNN training method is used to leverage more resources such as GPU, CPU and memory from a cluster of machines. As illustrated in Figure 1(c), distributed GNN training incorporates graph partitioning techniques to shard the graph data into multiple parts, corresponding to the number of machines used. Though each machine manages its dedicated partition, the sampling and graph data loading steps have to still be distributed, due to the graph connectivity.

Table 1: Data processing (graph sampling + feature loading) time cost ratio of one epoch on GAT model. The notation 5% indicates that 5% of the total nodes in the graph comprise the training set, larger than the original training set size. This setting mimics the increase in the training workload. See Section 5.1 for detailed setting.

	Products	Papers100M	Papers100M (5%)
Processing Ratio (%)	36.9%	42.0%	65.7%

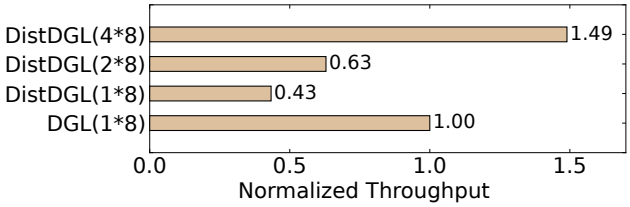


Figure 2: Training performance comparison of single-machine GNN training (DGL) and distributed GNN training (DistDGL) on Ogbn-Products dataset. The number of GPUs utilized is indicated in parentheses.

Table 2: DGL METIS 4-part graph partition cost

Dataset	Peak Memory (GB)	Time Cost
Ogbn-Products	30.6	245 seconds
Ogbn-Papers100M	650.9	5.9 hours

Compared to single-machine training, the distributed setting has much higher complexity with more severe performance bottlenecks, which lie in distributed sampling and graph data loading, and graph partitioning. The first bottleneck stems from the distribution of graph partitions across multiple GPU servers, which leads to a significant increase in cross-machine communication during distributed sampling and loading, even with the well-tuned METIS partitioning method [20]. Figure 2 demonstrates that, despite utilizing 4 machines and 32 GPUs, it is still unable to significantly surpass the training performance of a single-machine, 8-GPU GNN training configuration on the Ogbn-Products dataset. Additionally, the current distributed setting with graph partitioning precludes important optimizations, such as GPU-based graph sampling. This is because that distributed training should use on-CPU sampling.

Second, graph partitioning is a crucial aspect of distributed training. The contemporary GNN training systems, such as DGL [6] and PyG [29], commonly utilize METIS [20] for graph partitioning. METIS aims to minimize the number of cut edges across partitions, thereby reducing cross-machine communication during sampling and loading. However, it requires huge computational cost and

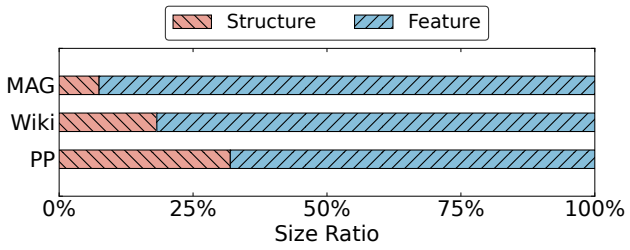


Figure 3: Structure and feature size ratio of Ogbn-Papers100M (PP), WikiKG90Mv2 (Wiki) and MAG240M(MAG) datasets.

significant CPU memory consumption. As shown in Table 2, using METIS for graph partitioning can take up to 5.9 hours, which are often much longer than model training, and consume memory of at least two or three times the size of the original graph.

2.4 Our Goals and Challenges

Observations. In this paper, our goal concentrates on eliminating the data processing bottlenecks of graph sampling and feature loading during GNN training. The primary solution we plan to apply is feature compression due to the following two key observations. First, graph features occupy a significant portion of storage space in real-world datasets. Figure 3 demonstrates that graph features account for approximately 68-93% of the total graph storage size across three widely used graph data sets in the GNN field. Even worse, during training, sampled subgraphs become sparser, leading graph structure to account for less than 2% of the loaded data volume. Second, the core training process of GNNs, as expressed in Equation 1, primarily depends on aggregating messages from neighboring nodes through reduction operations, which can tolerate a high compression ratio configuration possibly with negligible impacts on model convergence and accuracy (The detailed analysis can be found in Section 3.3).

System design opportunities. Through feature compression, the efficiency of on-GPU caching can be largely improved due to storing more feature vectors in GPU memory, during single-machine training. This speeds up the graph data loading since more features are served by on-GPU cache and the missing items transferred via PCIe will have smaller volumes. Consequently, feature compression-enabled training makes the single-machine training more scalable with respect to large graphs. In addition, it can give some spare GPU memory for caching graph structural information, leading to acceleration in on-GPU sampling.

In the case of distributed training, feature compression enables the compressed graph to have a size comparable to or even smaller than each graph partition on individual machines. This enables each machine to replicate the entire graph structure and compressed graph features, eliminating the need for graph partitioning, distributed graph sampling, and distributed feature loading. Furthermore, it allows distributed training to leverage optimizations, such as on-GPU caching and sampling, which are originally designed for single-machine training.

System design challenges. Nevertheless, the application of feature compression in GNN training still faces three challenges. The first one is how to strike the appropriate trade-off between compression ratio and the potential drop in model accuracy due to information loss led by feature compression. To address this issue, we propose a multi-level compression design based on the characteristics of data access patterns during sampled-based mini-batch training, where compression methods with lower compression ratio are applied to important features while the ones with much higher compression ratio are applied to features with lower priorities. Second, incorporating feature compression into data loading also changes the whole training process. In particular, it introduces complexity to the original cache system and cache policy, necessitating the need to update the on-GPU cache sub-system and cache policy accordingly. Third, compression comes with high computational and time costs, both offline and online. Therefore, efforts should be made to accelerate these processes and minimize their impact on training efficiency and costs.

3 GNN-ORIENTED FEATURE COMPRESSION

3.1 Design Rationale

The key is to find a proper compression method for reducing the feature loading volume in sampling-based GNN training. We first preclude lossless compression ones due to their extremely low compression ratios and high computation overhead, when being applied to float numbers of GNN graph features. As shown in Figure 4, for Ogbn-Products, fpzip [26] only achieves a compression ratio of 1.068, while resulting in more than 10× processing overhead, compared to lossy methods. On the other hand, as a representative lossy method in the scientific computing domain, zfp [25] compresses data by relying on spatial correlation in float arrays, where adjacent values are contiguous. However, as shown in Figure 4, zfp fails to deliver comparable accuracy. This is mainly because for graph datasets, there is no continuity between the feature values of adjacent vertex IDs or dimensions, for a given vertex.

We do not randomly choose existing compression methods, instead, we have drawn inspiration from the ones that have been extensively used in DNN models for quantizing various data types such as model weights, activations and gradients [14, 23, 37, 48]. Though they are not designed for GNNs and graph features, we still find their potentials to be the primary solution in our target context with necessary adaptions.

Most DNN data quantization algorithms are *scalar* quantization, which facilitates the representation of continuous values as discrete values, enabling the use of integers with lower bit-widths to represent floating-point numbers [43]. Numerous studies have adopted uniform scalar quantization, which maps floating-point values into fixed intervals [9, 37, 41]; however, this approach is not optimal for graph features and GNN training. Figure 4 shows that the uniform distribution setting makes the GraphSAGE model with scalar quantization enabled experience substantial accuracy drop. This connects to our observations indicate that the features of most large-scale graphs used for GNN training approximate a normal distribution rather than a uniform one. This is expected since these features are often derived from the outputs of upstream models. For example, in the MAG240M dataset, each paper node represents an arXiv paper, and its feature is a 768-dimensional output generated

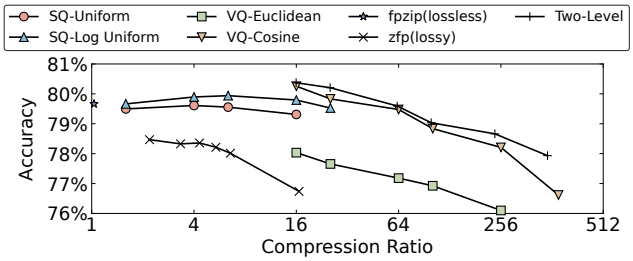


Figure 4: Accuracy performance of different compression methods when training GraphSAGE on Ogbn-Products.

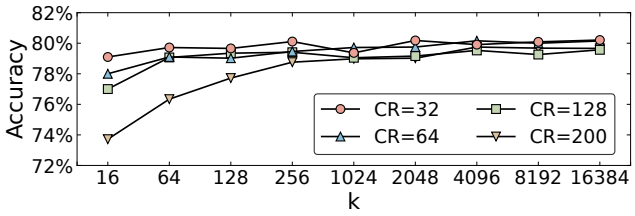


Figure 5: Accuracy performance of different k in vector quantization when training GraphSAGE on Ogbn-Products, each line represents a fixed compression ratio (CR).

by BERT. Therefore, we have devised a new scalar quantization method tailored for the distribution of features of graph datasets.

Unfortunately, when processing graphs at large scales using high-end hardware, the efficacy of scalar quantization may be limited, given its maximum compression ratio of 32 for binary quantized values. Hence, it is imperative to explore some other quantization methods offering significantly higher compression ratios. Here, we focus on vector quantization, which represents multiple floating-point numbers as discrete values by a single integer. It employs the k-means clustering algorithm to organize feature vectors into distinct clusters, each of which is presented by its centroid. These centroids are cataloged in codebooks, and their indices facilitate retrieval of the associated values. Figure 4 highlights that GraphSAGE with vector quantization and the widely-used Euclidean distance metrics for k-means clustering achieves a high compression ratio of up to 256, at the cost of substantial drops in accuracy. Equally importantly, compared to scalar counterpart, vector quantization imposes significantly higher preprocessing overhead for offline compression and online decompression. In summary, the negative accuracy and performance implications of vector quantization lead us to design a new vector quantization method and also think of how to make the best use of it.

3.2 Feature Compression Methods

3.2.1 Scalar quantization with log uniform distribution. To adapt scalar quantization to data distribution of graph features, we introduce a log-uniform scalar quantization, which concentrates the quantization of central data more effectively, further reducing quantization error, which is better for GNN training and improving model accuracy. The idea is summarized in Equation 2. x represents the original feature, while e_{min} and e_{max} represent the minimum and maximum values, respectively, obtained from the binary logarithm of the non-negative x after clipping outlier values (i.e., $Clip(log_2|x|)$). We empirically clip 0.5% of values with largest and smallest absolute value, and varying the portion has little impacts on model accuracy. The parameter k refers to the target value bit-width. This quantization method essentially performs a uniform quantization of the clipped logarithm.

$$Q(x) = \begin{cases} -\lceil \frac{Clip(log_2(-x)) - e_{min}}{e_{max} - e_{min}} * 2^{k-1} \rceil, & x < 0 \\ \lfloor \frac{Clip(log_2 x) - e_{min}}{e_{max} - e_{min}} * 2^{k-1} \rfloor, & x \geq 0 \end{cases} \tag{2}$$

We highlight that this formula also works for features that do not follow a normal distribution. This is because we do not differentiate zero and negative values for the compatibility of discrete original values like one-hot encoding [32] and binary features [16], which are commonly used in smaller-sized graphs. In in Figure 4, we have experimentally proved that log-uniform quantization is more effective than the uniform one with better training accuracy guarantees. For instance, when setting the quantization bit-width of Ogbn-Papers100M to 2, the log-uniform quantization achieves 0.3% better training accuracy than its uniform counterpart.

3.2.2 Vector quantization with cosine similarity. We propose to use cosine similarity rather than Euclidean to measure vector distances in k-means, which can produce more balanced centroid results. The cosine similarity is summarized in the following equation:

$$\text{Similarity}(x_1, x_2) = 1 - \frac{x_1 \cdot x_2}{\|x_1\|_2 \cdot \|x_2\|_2} \tag{3}$$

where x_1 and x_2 represent two vectors. This formula first computes the cosine value between the two vectors and then subtracts it from 1 to produce non-negative results. Using cosine similarity helps almost evenly partitions input features. Figure 4 demonstrates that cosine similarity significantly outperforms Euclidean distance in GNN feature quantization. For instance, when setting the compression ratio to 256 for the Ogbn-Products dataset, vector quantization with cosine similarity achieves 2.1% better accuracy than the counterpart with Euclidean distance.

Note that there is an important parameter k in the k-means clustering algorithm, which represents the number of centroids to be generated in the k-means clustering algorithm, and determines the quality of vector quantization. If k is too small, the features will be quantized into a limited number of centroids without enough information for decompression. Thus the accuracy of the model will be greatly reduced. On the other hand, too large k imposes heavy quantization overhead, as the time complexity of k-means is $O(knm)$, where n is the number of vectors and m is the dimension of the vectors. Furthermore, k affects the maximum compression ratio that the vector quantization can achieve. This is because the larger k is, the more bytes are required for storing centroids. We conduct an empirical study to understand the trade-offs between model accuracy and compression ratios w.r.t varied k choices. Figure 5 shows that 256 is a good choice as setting k to 256 can achieve 78.3% accuracy, as comparable as 78.5%, delivered by DGL without feature compression, even at a high compression ratio of 200.

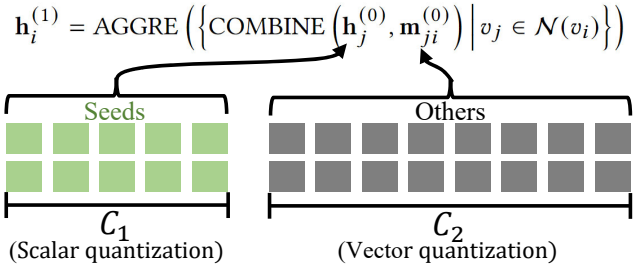


Figure 6: Two-level compression, compression ratio $C_1 < C_2$

3.2.3 Two-level quantization. To achieve a high compression ratio without compromising model accuracy, we propose a GNN-specific two-level feature compression strategy. By increasing the compression ratio, we can significantly reduce data volume transferred over intra-machine PCIe or cross-machine links and improve GPU cache efficiency for graph structural information and features vectors, thereby enhancing training performance.

Figure 6 illustrates the high-level idea of our two-level compression approach. Our approach divides graph nodes within one sampled subgraph in GNN training into two categories. The first category consists of the seed nodes which the sampling procedure begins with. Their features correspond to $\mathbf{h}_i^{(0)}$ in Equation 1, and have more impact on the model output, pivotal for model accuracy. Therefore, we have to minimize information loss for these features by limiting the compression ratios. To this end, our approach assigns the first level compression method—scalar quantization—with lower compression ratios to compress them. In contrast, for other graph nodes, their errors have much smaller impacts on model outputs since they have to go through at least one more aggregation step, canceling out more errors. Therefore, our approach adopts a more aggressive compression strategy—vector quantization—for nodes other than seeds, achieving significantly high compression ratios to reduce the volume of graph data.

Upon the use of the two-level feature compression strategy, a reduction of approximately 70% in graph data volume can be achieved. For instance, when employing a compression ratio of 58.4, the Ogbn-Papers100M dataset shrinks from 79 GB to 28 GB, a size comparable to one segment of a four-machine graph partition (36 GB), which can be replicated across all machines. This can completely eliminate cross-machine communication of data processing.

Note that when using two-level quantization, we only distinguish between seed nodes and other nodes in the graph. Seed nodes are from train/test/valid nodes and act as root nodes in each sampled subgraph. We do not require these train/test/valid nodes to be labeled, thus the two-level quantization can work for most GNN training tasks, not just semi-supervised learning ones. However, we do impose a constraint that train/test/valid nodes should be known prior to training, since the compression is done offline.

3.3 Theoretical Accuracy Analysis

In addition to experimental validation, we conduct a theoretical analysis to explore the impact of feature compression errors on GNN training, i.e., we prove that GNN training with feature compression will converge as usual and achieve a low increase in loss[3]. We find that GNNs' unique properties often lead to a relatively small increase in loss. With appropriate compression parameter selection, the impact on model loss can be negligible. We formally prove the upper bound of loss increase and identify key factors affecting the magnitude of loss increase, including GNN model depth and graph structural characteristics. Here, we define the bound with ϵ, compared to the optimal loss achieved with original features.

We build our proof upon the prior works of the convergence properties of over-parameterized conventional neural networks [12], and extend the theoretical framework to GNNs. The necessary formulations are as follows. Let $X = (x_1, x_2, ..., x_N) \in \mathbb{R}^{N\times d}$ represent input features, where N is the number of nodes and d is the number of feature dimensions. The quantized features are denoted as $X' = (x'_1, x'_2, ..., x'_N) \in \mathbb{R}^{N\times d}$ with $|x_i - x'_i|_2 \leq \delta$ for all $i \in [N]$.

Assumption 3.1 (Independent error). *Randomness of rounding error is independent among nodes and feature dimensions.*

Remark. Despite the relevance of a node's neighboring nodes' features, the compression error from rounding is normally considered independent [43], being a high-order small quantity.

Assumption 3.2 (Assumption on loss function). *The loss function is Lipschitz-smooth and convex.*

Remark. This assumption is commonly used and necessary within the framework of machine learning optimization theory [12]. Mean Squared Error (MSE) [2] and Cross-Entropy(CE) [27] are two frequently used loss functions. MSE satisfies the assumption by design. Unlike this, CE is convex but not entirely Lipschitz-smooth, as its gradients have no bounds as the input values approach 0 and 1. However, CE still satisfies this assumption in practice, since typically it is used along with softmax, which constrains input values so that they are far from 0 and 1, and gradients are bounded.

Theorem 3.3. *Given $\epsilon > 0$, L-layer GNN with a large enough width m and quantization error $\delta \leq \frac{\epsilon}{\hat{C}_L}$ satisfying Assumption 3.1, with loss function satisfying Assumption 3.2. If we run gradient descent for T steps, then with a high probability, we have*

$$\min_{t=1,\cdots,T}(Loss(W_t, X') - Loss(W_*, X) \leq O(\hat{C}_L\delta) \leq \epsilon$$

Remark.[4] $L(W_*, X)$ is the optimal loss of GNNs trained with original features, and $L(W_t, X')$ is the minimum loss with F^{2}CGT. So this implies the difference of loss with F^{2}CGT methodology and original features is bounded. With δ representing the scale of input error, the factor $\hat{C}_L$ delineates the sensitivity of an L-layer GNN to the input error. This sensitivity is influenced by both the number of GNN layers and the structure of the target graph.

Our proof realizes the fundamental difference between GNNs and conventional DNNs, which lies in the GNN's aggregation step. Based on the assumption of independently quantized errors, we demonstrate that the relative scale of errors decreases at each layer's aggregation stage and does not significantly impact the loss to which the model eventually converges. We use the parameter $\hat{C}_L$ to denote the scale of error contributions relative to the loss and

[3]The low increase in loss implies the negligible impacts on model accuracy.
[4]The detailed proof of Theorem3.3 is present in supplemental material [4].

input features, which typically remains below 0.1 for GNNs. In contrast, for traditional DNNs, $\hat{C}_L$ remains equal to 1 as they lack the aggregation stage to mitigate these influences.

4 DESIGN AND IMPLEMENTATION

We introduce our system F^{2}CGT, which offers Fast Feature Compression enabled GNN Training, eliminating the data processing bottlenecks across single-machine and distributed environments.

4.1 System Overview

F^{2}CGT incorporates the aforementioned scalar quantization (SQ), vector quantization (VQ) and two-level (TL) quantization strategy for features into the conventional GNN training workflow. As shown in Figure 7, F^{2}CGT is comprised of three key system components: an offline feature compressor, a high-efficiency GPU cache sub-system, and a high-performance runtime decompressor.

Prior to training, F^{2}CGT performs offline preprocessing (Section 4.2) to produce compressed feature vectors. Recognizing that decompression introduces extra overhead during training, F^{2}CGT incorporates high-performance decompression techniques to mitigate this overhead and fuses decompression and aggregation steps to further improve performance (Section 4.4). In addition to the offline component, at runtime, F^{2}CGT employs on-GPU caching, which caches both graph structural information and compressed feature vectors, according to their access frequencies (detailed cache policies can be found in Section 4.3).

In single-machine training, F^{2}CGT populates the on-GPU cache with both graph structure and compressed feature vectors, chosen based on a cost model and the specified cache policy. Subsequently, F^{2}CGT runs on-GPU sampling to extract subgraphs, followed by retrieving compressed feature vectors either from the on-GPU cache (cache hits) or host memory (cache misses). Once the data is ready, feature vector decompression is conducted by F^{2}CGT in GPU. Finally, F^{2}CGT feeds the subgraphs and decompressed features into the GNN model for training. When involving multiple GPUs, an allreduce operation is initiated to synchronize the gradients generated by each GPU for global model parameter updating.

F^{2}CGT can avoid graph partitioning, enabling each machine to retain both the entire graph structural information and the compressed graph feature vectors. In this case, the distributed training works very similarly with the single-machine setting. Graph sampling can perform locally within each machine, starting from assigned per-machine train/test/valid node nids and determines different training tasks. Likewise, feature loading is also done locally.

4.2 Offline Preprocessing

At offline, F^{2}CGT performs two primary tasks as follows. First, it applies compression to all feature vectors, which results in the compressed feature vectors that will be replicated across all machines. Second, it needs to determine graph node hotness, which is crucial for improving the efficiency of on-GPU cache, particularly, in cooperation with feature compression.

Feature Compression. Both scalar and vector quantization methods require collecting statistics from graph datasets for performing feature compression, i.e., the max, min, and mean values of features for scalar methods, while the centroid results of the k-means clustering algorithm for vector quantization. We generalize the two methods and propose an unified feature compression algorithm, which fully exploits massive GPU parallelism for acceleration. From a per GPU's perspective, Algorithm 1 takes GPU id (gid), feature vectors, the compression method (SQ or VQ), and the compression ratio (CR) as input, and executes three common steps. At the first step, each GPU samples a subset of features. Then, each GPU collects the aforementioned statistics independently, followed by an all-gather operation to synchronize to produce global compression statistics. At the final step, each GPU consumes these statistics to perform independent compression over its own feature partition, followed by another all-gather to create the entire compressed feature map. In addition, for the two-level quantization method, we execute the algorithm twice, where the first execution with seed node features and the SQ method as inputs, while the second run with other node features and the VQ method.

Algorithm 1 Feature Compression

Input: features ($N \times D$), gid(GPU ID), method (VQ or SQ), CR(Compression ratio)
Output: Compressed Data

```
index ← Sampling(N)
sub_features ← feature[index, :]        ▷ Step 1: Sample a subset

col_slice_index ← Slice(D, gid, NUM_GPUS)
part_parameters ← GenParameters(sub_features[
        :, col_slice_index], method, CR)
parameters ← AllGather(part_parameters)
                                  ▷ Step 2: Generate parameters

row_slice_index ← Slice(N, gid, NUM_GPUS)
row_compressed_data ← Compress(features[
        row_slice_index, :], parameters, method, CR)
compressed_data ← AllGather(row_compressed_data)
                                  ▷ Step 3: Apply feature compression
return compressed_data
```

Caching-related computation. To guide on-GPU cache for data selection, F^{2}CGT incorporates pre-sampling to compute node access hotness. The pre-sampling method runs the sampling phase of the end-to-end training but not the training phase, to capture the memory access patterns across iterations of sampling-based mini-batch training. This computed hotness informs F^{2}CGT's cache policy and cost model, guiding decisions on caching graph structures and compressed features under the GPU memory constraints.

Cost discussion. Compared to graph partitioning, the offline preprocessing in F^{2}CGT demands less memory consumption and time costs while scaling to comparable graph sizes. In terms of memory usage, graph partitioning usually employs coarsening and uncoarsening algorithms [20], resulting in numerous intermediate, smaller graphs that consume a substantial amount of memory. In contrast, F^{2}CGT primarily allocates memory for the graph itself, producing minimal intermediate data via batch processing. Regarding time efficiency, the intricate nature of graph partitioning precludes the usage of GPU parallelism. Unlike this, the offline preprocessing in

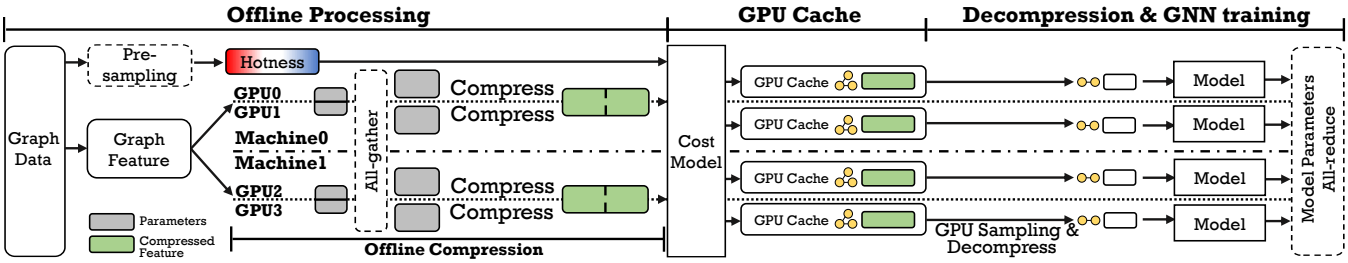


Figure 7: The overall architecture of F^{2}CGT and workflow of the feature compression-enabled GNN parallel training.

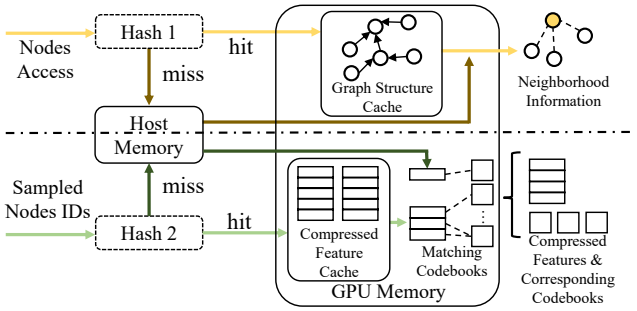


Figure 8: The workflow of on-GPU Cache

F^{2}CGT benefits from a streamlined feature compression algorithm that facilitates straightforward parallelization.

4.3 New Cache Design

F^{2}CGT co-designs on-GPU cache with feature compression to further eliminates data loading bottleneck. The novelties of our cache design are two-folded. First, we propose a holistic collection of GPU memory optimizations (Section 4.3.1 4.3.2), such as partitioning space and designing efficient data structure, for maximizing cache space for graph structure and compressed feature vectors with different compression ratios. Furthermore, this also includes a computation fusion of feature decompression and GNN aggregation for both memory and computation optimization (Section 4.4). Second, we propose a general cache policy (Section 4.3.3), which greedily searches the proper cache space allocation plans for various data types and chooses cache candidates for maximized cache hit rates, by jointly and efficiently considering a large number of factors such as node hotness, I/O sizes, compression ratios, hardware capacities, etc. These points make F^{2}CGT's cache different from related approaches such as Legion [35] and DUCATI [45].

4.3.1 Cache Structure. As shown in Figure 8, the cache consists of three components: graph structure cache, segmented cache for compressed features, and buffer for the feature compression codebooks. The graph structure cache stores a set of key and value pairs, where key refers to each selected node, according to their hotness, and value keeps the list of IDs of the selected node's out-edge neighbors, organized in a compressed sparse row (CSR) format. The segmented feature cache consists of two 2D arrays, each of which stores the compressed node features of the selected nodes. The reason for maintaining two segments in the feature cache is that the two-level compression strategy will result in compressed features with two different sizes and the separation will lead to better memory access efficiency. Finally, the compressed codebook cache retains all codebooks generated during preprocessing, stored in multiple 2D arrays, which are subsequently used for feature decompression.

4.3.2 Cache Management. Figure 8 also illustrates the workflow within F^{2}CGT's on-GPU cache. When sampling starts, on-GPU cache performs lookups in a GPU hashmap to determine whether a requested graph node's neighbor data are in GPU memory. In the case of a cache hit, the cache system retrieves the neighboring nodes of the accessed node to form the next-layer sampling frontier (the top part of Figure 8). Otherwise, on-GPU cache fetches the missing structure information from host memory. Upon the readiness of the sampled subgraph, on-GPU cache performs lookups in the segmented feature cache. If cache hit, the target feature vector is served by GPU; Otherwise, the missing feature is collected from host memory. Following this, to facilitate feature decompression during the model updating phase, it also needs to fetch the corresponding codebooks. Given their frequent accesses during the decompression process, codebooks, typically several megabytes in size, are fully stored within the GPU cache. Finally, the on-GPU cache returns the set of compressed features along with their corresponding codebooks, either from GPU memory or host memory.

4.3.3 Cache Policy. A suitable cache policy is crucial for determining the efficiency of F^{2}CGT's on-GPU cache due to the following reasons. First, it is important to determine the cache space allocation for both structure and compressed features so as to fully utilize the available GPU memory and maximize the joint benefits. Second, the cache space allocation and performance optimization are also subject to the compression ratios and the node hotness. To balance these aspects, we propose the following cost model to formally define the cache efficiency.

Initially, we evaluate the time cost T_e associated with each memory access for entry e, either a neighbor list in CSR format or a feature vector. We have $T_e = k_e \times A_e$, where A_e denotes the accessed bytes required for entry e and k_e is a derived parameter. Crucially, it's imperative to recognize that the value of k_e varies based on whether e resides in CPU memory or GPU memory. The expected time saving caused by caching e can be calculated by multiplying hit probability p_e and benefit of each hit, i.e. $\Delta T_e = p_e \times (T_e^{CPU} - T_e^{GPU}) = p_e \times (k_e^{CPU} - k_e^{GPU}) \times A_e = p_e \times \hat{k}_e \times A_e$. Here, k_e^{CPU} and k_e^{GPU} represent k_e when e is on the CPU or GPU, respectively, p_e can be calculated by $p_e = Hotness(e)/\sum Hotness$.

Considering hotness and storage cost, we can calculate the benefits for caching node feature vector or neighbor list using the following formula, where $Hotness(e)$ represents the hotness of entry e and S_e signifies the memory storage size of entry e.

$$B_e = \frac{\Delta T_e}{S_e} = \frac{Hotness(e)}{\sum Hotness} \times \hat{k}_e \times \frac{A_e}{S_e}$$

In F^{2}CGT, the hotness derived from pre-sampling [44] informs the computation of hit probability. To simplify implementation and minimize overhead, when e denotes a node's neighbors, F^{2}CGT employs the average count of sampled neighbors for each node throughout the sampling process as A_e. Furthermore, it utilizes the average $\hat{k}$ value in lieu of $\hat{k}_e$ These average slope values are determined through straightforward profiling. By adopting this methodology, F^{2}CGT harmonizes the benefits derived from caching graph structures and compressed features, factoring in all considerations like hotness, access size, and storage dimensions. Subsequently, F^{2}CGT arranges all B_e values in descending order and prioritizes caching based on the most significant data elements.

4.4 Implementation Details

F^{2}CGT runs atop DGL [6] and PyTorch [30], with 3 k lines of code, 760, 1689, and 565 lines for offline preprocessing, on-GPU cache, and decompression function, respectively.

Decompression and aggregation fusion. F^{2}CGT introduces an online feature decompression step, which converts compressed features into their original size, and then passes decompressed data to the aggregation computation of GNN training. The decompressed input features occupy a major portion of the GPU memory space, ranging from hundreds of MBs to several GBs. Based on this observation, we further fuse the decompression operator and the aggregation operator into a single operator. This brings two benefits. First, it improves the computation efficiency by reducing overhead of memory accessing. Second, it eliminates the need to allocate GPU memory for storing intermediate results after decompression, leaving more GPU memory available for the cache system. We conduct performance evaluation in Section 5.5.

PCIe transfer acceleration. While feature compression improves GPU cache efficiency, intermittent memory accesses to the host memory can still arise. To expedite these occasional accesses, we leverage CUDA Unified Virtual Addressing (UVA) [5], enabling the GPU to access CPU memory at the cache-line level with reduced latency and without the need for additional memory copies.

Compression method choices and ratio determination. For a given GNN training task, we propose a simple method to automatically choose the proper feature quantization method out of the three ones introduced in Section 3.2, as well as its compression ratio. It first estimates the minimum compression ratio r at which the task can run on a large graph, by inspecting the training hardware setups (e.g., per-machine host memory size, per-GPU memory capacity, etc), subgraph sampling patterns, and statistics of graph datasets (e.g., scale, feature dimensions, seed ratios, etc). If r is smaller than 32, then we set it to 32 and use *scalar quantization* with log-uniform distribution, which is sufficient to alleviate data loading bottlenecks, while preserving comparable accuracy, according to the analysis in Figure 4. Otherwise, our configuration method suggests to use the two-level compression method. In this case, we let the non-seed nodes of the target graph use *vector quantization* with a compression ratio of r. In addition, we assign *scalar quantization* for seed nodes with a compression of r'. We compute r' by balancing the contributions from the input errors of seed nodes and other nodes on the output error. Here, we introduce a formula $r' = 64 \cdot r/(64 + (\log_2 fan\_out + k) \cdot r)$, where k is a knob controlling contribution of seed nodes and other nodes, and fan_out is a sampling parameter of the last layer of GNN model. In Section 5.1 and Table 3, we apply this method to various training tasks over four datasets to generate compression configurations.

Table 3: The graphs and compression strategies used in the evaluation. TL, VQ, and SQ stand for two-level, vector, and scalar quantization, respectively. The value represents compression ratio and for TL, ratio of both levels are shown.

Dataset	Abbr.	|V|	|E|	Size (GB)	Config
Ogbn-Products	PD	2.4M	123M	1.9	VQ64
Ogbn-Papers100M	PP	111M	3.2B	77.8	SQ32
Friendster	FS	124M	1.8B	133.4	SQ32
MAG240M	MAG	244M	3.5B	376.8	TL8,128

5 EVALUATION

5.1 Experimental Setup

Experiment platform. For our experiments, we selected AWS instance g4dn.metal [34], which offers 96 vCPUs, 384 GB memory, and 8 NVIDIA T4 GPUs with 16 GB memory each. These instances are interconnected by 100 Gbps network. Our operating environment includes Ubuntu 20.04 with NCCL 2.18.6, DGL 1.1.3 [6], PyTorch 2.1.2 [30], OpenMP 4.5, and CUDA 11.8.

Datasets. We use four graph datasets (Table 3). First, we use a medium dataset, Ogbn-Products (PD) [18]. Additionally, we incorporate three large-scale graph datasets: Ogbn-Papers100M (PP) [18] and MAG240M (MAG) [17], both derived from academic sources (Microsoft Academic Graph) [42], and Friendster (FS) [38], a social network within an online gaming platform. As the FS only contains topology message, we generate the training set, labels, and 256-dimensional features for it. All the graph datasets used are homogeneous and undirected. For F^{2}CGT, we select the compression configurations (Table 3) according to the guideline in Section 4.4. Specifically, for Ogbn-Products, we use VQ with a compression ratio of 64 to show the performance of vector quantization.

Models. We utilize three layers of GraphSAGE [15] and GAT [39] with a "20-20-20" neighbor sampling method and a batch size of 1536 for RD, PD, PP, and FS datasets. For the MAG dataset, a "5-10-15" neighbor sampling is employed due to the GPU memory limitation. By default, the hidden dimension for the models is 128.

Baselines. We conducted evaluations on both single-machine training and distributed training scenarios. For the single-machine experiments, we compared our method, F^{2}CGT, against three existing frameworks: DGL, DUCATI [45], and Legion [35]. DGL is a widely used and leading GNN framework, in which we enabled GPU sampling and feature prefetching but did not utilize any GPU cache. DUCATI and Legion are state-of-the-art systems that accelerate GNN training by caching graph structures and features. Legion

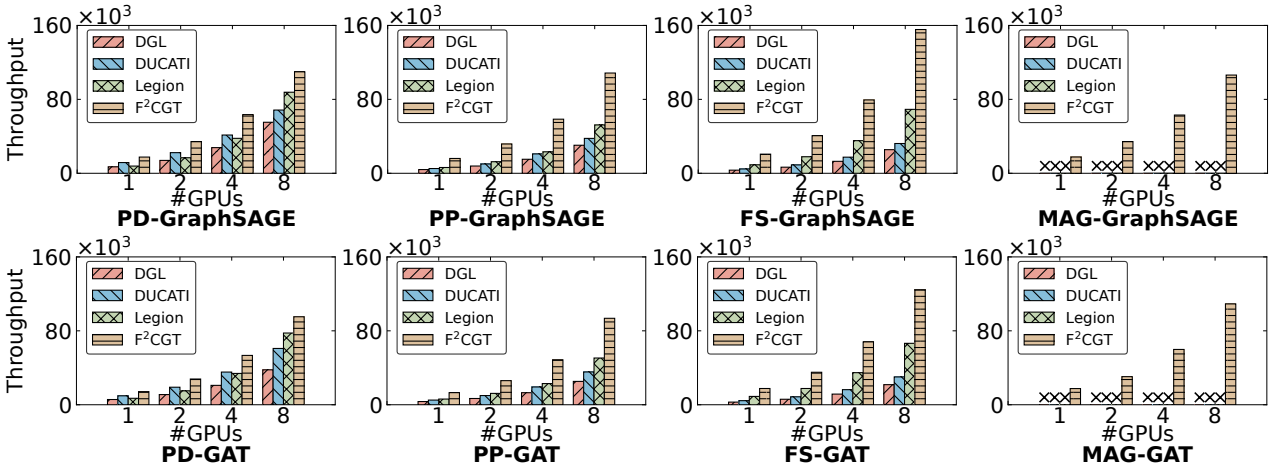


Figure 9: Single-machine training throughput comparison for DGL, DUCATI, Legion and F²CGT.

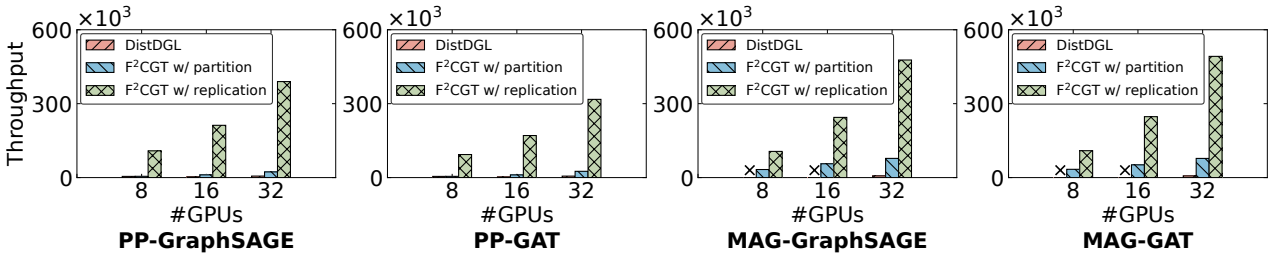


Figure 10: Distributed training throughput comparison for DGL and F²CGT using GraphSAGE and GAT on PP and MAG.

further leverages graph partitioning to improve cache efficiency. However, the partition algorithm xtraplup used by Legion fails to balance computation cost, i.e., the number of the training nodes, across trainers for real-world datasets. To address this, we replace xtraplup with METIS and observe that this replacement makes Legion perform better. For the distributed training experiments, we compared our method, F²CGT, with DistDGL [47], which is a version of DGL optimized for distributed training.

5.2 Single-machine Training

Overall performance. We compare F²CGT with DGL, DUCATI, and Legion. For DUCATI, Legion and F²CGT that support on-GPU cache, we set the cache capacity to the maximum. Figure 9 illustrates the single-machine training performance. DGL typically exhibits the worst performance, primarily because of its extensive PCIe feature transmission between the CPU and GPU. DUCATI optimizes performance by caching both the graph structure and features in GPU memory, thereby minimizing PCIe transmission. Legion performs better than DUCATI due to two reasons: (1) an advanced GPU memory management method that improves on-GPU cache's capacity; and (2) the combination with graph partitioning to improve cache efficiency. Impressively, F²CGT delivers 1.25-2.56× throughput speedups (number of samples processed per second) for the GraphSAGE model, and 1.23-2.12× speedups for the GAT model, respectively, compared to the best-performed baseline across the PD, PP, and FS datasets. Notably, for the MAG dataset, only F²CGT can successfully complete the training process, while the others fail since the dataset size exceeds the CPU memory limitation.

For the medium-size PD dataset, DUCATI and Legion are all able to cache the entire dataset in GPU memory without compression. Even though, F²CGT with feature compression still outperforms the best-performed baseline by 22.7-54.0%. This enhanced performance is mainly because of the reduced GPU memory usage by fusing the decompression and aggregation operators. F²CGT delivers better improvements when processing larger datasets. For the PP and FS datasets, F²CGT achieves 107.4-155.9% throughput increases over Legion. This is because, with feature compression, F²CGT can cache significantly more features than Legion.

Impact on cache. We performed a comprehensive analysis of feature compression's impact on GPU cache, as detailed in Table 4. F²CGT demonstrates an enhanced cache ratio for both structural and feature components. Specifically, on the PP and FS datasets, F²CGT outperforms Legion, realizing an average enhancement of 35.5% and 84.3% in structure and feature caching, respectively. Due to the lower memory consumption caused by kernel fusion, F²CGT retains ample working space, enabling F²CGT to effectively cache a greater portion of features following feature compression. As a result, F²CGT consistently maintains a significant cache ratio for graph structures and features.

Impact on PCIe transmission. We also delve deeper into the impact on PCIe transmission, as depicted in Table 5. While Legion

Table 4: The ratios of cached graph structure and features, from a single trainer's perspective. Every $v1, v2$ tuple means $v1$ cache ratio for structure and $v2$ cache ratio for features.

Sys.\Data	PD	PP	FS	MAG
DUCATI	100%,100%	0.10%,3.72%	0.22%,1.64%	N.A.
Legion	100%,100%	0.03%,21.0%	0.05%,10.5%	N.A.
F^{2}CGT	100%,100%	40.5%,100%	30.5%,100%	27%,53%

Table 5: The PCIe transmission volume during one epoch, from a single trainer's perspective. Every $v1 + v2$ tuple means $v1$ GB of data transmission during graph sampling and $v2$ GB during feature loading.

Sys.\Data	PD	PP	FS	MAG
DGL	1.94+16.07	4.56+121.87	2.84+275.36	N.A.
DUCATI	0+0	4.29+48.58	2.83+195.18	N.A.
Legion	0+0	3.89+29.24	2.83+157.30	N.A.
F^{2}CGT	0+0	0.10+0	2.07+0	0.09+0

has successfully minimized data transfers between the CPU and GPU, F^{2}CGT achieves even further reductions in data movement volume. Specifically, for datasets like PP and FS where the GPU cannot cache the entire dataset, F^{2}CGT averages a reduction of 62.1% in data transmission during graph sampling and 100% during feature loading, compared with Legion. This efficiency stems from F^{2}CGT's enhanced GPU cache ratio and the diminished overhead associated with loading compressed features via PCIe.

5.3 Distributed Training

For distributed training, we compare F^{2}CGT with DistDGL, which partitions the graph structure and features, and assigns disjoint partitions to each machine, leading to distributed graph sampling and feature loading. We configure F^{2}CGT with two variants, namely, F^{2}CGT w/ partition and F^{2}CGT w/ replication. Both variants enable feature compression. However, the first variant combines graph partition and feature compression by further partitioning graph structure and distributing slices to multiple machines. With feature compression, graph structure makes up the vast majority of the dataset. F^{2}CGT w/ partition leads to cross-machine sampling. Unlike this, the second variant replicates graph structure and compressed features across all machines.

Overall performance. Figure 10 illustrates the distributed training throughput on PP and MAG datasets. DistDGL performs worst, F^{2}CGT w/ partition outperforms DistDGL, and F^{2}CGT w/ replication performs best. F^{2}CGT w/ partition and F^{2}CGT w/ replication achieve 3.58-11.31× speedups and 52.75-71.46× speedups, respectively, across more than one machine. Both F^{2}CGT variants achieves great speedups due to the remarkable reduction in communication volume. As shown in Table 6, DistDGL introduces 12.0 GB and 386.4 GB data across-machine network consumption per training epoch for distributed sampling and feature loading for the MAG-GraphSAGE training task, respectively. F^{2}CGT w/ partition does not pull features from other machines (network usage for this part drops to zero), but still perform distributed sampling due to graph structure partitioning. Finally, F^{2}CGT w/ replication makes the best use of feature compression and replicate both graph structure and compressed feature across machines, thus completely eliminating cross-machine communication (both are zero).

Table 6: Volume of network communication for a single epoch distributed training across 4 machines. Every $v1 + v2$ tuple means $v1$ GB of network communication during graph sampling and $v2$ GB during feature loading.

	PP	MAG
DistDGL	12.0 + 386.4	1.7 + 233.7
F^{2}CGT w/ partition	12.0 + 0	1.7 + 0
F^{2}CGT w/ replication	0 + 0	0 + 0

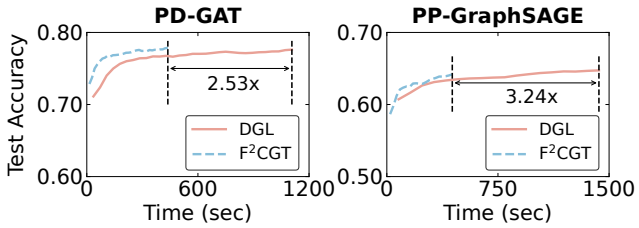


Figure 11: Training convergence speedup with 8 GPUs

Preprocessing cost. Table 7 reports the time, memory requirements, and output sizes relevant to feature compression and graph partitioning. F^{2}CGT exhibits significant time efficiency advantages for large-scale graphs, consuming diminished CPU resources and producing smaller output sizes. Leveraging GPU acceleration, F^{2}CGT completes the preprocessing task within minutes, leading to notable time savings. For memory utilization, F^{2}CGT minimizes the generation of intermediate data. Conversely, METIS requires substantial memory allocations for storing intermediate data. Furthermore, by adopting feature compression, F^{2}CGT efficiently reduces feature dimensions, achieving output sizes comparable to those of individual partitioned graphs in METIS.

5.4 Accuracy and Convergence Speedup

We performed convergence validation experiments using single-machine training with 8 GPUs (Figure 11). First, F^{2}CGT delivers comparable convergence results as DGL, the baseline with no compression. For instance, F^{2}CGT achieves 77.5% and 64.4% accuracy for PD-GAT and PP-GraphSAGE, respectively, while DGL's records are 77.4% and 64.9%. Second, in addition to the throughput speedups shown before, here, we report the end-to-end training improvements, measured by the time to convergence. For the PD-GAT and PP-GraphSAGE tasks, F^{2}CGT achieves an 2.53× and 3.24× convergence speedup, respectively. These results are consistent with the throughput speedups (i.e., 2.53× for PD-GAT and 3.60× for PP-GraphSAGE in Figure 9). More results about accuracy with different tasks and datasets can be found in the supplementary material [4].

Table 7: Preprocessing cost for F^{2}CGT (feature compression) and METIS. For METIS, the output size represents the average size of the partitioned graphs.

Data	Sys.	Time Cost	Peak Memory	Output Size
PD	METIS	248 sec	31 GB	1.1 GB
	F^{2}CGT	0.68 sec	11 GB	1.0 GB
PP	METIS	5.98 hrs	641 GB	35 GB
	F^{2}CGT	10.8 sec	124 GB	27 GB
FS	METIS	3.95 hrs	500 GB	44 GB
	F^{2}CGT	14.8 sec	267 GB	32 GB
MAG	METIS	6.24 hrs	999 GB	112 GB
	F^{2}CGT	55.9 sec	150 GB	31 GB

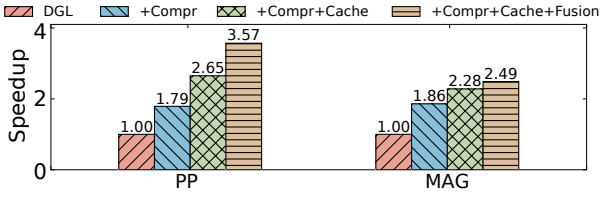


Figure 12: Speedups of F^{2}CGT, normalized to DGL

5.5 Ablation Study

Next, we evaluate the individual performance gains of three optimizations we introduced, namely, feature compression (**+Compr**), on-GPU caching (**+Cache**), and decompression and aggregation fusion (**+Fusion**). We report the normalized speedups against DGL when enabling optimization one by one for training GraphSAGE across 8 GPUs in Figure 12. For the PP datast, compared with DGL, **+Compr** greatly enhances performance by 79%, while combining **+Compr** with **+Cache**, the performance of F^{2}CGT is further improved by 49%. Finally, when incorporating **+Fusion**, the performance of F^{2}CGT is profoundly enhanced by 35%. We observe similar performance trends for MAG dataset. For instance, gradually adding the three optimizations introduce 86%, 128%, and 149% higher throughput over DGL, respectively. We also further evaluate the time ratios of sampling and feature loading, both affected by compression, with respect to different optimizations. For the large MAG dataset, **+Compr** reduces the original time ratio of 69% to 42%, while **+Cache** further reduces it to 29%. Finally, when stacking up **+Fusion**, this time ratio slightly increases to 30%. This is because **+Fusion** does not reduce the data transferring volume, instead, it shortens the training computation time cost.

6 RELATED WORKS

PaGraph [24] is the first work that incorporates on-GPU cache into GNN training. GNNLab [44] extends this idea by incorporating pre-sampling strategies, calculating the sampling probabilities for each node before training, and caching the features of nodes with the highest probabilities. DUCATI [45] and Legion [35] caches both graph structure and features directly on the GPU. Despite of improvements, they demonstrate limited efficacy on graphs exceeding the GPU memory capacity, as the proportion of features that can be cached diminishes with increasing graph size, and data retrieval from the host memory still incurs significant overhead.

Global Neighborhood Sampling [8] prioritize nodes present in the cache for cache hit rate enhancement. GNNAutoScale [10] optimizes forward computations by utilizing previously computed embeddings rather than recalculating neighbor embeddings in each iteration. However, they both require to change GNN models, potentially hurting accuracy, and impose high complexity and overhead when dealing with large-scale graphs.

There have been some initial attempts to employ various quantization methods in GNNs. However, none of these attempts have focused on eliminating data processing bottlenecks in sampling-based GNN training. For instance, Bi-GCN[41] and VQ-GNN[7] employ model quantization to reduce GPU memory usage and computational overhead.

Degree-Quant[37], SGQuant[9], A2Q[48], and AdaQP[40] target full graph training and assume all the datasets are already stored on the GPU. In this case, there is no data loading bottleneck, instead, there is a cross-GPU message passing bottleneck. Therefore, these approaches utilize scalar compression methods to accelerate full graph training by compressing messages, where compression and decompression occur in the GPU for each iteration at runtime. Our work significantly differs from them as follows. First, these quantization methods and full graph training are not scalable w.r.t large graphs. In fact, the largest graph dataset in AdaQP is no more than 4 GB. Second, our attempt to apply them to sampling-based GNN training points out the runtime quantization overhead is extremely high. Even worse, they rely on runtime computation states, precluding the opportunities for being done offline. Third, it is challenging to use vector quantization in full graph training as it would impose much higher overhead than scalar quantization. Thus, these methods exhibit limited compression ratios (usually 8 or 16, up to 32), making it hard to address the feature loading bottleneck experienced by sampling-based GNN training.

C-SAW [28], NextDoor [19], GNNLab [44] and gSampler [13] accelerate graph sampling using GPUs, demanding higher data loading pressure. Fortunately, F^{2}CGT can addresses this challenge.

7 CONCLUSION

F^{2}CGT employs feature compression to address data processing bottlenecks in single-machine and distributed GNN training. It introduces a GNN tailored compression approach to balance compression ratio and model accuracy. With highly optimized and accelerated compression and decompression, as well as the collaboration with re-designed on-GPU cache sub-system, F^{2}CGT significantly outperforms the SOTA baselines across two popular GNN models and four datasets, where three are large-scale graphs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work is supported in part by the National Natural Science Foundation of China under Grant No. 62141216, the University Synergy Innovation Program of Anhui Province under Grant No.: GXXT-2022-045, the 111 Project 2.0 under Grant No. BP0719016, and Guangdong OPPO Mobile Telecommunications Corp., Ltd.

REFERENCES

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2021. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–38.

[2] David M Allen. 1971. Mean square error of prediction as a criterion for selecting variables. *Technometrics* 13, 3 (1971), 469–475.

[3] Anonymous. 2024. F^{2}CGT open source code. https://github.com/gpzlx1/F2CGT/. [Online; accessed July-2024].

[4] Anonymous. 2024. F^{2}CGT supplemental material. https://github.com/gpzlx1/F2CGT-supplemental. [Online; accessed July-2024].

[5] NVIDIA Corporation. 2023. NVIDIA CUDA Unified Addressing. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__UNIFIED.html. [Online; accessed July-2024].

[6] Team DGL. 2023. DGL Homepage. https://www.dgl.ai/. [Online; accessed July-2024].

[7] Mucong Ding, Kezhi Kong, Jingling Li, Chen Zhu, John Dickerson, Furong Huang, and Tom Goldstein. 2021. VQ-GNN: A universal framework to scale up graph neural networks using vector quantization. *Advances in Neural Information Processing Systems* 34 (2021), 6733–6746.

[8] Jialin Dong, Da Zheng, Lin F Yang, and George Karypis. 2021. Global neighbor sampling for mixed CPU-GPU training on giant graphs. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 289–299.

[9] Boyuan Feng, Yuke Wang, Xu Li, Shu Yang, Xueqiao Peng, and Yufei Ding. 2020. Sgquant: Squeezing the last bit on graph neural networks with specialized quantization. In *2020 IEEE 32nd international conference on tools with artificial intelligence (ICTAI)*. IEEE, 1044–1052.

[10] Matthias Fey, Jan E Lenssen, Frank Weichert, and Jure Leskovec. 2021. Gnnautoscale: Scalable and expressive graph neural networks via historical embeddings. In *International Conference on Machine Learning*. PMLR, 3294–3304.

[11] Chen Gao, Yu Zheng, Nian Li, Yinfeng Li, Yingrong Qin, Jinghua Piao, Yuhan Quan, Jianxin Chang, Depeng Jin, Xiangnan He, et al. 2023. A survey of graph neural networks for recommender systems: Challenges, methods, and directions. *ACM Transactions on Recommender Systems* 1, 1 (2023), 1–51.

[12] Ruiqi Gao, Tianle Cai, Haochuan Li, Cho-Jui Hsieh, Liwei Wang, and Jason D Lee. 2019. Convergence of adversarial training in overparametrized neural networks. *Advances in Neural Information Processing Systems* 32 (2019).

[13] Ping Gong, Renjie Liu, Zunyao Mao, Zhenkun Cai, Xiao Yan, Cheng Li, Minjie Wang, and Zhuozhao Li. 2023. gSampler: General and Efficient GPU-based Graph Sampling for Graph Learning. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 562–578. https://doi.org/10.1145/3600006.3613168

[14] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. 2014. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115* (2014).

[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[16] Jared Heinly, Enrique Dunn, and Jan-Michael Frahm. 2012. Comparative evaluation of binary features. In *European Conference on Computer Vision*. Springer, 759–773.

[17] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. *arXiv preprint arXiv:2103.09430* (2021).

[18] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[19] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *Proceedings of the sixteenth European conference on computer systems*. 311–326.

[20] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).

[21] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

[22] Xiao Li, Li Sun, Mengjie Ling, and Yan Peng. 2023. A survey of graph neural network based recommendation in social networks. *Neurocomputing* 549 (2023), 126441.

[23] Xiaofan Lin, Cong Zhao, and Wei Pan. 2017. Towards accurate binary convolutional neural network. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 344–352.

[24] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 401–415.

[25] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2674–2683.

[26] Peter Lindstrom and Martin Isenburg. 2006. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics* 12, 5 (2006), 1245–1250.

[27] Shie Mannor, Dori Peleg, and Reuven Rubinstein. 2005. The cross entropy method for classification. In *Proceedings of the 22nd international conference on Machine learning*. 561–568.

[28] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S. Li, and Hang Liu. 2020. C-SAW: A Framework for Graph Sampling and Random Walk on GPUs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00060

[29] Team PyG. 2023. PyTorch Geometric. https://pyg.org/. [Online; accessed July-2024].

[30] Team PyTorch. 2023. Pytorch Homepage. https://pytorch.org/. [Online; accessed July-2024].

[31] Manon Réau, Nicolas Renaud, Li C Xue, and Alexandre MJJ Bonvin. 2023. DeepRank-GNN: a graph neural network framework to learn patterns in protein–protein interfaces. *Bioinformatics* 39, 1 (2023), btac759.

[32] Pau Rodríguez, Miguel A Bautista, Jordi Gonzàlez, and Sergio Escalera. 2018. Beyond one-hot encoding: Lower dimensional target embedding. *Image and Vision Computing* 75 (2018), 21–31.

[33] T. Konstantin Rusch, Michael M. Bronstein, and Siddhartha Mishra. 2023. A Survey on Oversmoothing in Graph Neural Networks. arXiv:2303.10993 [cs.LG]

[34] Amazon Web Services. 2023. Amazon EC2 G4 Instances. https://aws.amazon.com/ec2/instance-types/g4/. accessed, July-2024.

[35] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, and Fei Wu. 2023. Legion: Automatically Pushing the Envelope of Multi-GPU System for Billion-Scale GNN Training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 165–179. https://www.usenix.org/conference/atc23/presentation/sun

[36] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, and Fei Wu. 2023. Legion: Automatically Pushing the Envelope of Multi-GPU System for Billion-Scale GNN Training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 165–179. https://www.usenix.org/conference/atc23/presentation/sun

[37] Shyam A Tailor, Javier Fernandez-Marques, and Nicholas D Lane. 2020. Degree-quant: Quantization-aware training for graph neural networks. *arXiv preprint arXiv:2008.05000* (2020).

[38] SNAP Team. 2023. Friendster social network and ground-truth communities. https://snap.stanford.edu/data/com-Friendster.html. accessed, July-2024.

[39] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[40] Borui Wan, Juntao Zhao, and Chuan Wu. 2023. Adaptive message quantization and parallelization for distributed full-graph gnn training. *Proceedings of Machine Learning and Systems* 5 (2023).

[41] Junfu Wang, Yunhong Wang, Zhen Yang, Liang Yang, and Yuanfang Guo. 2021. Bi-gcn: Binary graph convolutional network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1561–1570.

[42] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. 2020. Microsoft academic graph: When experts are not enough. *Quantitative Science Studies* 1, 1 (2020), 396–413.

[43] Bernard Widrow, Istvan Kollar, and Ming-Chang Liu. 1996. Statistical theory of quantization. *IEEE Transactions on instrumentation and measurement* 45, 2 (1996), 353–361.

[44] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 417–434.

[45] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2023. DUCATI: A Dual-Cache Training System for Graph Neural Networks on Giant Graphs with the GPU. *Proc. ACM Manag. Data* 1, 2, Article 166 (jun 2023), 24 pages. https://doi.org/10.1145/3589311

[46] Zhe Zhang, Ziyue Luo, and Chuan Wu. 2023. Two-level Graph Caching for Expediting Distributed GNN Training. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*. 1–10. https://doi.org/10.1109/INFOCOM53939.2023.10228911

[47] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE Computer Society, Los Alamitos, CA, USA, 36–44. https://doi.org/10.1109/IA351965.2020.00011

[48] Zeyu Zhu, Fanrong Li, Zitao Mo, Qinghao Hu, Gang Li, Zejian Liu, Xiaoyao Liang, and Jian Cheng. 2023. A2Q: Aggregation-Aware Quantization for Graph Neural Networks. *arXiv preprint arXiv:2302.00193* (2023).